Compliance and Data Lifecycle Management in Databases and Backups

Nick Scope[1], Alexander Rasin[1](✉), Ben Lenard[1], and James Wagner[2]

[1] DePaul University, Chicago, IL 60604, USA
nscope52884@gmail.com, arasin@cdm.depaul.edu, blenard@anl.gov
[2] The University of New Orleans, New Orleans, LA 70148, USA
jwagner4@uno.edu

Abstract. From the United States' Health Insurance Portability and Accountability Act (HIPAA) to the European Union's General Data Protection Regulation (GDPR), there has been an increased focus on individual data privacy protection. Because multiple enforcement agencies (such as legal entities and external governing bodies) have jurisdiction over data governance, it is possible for the same data value to be subject to multiple (and potentially conflicting) policies. As a result, managing and enforcing all applicable legal requirements has become a complex task. In this paper, we present a comprehensive overview of the steps to integrating data retention and purging into a database management system (DBMS). We describe the changes necessary at each step of the data lifecycle management, the minimum functionality that any DBMS (relational or NoSQL) must support, and the guarantees provided by this system. Our proposed solution is 1) completely transparent from the perspective of the DBMS user; 2) requires only a minimal amount of tuning by the database administrator; 3) imposes a negligible performance overhead and a modest storage overhead; and 4) automates the enforcement of both retention and purging policies in the database.

Keywords: Databases · Privacy Compliance · Retention · Purging

1 Introduction

Organizations are subject to a variety of data management rules for how data must be archived, preserved, or destroyed. As new legislation is passed, these requirements are becoming more expansive and more strictly enforced. For example, Europe's General Data Protection Regulation (GDPR) privacy rules extend to organizations with customers in Europe (even when the organization is based outside of Europe). Organizations that fail to adhere to these policies risk their customers' privacy and are subject to potentially large fines. Thus, databases must incorporate the features and functionality necessary to remain compliant.

For purposes of this paper, we define *policy* as the set of rules an organization must follow with respect to data preservation and destruction. These policies can be the result of internal requirements, other business partners, or government agency mandates. Failure to comply with these policies could result in large fines, a loss of customers, and an irrecoverable breach of customer data privacy.

Although current industry tools (see Sect. 2) offer some important data governance capabilities, database management systems (DBMS) must be updated to support compliance in data storage. DBMSes do not currently include native retention or purging functionality that can be applied at record level. Google, Amazon, Oracle, and IBM all offer various object-storage compliance functionality for their remote storage. These all use date-criteria for defining policies timelines, but none of these offer tuple, cell, or value based policy enforcement in a database. Instead, objects are placed into "buckets" and policies are applied at the bucket-level. Because databases contain intermixed records which are subject to different policies, applying the policy at the bucket level risks non-compliance with one policy at the cost of another.

However, DBMS storage (relational or NoSQL) is much more complex and fine-grain, representing the data at individual record and value level. Currently, with respect to databases, organizations are forced to create ad-hoc solutions to meet policy compliance requirements; these solutions are typically developed by either re-purposing other existing tool functionality or manually performing the steps to enforce compliance.

Governance policies depend on multiple factors and can be surprisingly complex. The Office of the National Coordinator for Health Information Technology provides a summary overview with examples for how many states in the United States have their own requirements for retaining and destroying healthcare data [30]: Oregon requires hospitals to retain all records for 10 years after the date of the last discharge; Hawaii requires the full medical record history to be retained for 7 years after the last data entry. Adding to the complexity, the data of minors and adults can be governed by different policies. For example, in North Carolina, hospitals are required to retain adult patients data for 11 years following discharge, while the data of patients who are minors (at the time of record creation) must be retained until the patient's 30^{th} birthday. Thus, the policy expiration must reference patient's date of birth, with different rows or columns of a database table governed by different requirements.

Adding to the complexity, database administrators must consider the possible conflict between multiple requirements (e.g., retention versus destruction of the same data item). For example, GDPR's Article 17 requires that an organization purge personal data "the personal data are no longer necessary in relation to the purposes for which they were collected [5]", but if the same data item was pertinent to an impending or an ongoing lawsuit, an organization must retain the data until it is no longer required to be retained (i.e., the lawsuit has been resolved). Therefore, any organization relying on manual solutions for their compliance must consider the high labor cost of enforcing compliance.

1.1 System Overview

Ataulla et al. [9] first proposed the idea of defining data governance policies through a SQL query (see Sect. 4.1) as a first step towards native DBMS pol-

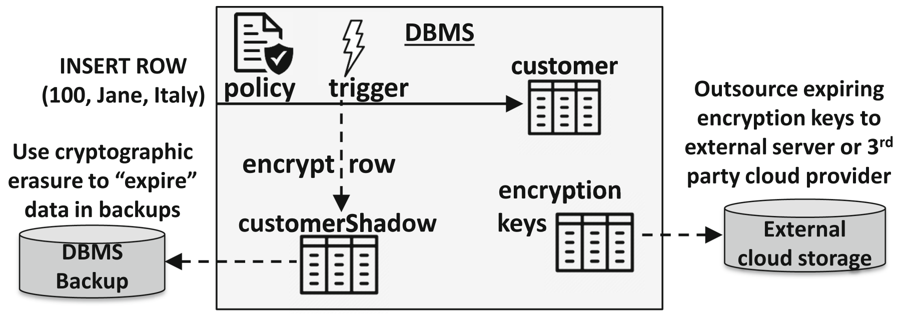
INSERT ROW
(100, Jane, Italy)
DBMS
policy
trigger
customer
encrypt row
customerShadow
encryption keys
Use cryptographic erasure to "expire" data in backups
DBMS Backup
Outsource expiring encryption keys to external server or 3rd party cloud provider
External cloud storage

