Purging Data from Backups by Encryption

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

[1] DePaul University, Chicago, IL 60604, USA
[2] The University of New Orleans, New Orleans, LA 70148, USA

Abstract. Data retention laws establish rules intended to protect privacy. These define both retention durations (how long data must be kept) and purging deadlines (when the data must be destroyed in storage). To comply with the laws and to minimize liability, companies should destroy data that must be purged or is no longer needed. However, database backups generally cannot be edited to purge "expired" data and erasing the entire backup is impractical. To maintain compliance, data curators need a mechanism to support targeted destruction of data in backups.

In this paper, we present a cryptographic erasure framework that can purge data from all database backups. Our approach can be transparently integrated into existing database backup processes. We demonstrate how different purge policies can be defined through views and enforced by triggers without violating database constraints.

Keywords: Purging compliance · Databases · Privacy · Encryption

1 Introduction

Efforts to protect user data privacy and give people control over their data have led to passage of laws such as the European General Data Protection Regulation (GDPR) [6] and California Consumer Privacy Act (CCPA) [11]. With the increased emphasis on proper data governance, many organizations are working to implement the data retention requirements into their databases. Laws can dictate how long data must be retained (e.g., United States Income Revenue Service tax document retention [8]), the consent required from individuals on how their data may be used (e.g., GDPR Article 6), or purging policies for when data must be destroyed (e.g., GDPR Article 17).

In this paper, we consider the problem of data purging in a database. Prior research has only considered the problems of retention and purging policies for data in an active (i.e., current instance) database [1]. Nevertheless, to fully comply with the laws mandating data purging, a system must purge data from the active database as well as from backups. Although backups are not part of the active database, they can be restored into an active database at any time.

C. Strauss et al. (Eds.): DEXA 2021, LNCS 12923, pp. 245–258, 2021.
https://doi.org/10.1007/978-3-030-86472-9_23

1.1 Motivation

A variety of factors make purging data from backups difficult. Backups may potentially be edited by 1) restoring the backup, 2) making changes in the restored database, and then 3) creating a new ("edited") backup. Outside of this cumbersome process, there is no other method of safely editing a backup. Only a full (i.e., non-incremental, see Sect. 2.2) backup can be altered in this manner. Furthermore, editing a full backup would invalidate all of its dependent incremental backups. Additionally, backups may be be stored remotely (e.g., off-site) and on sequential access media (e.g., on tape). Therefore, the ability to make changes to any data within backups is both limited and costly.

In order to solve this problem, we propose to implement data purging through cryptographic erasure [2]. Intuitively, a cryptographic erasure [2] approach encrypts the data and then purges that data by deleting decryption keys. The advantage of this approach is that it deletes the data "remotely" without having to access the backups. When a backup is restored, the irrecoverable data is purged while the recoverable and non-encrypted data are fully restored into the active database. Furthermore, this process does not invalidate partial backups.

Our framework creates *shadow tables* which contain an encrypted copy of all data subject to purging policies. These shadow tables are backed up instead of the original tables; we then use cryptographic erasure to simultaneously purge values across all existing backups. Our approach requires no changes to current backup practices and is compatible with both full and incremental backups. One challenge of implementing cryptographic erasure is in balancing different policy requirements across a relational database schema. A single row in a table may have columns subject to different retention and purging requirements.

Our framework only applies encryption to data which is updated or inserted after the purge policy is defined and does not retroactively apply encryption to the already-present data (e.g., if an existing policy is changed). Our approach focuses on addressing compliance rather than security. It will guarantee data destruction based on defined policies; thwarting a malicious insider who previously copied data or decryption keys is beyond the scope of this paper. Furthermore, purging data that remains recoverable via forensic tools is out of scope for this paper. Our contributions are:

- We outline the requirements for defining and enforcing data purge policies
- We describe an implementation (and present a prototype) for backup data purging that can be seamlessly integrated into existing DBMSes during backup and restore
- We design a key selection mechanism that balances multiple policies and retention period requirements

2 Background

2.1 Compliance Terminology

Business Record: Organizational rules and requirements for data management are defined in units of business records. 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" [4]. A business record may consist of a single document for an organization (e.g., an email message). In a database, a business record may span combinations of rows across multiple tables (e.g., a purchase order consisting of a buyer, a product, and the purchase transaction from three tables).