Fig. 1. Data lifecycle workflow changes in a DBMS to support data purging policies. Retention policies will use triggers and defined policies to retain data in an additional `customerArchive` table

icy support. Scope et al. [27] proposed leveraging DBMS triggers (natively supported by all major database vendors) and revising the backup process to support policy-based data purging. Scope et al. [26] also prototyped the same strategy in the context of NoSQL (MongoDB) databases. In this paper, we present and evaluate an end-to-end approach to offer a native support for data governance (retention and purging) in relational and NoSQL DBMSes.

Figure 1 summarizes the integration of our data purging mechanism steps into a DBMS (retention mechanism details are not pictured). Policies are defined with database queries, SQL or NoSQL, such as "rows inserted into a customer table must be retained for a duration of 5 years". Each inserted (or updated) row is checked by a trigger against applicable purging policies, if any. The values covered by purging requirement are encrypted with a corresponding policy-based key, and inserted into the `customerShadow` table (an encrypted counterpart copy of the original `customer` table).

The `customerShadow` table is backed up instead of backing up the `customer` table, to enable "remote" erasure by destroying the corresponding key upon policy expiration. In order to fully satisfy purging requirements, the database must also securely delete encryption keys from backups [22]. Towards that end, the encryption key table is backed up separately with an independent storage service, to ensure that keys are expunged upon expiration. The encryption key table is itself encrypted to minimize the impact of a potential data breach. However, we note that the mechanisms described here are not designed to be a security solution but are a governance compliance mechanism. Thus, if the encryption keys were somehow compromised (or inadvertently copied), this framework can re-create a new encryption key table (and underlying keys) and re-encrypt the backups and shadow tables. This would not address the data theft, but once all of the data is encrypted with new keys, it would restore data storage compliance.

Our corresponding retention mechanism (not pictured in Fig. 1) checks deleted rows for values that are currently protected by retention policy. Such values are stored in an archive table and purged through the same means (see Sect. 4.1).

In sum, the contributions in this paper are:

– Defining the current state of privacy compliance functionality in databases
– Outline an external encryption key management system that guarantees data retention compliance in a DBMS
– Implementing the proposed framework for both relational and NoSQL JSON databases and evaluating the performance for daily use, backups, and restores

2 Related Work

Kamara and Lauter [16] concluded that using cryptography can improve privacy protections when using remote storage. Furthermore, their research has shown that erasing an encryption key can be a means to rendering remote data irrecoverable. We leverage cryptographic erasure mechanism to remotely purge database values to ensure data privacy purging compliance. Kamara and Lauter's work does not discuss how to manage encryption keys or to apply them at a fine-grained level necessary for compliance.

Reardon et al. provided a extensive overview of secure deletion [22]. The authors defined three user-level approaches to secure deletion: 1) execute a secure delete feature on the physical medium 2) overwrite the data before unlinking or 3) unlink the data to the OS and fill the empty capacity of the physical device's storage. Their methods require the ability to directly interact with the physical storage device, which may not be possible for all database backups. Offline backups (e.g., backup tapes in a warehouse) are still subject to purging and retention policies. Thus, destroying (either physically or with a complete deletion wipe) an entire backup to guarantee purging compliance, the destruction would come with sacrificing retention compliance.

Scope et al. [27] presented a generalized data purging workflow which supports "remote" destruction of expired data (e.g., inaccessible records stored in a backup) in a relational database via cryptographic erasure. Encryption keys are chosen based on the purging duration and policy; values not subject to purging are stored without encryption. When the purge criteria has been met, the corresponding encryption key is deleted, rendering all encrypted data permanently irrecoverable (i.e., purged). Additionally, research was conducted on only purging compliance in NoSQL JSON databases [26]. Neither paper addressed how to guarantee retention compliance while implementing purging functionality.

Scope et al. [25] later expanded the previous work to incorporate functionality that simultaneously considered both retention and purging policies. Although these papers did leverage encryption, they did not provide a framework to manage the encryption keys (i.e., how to store the encryption key backups). Additionally, this paper focused exclusively on relational databases. This paper aims to incorporate a compliant approach for managing the encryption keys (regardless of the database logical layout, including both relational and NoSQL).

On the industry side, Amazon S3 offers an object life-cycle management tool [8]. S3 is file-based and lacks the granularity to fully support retention and purging at the individual tuple level. Furthermore, NoSQL stores (e.g., MongoDB evaluated in this paper) also require a value-level granularity to implement data governance policies.

Google Cloud Platform (GCP) offers a similar tool to Amazon S3 by supporting file-level compliance [3]. GCP's Bucket Lock offers a retention solution which guarantees all files are protected until the *retention lock* has expired. Conversely, GCP's Object Lifecycle Management tool uses rules which trigger an automated deletion of files. Overall, real-world retention and purging policies require fine-grain destruction and retention of data which is currently not supported by current industry tools.

3 Data Governance and Compliance

Business Records are the units for organizational rules and requirements for data management. United States federal law refers to a business record broadly as any "memorandum, writing, entry, print, representation or combination thereof, of any act, transaction, occurrence, or event [that is] kept or recorded [by any] business institution, member of a profession or calling, or any department or agency of government [...] in the regular course of business or activity" [31]. In other words, business records describe any interaction or transaction resulting in new data.

Policy is any formally established rule for organizations. Policies can originate from a variety of sources such as legislation or as a byproduct of a court ruling. Companies may also establish their own internal data retention policies to protect confidential data. In practice, database administrators work with domain experts and sometimes with legal counsel to define business records and retention requirements based on the written policy. Policies can use a combination of time and external events as the criteria for data retention and destruction.

Retention is the preservation of all data subject to a policy. Retention requirements supersede the requirement to destroy data.

Purging is the permanent and irreversible destruction of data in a business record [15]. A business record purge can be accomplished by physically destroying the device, fully erasing all data on the device, or encrypting and erasing the decryption key (although the ciphertext still exists, destroying the decryption key makes it irrecoverable). If any part of a business record's data remains recoverable or accessible, then the data purge is not considered successful. If a purging policy overlaps with a retention policy, the data must not be purged until after all retention policies have expired.

Problem Statement: All encryption used by this framework is deployed with the intention of facilitating compliance and not for security purposes. Thus, all security considerations are beyond the scope of this paper. Additionally, data processing compliance (i.e., only using customer data where consent has been given for processing) is beyond the scope of this paper. Our goal is to implement automated retention and purging policy enforcement procedures during database transactions, backups, and restores, agnostic of DBMS logical layout.

4 System Overview

In this section, we describe our system that offers a comprehensive support for data governance policy compliance in DBMSes. We first describe the components that were previously proposed and then discuss changes and new components introduced as part of this paper. In this paper, we use the term *table* to refer to both a relational database table and a collection in JSON NoSQL database.

4.1 Background

The policies are defined using SQL or NoSQL queries (the idea originally pioneered by Ataullah et al. [9]). Therefore, the database could return the rows and columns that were subject to any particular policy by executing the corresponding query. For example, the following SQL query expresses a policy to retain all data from the tables `customerPayment` and `orderShipping` minimally 90 days after the payment date.

```
SELECT * FROM customerPayment NATURAL JOIN orderShipping
WHERE DATEDIFF(day, orderShipping.paymentDate,
      date_part('day', CURRENT_DATE)) < 90;
```

Each table containing data subject to retention rule has a corresponding *shadow archive table*. The shadow archive table stores data which was deleted (i.e., no longer needed by users) but that is protected by retention policy (for some duration or indefinitely). Similarly, each table with data subject to a purging rule has a corresponding *shadow table*. For records subject to purging, the record's values are encrypted, before a copy of the record is placed into the shadow tables; data not subject to purging is copied into shadow table in its original form. The shadow tables replace the original tables in backup; they also contain columns that provide a mapping to the corresponding encryption key.

For all defined policies, we store encryption keys and corresponding policies in the `policyOverview` table; the DDL (using Postgres) for the `policyOverview` table can be found below. The `policyOverview` table contains the date on which each key will be purged. Purging the key would purge all corresponding encrypted values across all of the shadow tables and shadow archive tables.

```
CREATE TABLE public.policyOverview (
   policyid integer NOT NULL,
   policy character varying(50),
   expirationDate date,
   encryptionkey character varying(50));
```

Whenever a user executes an `INSERT`, `DELETE`, or `UPDATE`, the framework determines if any of the data is subject to a retention or purging policy. Because retention takes priority over purging, data which is subject to both must be retained until the retention policy requirements have been met. During a restore, the shadow tables are restored and then loaded into the user-facing tables (e.g., `customer` is loaded from `customerShadow`). The data for which has not been purged (i.e., encryption key is still available) is decrypted. If the encryption key has been deleted due to a purging requirement, the values are restored as a `NULL`. For relational databases, if the primary key of a tuple cannot be restored, the entire tuple is deleted. With JSON NoSQL databases, when a key has been purged, all associated values are not restored.

One of the major challenges to guaranteeing compliance is the problem of handling encryption keys. Backing up the keys would interfere with being able to purge data (because database backups cannot be edited to selectively remove data). Scope et al. [27] proposed storing the encryption keys in a separate linked database to reliably support purging. However, the question of how to manage encryption keys was not considered in prior work.

4.2 Leveraging External or Third Party Servers

In order to successfully apply cryptographic erasure, we must guarantee that the deleted encryption keys have been irrecoverably erased. Otherwise, deleted encryption keys may be restored from a backup and decrypt purged data. Many industry tools (e.g., AWS S3) provide the ability to automatically "expire" objects at the file granularity, but any external storage which provides automated file-level time-based erasure would satisfy the requirements of this framework. We propose using such a system for automatic deletion of files to purge encryption keys (based on expiration date).

Because this framework depends on using external servers to backup the encryption keys, there is a risk of a server outage. The ability to access encryption keys is only needed during a restore of the `policyOverview` table (which would only occur during a database restore). Therefore, if restores are not common, the risk would be minimal and acceptable.