Policy: A policy is any formally established rule for organizations dictating the lifetime of data. Retention policies can dictate how long data must be saved while purge policies dictate when data must be destroyed. Policies can originate from a variety of sources such as legislation or a byproduct of a court ruling. Companies may also establish their own internal data retention policies to protect confidential data. In practice, database curators work with domain experts and sometimes with legal counsel to define business records and retention requirements based on the written policy.

Purging: In data retention, purging is the permanent and irreversible destruction of data in a business record [7]. A business record purge can be accomplished by physically destroying the device which stored the data, encrypting and erasing the decryption key (although the ciphertext still exists, destroying the decryption key makes it inaccessible and irrecoverable), or by fully erasing the data from all storage.

2.2 Database Backups and Types

Backups are an integral part of business continuity practices to support disaster recovery. There are many mechanisms for backing up a database [5] both at the file system level and internal to the DBMS. File system backups range from a full backup with an offline, or quiesced, database to a partial backup at file system level that incrementally backs up changed files. Most DBMS platforms provide backup utilities for both full and partial backups, which create backup in units of pages (rather than individual rows).

Some utilities provide block-level backups with either a *full* database backup or a partial backup capturing pages that changed since the last backup. Partial backups can be *incremental* or *delta*. For example, if we took a full backup on Sunday and daily partial backups and needed to recover on Thursday, database utilities would restore the full backup from Sunday and then either 1) apply delta backups from Monday, Tuesday, and Wednesday or 2) apply Wednesday's incremental backup. Because most organizations use multiple types of backups, any purging system must work on full, incremental, and delta backups [10].

2.3 Related Work

Kamara and Lauter's research has shown that using cryptography can increase storage protections [9]. Furthermore, their research has shown that erasing an encryption key can fulfill purging requirements. Our system expands on their

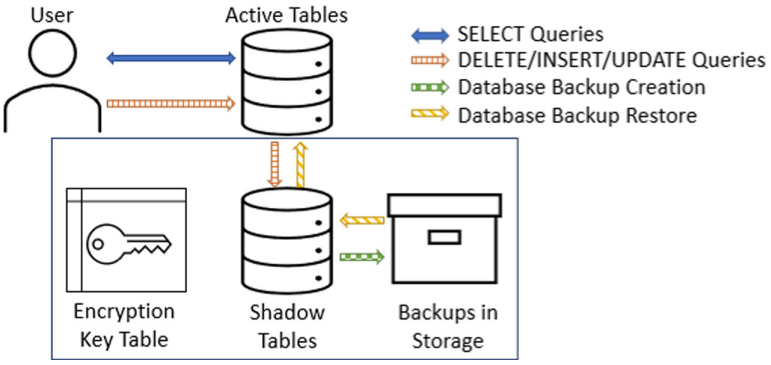

Fig. 1. Framework overview

research by using policy definitions to assign different encryption keys relative to their policy and expiration date.

Reardon et al. provided a comprehensive overview of secure deletion [12]. 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. All methods require the ability to directly interact with the physical storage device, which may not be possible for database backups in storage.

Boneh et al. used cryptographic erasure, but each physical device had a single key [2]. We introduce an encryption key assignment system to facilitate targeted cryptographic erasure of business records across all backups. In order to fully destroy the data, users must also securely delete the encryption keys used for cryptographic erasure. Reardon et al. [12] provide a summary for how to destroy encryption keys to guarantee a secure delete. Physically erasing the keys depends on the storage medium and is beyond the scope of this paper. However, unlike backups, encryption keys are stored on storage medium that is easily accessible.

Ataullah et al. described some of the challenges associated with record retention implementation in relational databases [1]. The authors proposed an approach that uses view-based structure to define business records (similar to our approach); they used view definitions to prohibit deletion of data that should be retained. Ataullah et al. only consider purging data in an active database; they did not consider how their approach would interact with backups.