In instances where restores are common and must not be delayed or where high availability of backups is required, leveraging multiple external servers can be used for storing the encryption key backups in parallel. AWS, Google, and IBM all offer file-level automatic deletion at a set time [3,4,8].

4.3 Encryption Keys During the Backup and Restore Process

During the creation of the standard database backups, our framework leverages backup scripts to create the backup of the encryption keys and automatically uploads them to the external server(s) designated (using whichever scheduler an organization deploys, e.g., CRON job). These backup scripts must be revised to first upload the encryption keys to the third-party servers before executing the database backup procedure. Keys which have already been uploaded and have their automatic deletion criteria set do not need to be re-uploaded again. In our `policyOverview` table, we store the date at which an encryption key was uploaded to prevent it from being redundantly re-uploaded during future backups. Thus, only newly created keys would require being backed up in addition to the standard database backup procedure.

Because the policyOverview table is not backed up with the other tables, this framework requires some capability to backup individual table/collection spaces. In instances where an encryption key's purged date has passed, we can assume that the corresponding key has been automatically deleted (due to the system automatically removing the file based to the expiration date). All remaining encryption keys which have not been purged are downloaded from the remote server and used to decrypt their associated business records (using the original framework outlined by Scope et al. in [25]) during the restore process. Once restored, our framework moves the records which are either not encrypted or for which a decryption key is still available from the shadow tables into the active tables.

4.4 NoSQL Process Considerations

Although the functionality in this framework remains consistent between a relational database and a NoSQL JSON database, there are some additional factors to consider. For purposes of this discussion, we use terminology and commands from Postgres and MongoDB. When generating backups for a relational database, using a pg_dump command targeting the shadow tables guarantees that neither the unencrypted data nor the encryption keys is placed into standard backups (which would prohibit the keys from being purged). With MongoDB, we leverage collections during backups to limit the backup to only the shadow collections (using the command mongodump).

Any NoSQL JSON database which uses this framework must support triggers. Only MongoDB Atlas (the cloud version) offers trigger functionality, while the local (free) version of MongoDB does not currently support triggers.

With relational databases, during the restore, any tuple with a purged primary key is removed from the table. In NoSQL JSON databases this translates into removing all values when a corresponding key has been purged. If a subset of the values of a key have been purged, instead of NULLing out the keys, we simply remove the value from the corresponding key-value.

Although in Postgres we leverage PGP_SYM_ENCRYPT (a default supported module) to apply encryption, in MongoDB we utilize the ClientEncryption functionality found within the Explicit Encryption framework. Our proposed framework requires any database to support some level of encryption functionality which can be incorporated into a trigger.

5 Experiments

We implemented and evaluated a prototype of our framework with the active tables in Fig. 2. This schema reflects only the tables needed for the policies that we define and does not show all tables in the database schema. In practice, we expect most policies to cover data in one or two tables/collections. Other tables which do not directly apply to a policy will not impact policy-enforcing performance. We demonstrate that our approach can be implemented without

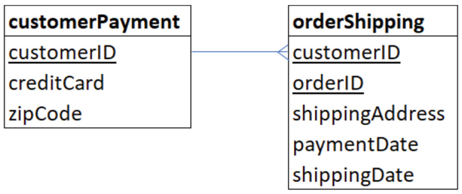

Fig. 2. Tables used in our policy definitions and experimental evaluation

changing the original (user-facing) database schema and by extending backup procedures using only natively available DBMS backup functionality.

5.1 Experimental Setup

Hardware: We used a server with dual Intel Xeon E5645, each with 6 physical cores and Hyper Threading enabled, 64 GB of ram, and an SSD for storage. The server was running CentOS 8 Stream x86_64 with Kernel Virtual Machine [17] (KVM) as the hypervisor software. We used two Virtual Machines (VMs) to carry out the experiments; since a majority of database interactions operate in a client-server model, we deployed two independent VMs to represent client and server. Both VMs were built with CentOS 8 Stream x86_64, Postgres 14.5, MongoDB 4.1, 1 x vNIC and a 25 GB QEMU copy-on-write [6] (QCOW2) file on an SSD. The client VM has 4 GB of RAM and 4 vCPUs and the server VM was allocated 8 GB of RAM and 4 vCPUs. QCOW2 file was partitioned into: 350 MB/boot, 2 GB swap space, with the remaining storage used for the / partition, using standard partitioning and ext4 file system. Only these two VMs were running on the hypervisor to minimize runtime fluctuations.

Policies and Keys: We created one retention and one purging policy; both policies covered all columns in tables from Fig. 2. We then generated 30,000 business records which approximately equally fell under 1) neither policy, 2) only the retention policy, 3) only the purging policy, or 4) both policies.

In Sect. 5.2, we evaluate framework performance overhead using a real-world simulated query workload on a local database. We use synthetic data and the additional generated encryption keys (total of 21 encryption keys) in our experiments. In Sect. 5.3 we analyze the performance overhead of our framework during the backup and restore of a relational and NoSQL database. Our experiments confirmed the framework enforces retention and purging compliance.

5.2 Query Overhead Imposed by the Framework

Relational Databases: `SELECT`s do not incur any retention or purging overhead in our framework. Because real-world data warehouse workloads are typically 90% `SELECT`s [14], in practice the compliance overhead would apply a relatively small fraction of queries. To mirror data warehouse workloads reported by Hsu

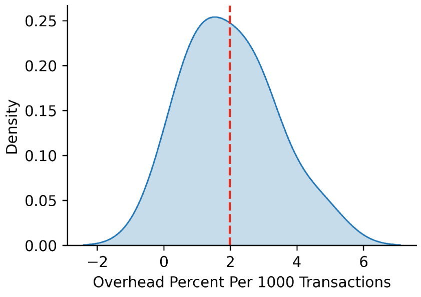

Fig. 3. Overhead of our framework during a simulated query workload

et al. [14], our query workload consisted of 9,000 (90%) `SELECT` queries, 700 (7%) `UPDATE` queries, and 300 (3%) `DELETE` queries. Because our framework runs the same process for `UPDATE`s and `INSERT`s, we use `UPDATE`s for performance evaluation.

In order to measure the runtime overhead, we ran an identical query workload on two identical databases, one with our framework enabled and one without any additions. Different types of queries were mixed in randomly in our workload; we measured the elapsed time after each 1,000 query transactions. We manually verified that the the encryption keys and archival processes were correctly applied to the data after running the simulated workload. The overall overhead distribution can be seen in Fig. 3. On average, our proposed framework had a 2% overhead compared to the database without the compliance framework.

Our workload replicates the average expected query distribution observed by Hsu et al. [14]. In practice, the overhead will depend on the policy sizes, the frequency which queries trigger a policy action, and the types of queries run. The evaluation of each of the such factors is beyond the scope of this paper.

The closest related research conducted by Ataullah et al. [9] uses triggers to determine whether or not an `UPDATE` or `DELETE` would violate a retention policy. In instances where a query would result in non-compliance, the query is blocked. Thus, both our framework and the research by Ataullah et al. require a trigger initiating and the code required to determine whether or not a query would result in a compliance violation. We have a small overhead of archiving data compared to their solution of blocking a query; this results in the trade-off of not having to adjust queries at the cost of automatic archiving overheads.

NoSQL Databases: In this paper, we focus on evaluating local (non-cloud) databases to minimize the number of factors outside of our control that may affect performance. MongoDB only supports triggers in a cloud-based version; thus we do not evaluate the query overhead performance in this paper. Scope et al. [26] verified the functionality of using triggers and cryptographic erasure to support purging in MongoDB Atlas (cloud-based version of MongoDB).

5.3 Backup and Restore Overhead Imposed by the Framework

In this experiment, we evaluate the cost of backing up and restoring the encryption keys for our framework. We discuss the overhead cost of backing up a single

Table 1. Backup and Restore File Sizes (in bytes)

File	Relational	NoSQL JSON
Full Database	64,011,916	149,151,434
Single Encryption Key File	2,810	3,437
Separate Encryption Key (21 Files)	72-73	165

key file (i.e., the `policyOverview` table) versus backing up each key independently, in addition to executing a full backup/restore from a local file. We use the encryption keys and data described in Sect. 5.2 (where 21 encryption keys were generated). Table 1 summarizes the file sizes in this experiment.

This experiment evaluates the additional overhead cost of backing up the encryption keys to an external server. To measure the upper-bound overhead of our framework, we disabled key caching during the restore.

Relational Backup and Restore: We ran 10 backups with and without our framework enabled to evaluate the overhead cost of backing up a single key file (i.e., backing up the entire encryption key table as a single file) and all 21 encryption keys independently. The observed overheads of these backups are shown in Figs. 4 and 5 (with the storage costs outlined in Table 1). For the single key file backup, the performance overhead was 72%, and for backing up the individual 21 keys the overhead was 100%.

We then evaluated the cost of restoring a relational database, introducing the additional step of restoring of the encryption keys from an external server. Figure 6 presents the overhead of restoring a single file, while Fig. 7 shows the overhead of restoring all 21 keys. For a single key file, the average restore overhead was approximately 105%. For the full restore of individual 21 keys, the overhead was 92%.

NoSQL Backup and Restore: To verify our framework backup and restore works with a NoSQL JSON logical layout, we performed the same evaluation

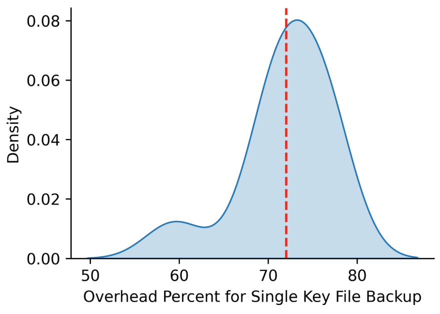

Fig. 4. Relational single-file backup overhead

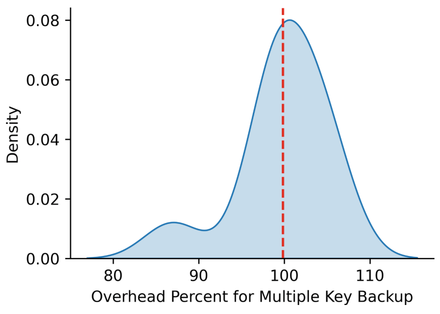

Fig. 5. Relational individual 21-file backup overhead

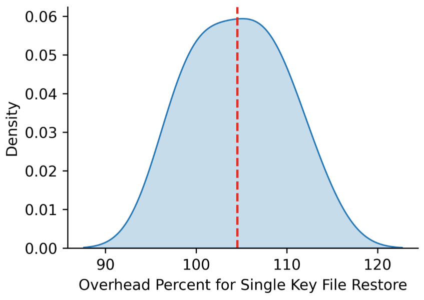

Fig. 6. Relational single-file restore overhead

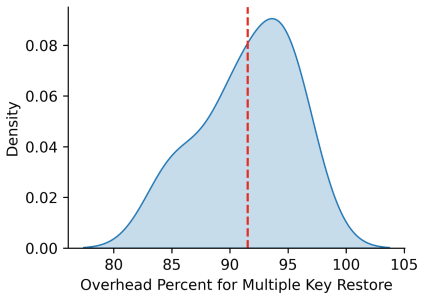

Fig. 7. Relational 21-file restore overhead

(using the same data in NoSQL JSON collections) with a local MongoDB. Thus, the *shadow collections* are backed up using standard MongoDB backup procedures (`mongodump`) and our `encryptionKeys` collection is backed up to external servers. As with the relational database, we evaluate the overhead for backing up and restoring a single encryption key file as well as for separately backing up and restoring all of the keys generated during our analysis of the overhead.

Figures 8 and 9 provide an overview of the overhead incurred by backing up the encryption keys in MongoDB. The average overhead of a single key file was 105%, and for backing up all of the 21 keys separately was 115%.

Figures 10 and 11 summarize the overhead impact of the framework on the restore process. Our framework adds a 107% overhead to restoring a single key file from a remote server; when restoring each key independently, the overhead is increased to 114%.

5.4 External Backup and Restore Performance Considerations

As the industry moves to the cloud environment, the backup and restore process can be influenced by a number of factors, including the internet connection speed, hypervisor load, or disk type. In our experiments, we used an external

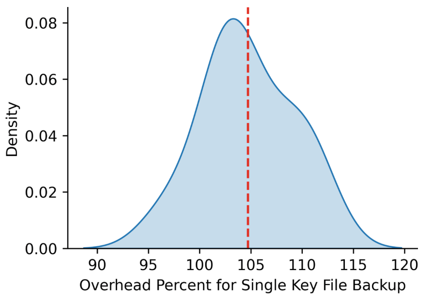

Fig. 8. NoSQL 1-file backup overhead

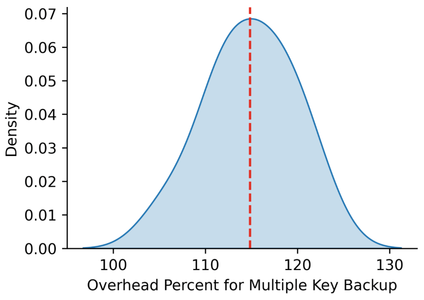

Fig. 9. NoSQL 21-file backup overhead

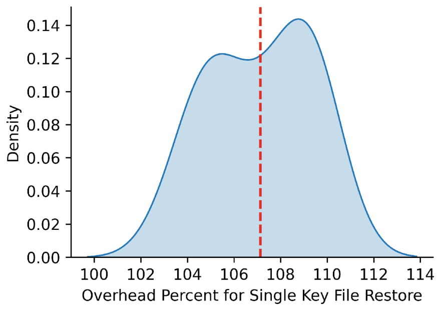

Fig. 10. NoSQL 1-file restore overhead

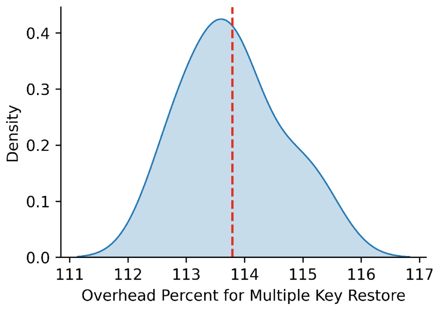

Fig. 11. NoSQL 21-file restore overhead

server offsite from our university as described in Sect. 5.1. We also used a NAS appliance that was located within the University, so the server and NAS are roughly 60 miles apart. The network connection from the server to the Internet is 1Gbps symmetrical and the NAS appliance also has a 1Gbps symmetrical connection. However, due to the nature of the Internet the speed for which these files were uploaded or downloaded can fluctuate based on Internet congestion. Moreover, the number of files that are uploaded or downloaded can influence the duration of the transfer as well as the load of the NAS device.

We use a simple server running KVM, but a cloud service would introduce additional complexity. For example, AWS offers different tiers, some tiers share the hypervisors resources while other tiers are dedicated. Larger organizations with dedicated on-premise hardware running this framework may be able to leverage their existing architecture resulting in a lower overhead percent. Furthermore, a higher ratio of business records to encryption keys will result in a smaller overhead during backup and restore.

Almost all modern file-transfer services require authentication (which is part of AAA [7]), and our SFTP connection authentication incurred a time cost. In other words, uploading the keys to a separate external server requires additional time even if the volume of the data remains the same. Moreover, the authentication time for a connection could vary based on a multitude of factors ranging from the load of the target server to the load and response time of the authentication service. Thus, the performance of backup and restore processes can fluctuate based on a multitude of factors outside of the control of our framework and systems.

6 Discussion

6.1 Third Party Server Vendor Considerations

For cryptographic erasure to fully satisfy purging requirements, all pertinent encryption keys must be rendered irreversibly irrecoverable. Many remote storage options do not offer the ability to implement a Secure Deletion process on specific files. Amazon's documentation [1,33] states, "When an object is deleted from Amazon S3, removal of the mapping from the public name to the object starts immediately [...] Once the mapping is removed, there is no external access to the deleted object. That storage area is then made available only for write operations and the data is overwritten by newly stored data [...] AWS Backup randomizes its deletions within 8 h following recovery point expiration to maintain performance."

GCP takes a similar approach [2,3], where the documentation states "[the] Google backup cycle is designed to expire deleted data within data center backups within six months of the deletion request. Deletion may occur sooner depending on the level of data replication and the timing of Google's ongoing backup cycles [...] After the data is marked for deletion, an internal recovery period of up to 30 days may apply depending on the service or deletion request."

Thus, the secure deletion guarantees provided would depend on the vendor. If one were to use an "in-house" external server for encryption key storage, a guaranteed secure deletion could be implemented. Organizations must balance these considerations to decide between a vendor or an in-house service.

6.2 Managing the Size of the Archive Tables

Although this framework supports the restoration of all archived data, in practice, many organizations may choose to limit the amount of data in the archive outside of backups. Because data in the archive is not expected to be used regularly (otherwise it would not be in an archive), many organizations may not want to use their high performance storage on data which is not frequently accessed. Thus, this framework supports a parameter which limits the restore of data from the archive backups during the restore process to a specified time range.

With larger databases, we would consider partitioning the archive tables to enable more efficient deletions. For example, we could partition tables on a policy date field and make a new partition every month. Before we drop the month that has expired, we would export it to a non-proprietary format for easy retrieval at a later date. When keeping records for 10+ years, it is simpler to recover the data in a non-proprietary format since versions of software, hardware, and operating systems change over the years. Many organizations use CSV, JSON, XML, or even HDF5 file formats for long term software-independent storage.

6.3 Reclaiming Unused Storage Space

After marking the data for deletion, databases will flag the row for deletion without actually purging the data from disk; in the case of an `UPDATE`, the database operation will often mark the old row for deletion and insert a new (updated) row. In that case, the row's pre-update data will still exist in the underlying database pages. This old data remains on disk until the RDBMS reuses the tablespace space or a reorganization of the tablespace purges the old data from database storage [18]. The challenge with management or purging such deleted data are due to DBMSes not providing tools or mechanisms to monitor or modify their internal storage.

6.4 Concerns with Forensically Recoverable Data

Deleted data that still remains on a storage medium but is no longer referenced by a file system or a DBMS can still be reconstructed using a variety of forensic methods (e.g., [10,24,34,35]). Although this paper addresses how to manage encryption keys used for purging data across backups, we consider forensically recoverable data to be beyond the scope of this paper. Lenard et al. [18] analyzed how various types of databases and their defrag options are able to remove the surviving deleted data from backups. We therefore recommend regularly running a defrag on a database to expedite the process of clearing out deleted data from database pages, particularly before these pages are placed in a backup.

There are data sanitization techniques that seek to destroy deleted data so that it can no longer be forensically reconstructed. Although Lenard et al. [18,19] investigated the data left forensically recoverable in different parts of database system and Wagner et al. [36] developed the API to interact with low-level database storage, there are currently no solutions available to sanitize database storage. Most research and tools for data sanitization involve overwriting blocks at the disk level (e.g., [11–13,23]) and cannot overwrite individual database records. SQLite is the only DBMS that supports data sanitization with the `secure_delete` setting [28], which is disabled by default due to a negative impact on performance. If enabled, `secure_delete` explicitly overwrites deleted data with zeros. Stahlberg et al. presented a similar method for MySQL [29].

Although the laws do not explicitly detail the technical steps of comprehensive data destruction, this level of data destruction is typically described by individual organizations or government agencies (e.g., NSA [21], NIST [20], or IRS [32]). Data sanitization is a problem that we consider to be outside the scope for this paper; our encryption protects data covered by purging policies, even from forensic recovery, but a more general data sanitization approach may compliment the work in this paper.

7 Conclusion

Data management research must continue to address and refine the support for database compliance functionality with respect to customer privacy. Although some research has begun to address current shortcomings, an increasing proliferation of new rules, requirements, and complexity will result in increased compliance pressures. Current purging and retention compliance support is limited to either coarse-grained (i.e., file-level) applications or does not consider both retention and purging simultaneously when enforcing compliance policies. Fine-grained compliance functionality must be researched and implemented in database systems to automatically enforce compliance. This paper outlines a comprehensive compliance support framework that implements retention and purging support throughout databases and their backups; our experiments demonstrate that our framework can guarantee compliance requirements with an acceptable performance overhead and with minimal additional infrastructure requirements.

References

1. Amazon web services: Overview of security processes. https://d1.awsstatic.com/whitepapers/aws-security-whitepaper.pdf
2. Data deletion on google cloud documentation. https://cloud.google.com/docs/security
3. GCP object lifecycle management. https://cloud.google.com/storage/docs/lifecycle
4. IBM cloud object storage - overview, https://www.ibm.com/cloud/object-storage
5. Regulation (eu) 2016/679 of the European parliament and of the council (2020). Accessed June 2021. https://gdpr.eu/tag/gdpr/
6. Qcow (2022). https://en.wikipedia.org/wiki/Qcow
7. AAA (computer security) (2023). https://en.wikipedia.org/wiki/AAA_(computer_security)
8. Amazon: Aws s3 (2020). Accessed Aug 2020. https://aws.amazon.com/s3/
9. Ataullah, A.A., Aboulnaga, A., Tompa, F.W.: Records retention in relational database systems. In: Proceedings of the 17th ACM Conference on Information and Knowledge Management, pp. 873–882 (2008)
10. Carrier, B.: The sleuth kit (2011). http://www.sleuthkit.org/sleuthkit/
11. Chow, J., Pfaff, B., Garfinkel, T., Rosenblum, M.: Shredding your garbage: Reducing data lifetime through secure deallocation. In: USENIX Security Symposium, pp. 22–22 (2005)
12. Garfinkel, S.L., Shelat, A.: Remembrance of data passed: a study of disk sanitization practices. IEEE Secur. Priv. **99**(1), 17–27 (2003)
13. Gutmann, P.: Secure deletion of data from magnetic and solid-state memory. In: Proceedings of the Sixth USENIX Security Symposium, vol. 14, pp. 77–89. San Jose, CA (1996)
14. Hsu, W.W., Smith, A.J., Young, H.C.: Characteristics of production database workloads and the TPC benchmarks. IBM Syst. J. **40**(3), 781–802 (2001)
15. International Data Sanitization Consortium: Data sanitization terminology and definitions (2017). Accessed Feb 2021. https://www.datasanitization.org/data-sanitization-terminology/
16. Kamara, S., Lauter, K.: Cryptographic cloud storage. In: Sion, R., et al. (eds.) FC 2010. LNCS, vol. 6054, pp. 136–149. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14992-4_13
17. KVM. https://www.linux-kvm.org/page/Main_Page
18. Lenard, B., Rasin, A., Scope, N., Wagner, J.: What is lurking in your backups? In: Jøsang, A., Futcher, L., Hagen, J. (eds.) SEC 2021. IAICT, vol. 625, pp. 401–415. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78120-0_26
19. Lenard, B., Wagner, J., Rasin, A., Grier, J.: SysGen: system state corpus generator. In: Proceedings of the 15th International Conference on Availability, Reliability and Security, pp. 1–6 (2020)
20. National Institute of Standards and Technology: Guidelines for media sanitization (2006)
21. National Security Agency Central Security Service: NSA/CSS storage sanitization manual (2014)
22. Reardon, J., Basin, D., Capkun, S.: Sok: secure data deletion. In: 2013 IEEE Symposium on Security And Privacy, pp. 301–315. IEEE (2013)
23. Reardon, J., Capkun, S., Basin, D.: Data node encrypted file system: efficient secure deletion for flash memory. In: Proceedings of the 21st USENIX Conference on Security symposium, pp. 17–17. USENIX Association (2012)

24. Richard III, G.G., Roussev, V.: Scalpel: a frugal, high performance file carver. In: DFRWS. Citeseer (2005)
25. Scope, N., Rasin, A., Lenard, B., Heart, K., Wagner, J.: Harmonizing privacy regarding data retention and purging. In: Proceedings of the 34th International Conference on Scientific and Statistical Database Management, pp. 1–12 (2022)
26. Scope, N., Rasin, A., Lenard, B., Wagner, J., Heart, K.: Purging compliance from database backups by encryption. J. Data Intell. **3**(1), 149–168 (2022)
27. Scope, N., Rasin, A., Wagner, J., Lenard, B., Heart, K.: Purging data from backups by encryption. In: Strauss, C., Kotsis, G., Tjoa, A.M., Khalil, I. (eds.) DEXA 2021. LNCS, vol. 12923, pp. 245–258. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86472-9_23
28. SQLite: PRAGMA statements (2018). https://www.sqlite.org/pragma.html#pragma_secure_delete
29. Stahlberg, P., Miklau, G., Levine, B.N.: Threats to privacy in the forensic analysis of database systems. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 91–102. ACM, Citeseer (2007)
30. The Office of the National Coordinator for Health Information Technology: State medical record laws: Minimum medical record retention periods for records held by medical doctors and hospitals (2022)
31. United States Congress: 28 U.S. code §1732 (1948). https://www.law.cornell.edu/uscode/text/28/1732
32. U.S. Internal Revenue Service: Media sanitization methods (2017). https://www.irs.gov/privacy-disclosure/media-sanitization-methods
33. Vliet, J.V., Paganelli, F., Geurtsen, J.: (2012). https://docs.aws.amazon.com/aws-backup/latest/devguide/deleting-backups.html
34. Wagner, J., Rasin, A., Grier, J.: Database forensic analysis through internal structure carving. Digit. Investig. **14**, S106–S115 (2015)
35. Wagner, J., Rasin, A., Grier, J.: Database image content explorer: carving data that does not officially exist. Digit. Investig. **18**, S97–S107 (2016)
36. Wagner, J., Rasin, A., Heart, K., Malik, T., Grier, J.: Df-toolkit: interacting with low-level database storage. Proc. VLDB Endowment **13**(12) (2020)