3 Our Process

Our proposed framework automatically applies encryption to data that is subject to purge policy requirements whenever data are inserted or updated. An overview of this process is presented in Fig. 1. We maintain and backup a shadow (encrypted) copy of the tables; other tables not subject to purging rules are not affected. `SELECT` queries always interact with the non-encrypted database tables

(rather than shadow tables) and are not impacted by our approach. We translate (using triggers) `DELETE`, `INSERT`, and `UPDATE` queries into a corresponding operation on the encrypted shadow copy of the table. Our framework is designed to remain transparent to the user. For example, one can use client-side encryption without affecting conflicting with our data purging approach. A change in purge policy has to be manually triggered to encrypt existing data.

In our system, shadow tables are backed up instead of the corresponding user-facing tables; tables that are not subject to purging policies are backed up normally. When the shadow tables are restored from backup, our system decrypts all data except for data purged per policy. For encryption keys that expired due to a purge policy, the underlying data would be replaced with `NULL` (unfortunately, purging of data unavoidably creates ambiguity with "real" `NULL`s in the database). In cases where the entire row must be purged (due to a purged primary key), the tuple would not be restored. Evaluation of possible conflicts (e.g., purge policy on a column that is restricted to `NOT NULL`) is resolved during the policy definition step.

Our default implementation uses a table called `encryptionOverview` (with column definition shown in Table 1) to manage encryption keys. This table is marked to never be backed up to avoid the problem of having the encryption keys stored with the backup; otherwise the encryption keys could not be truly purged. In our proof-of-concept experiments, the `encryptionOverview` table is stored in the database. However, in a production system the key management tables will be stored in a separate database. Access to these tables could be established via a database link or in a federated fashion, allowing the keys to be kept completely separate from the actual data.

Our framework uses time-based policy criteria for purging, bucketed per-day by default. A bucket represents a collection of data grouped by a time range and policy that is purged together as a single unit. All data in the same bucket for the same policy uses the same encryption key. Our default bucket size is set to one day because, for most purge policies, daily purging satisfies the requirements (e.g., GDPR: Article 25 [6]). We intend to study the performance and granularity trade off (by changing bucket size) in future work. The encryption keys can be deleted by a cron-like scheduler, available in most DBMSes. However, since we intend to separate the `encryptionOverview` table from the database in production, we did not evaluate that functionality in our experiments.

Tables may contain data belonging to multiple business records; columns in a single row may be subject to different policies. In the shadow tables, each original column explicitly includes its `[column name]EncryptionID`, which serves as its encryption key identifier (chosen based on which policy takes precedence).

Table 1. encryptionOverview table column definitions

encryptionOverview	
encryptionID	Int
policy	Varchar(50)
expirationDate	Date
encryptionKey	Varchar(50)

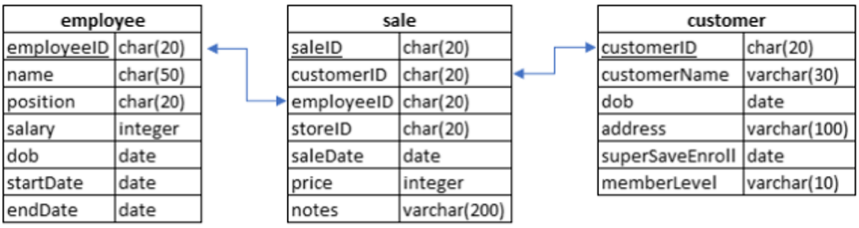

Fig. 2. Sample company schema

3.1 Defining Policies

Our method of defining purge policies uses SQL views to define the underlying business records and the purge criteria. We require defining a time-based purging period (which, at insert time, must provide at least one non-`NULL` value); if any one primary key attribute is included in a purge policy, all other columns must be included. The purge definition must also include all child foreign keys of the table to maintain referential integrity. For example, in the schema in Fig. 2, if the `customerID` in the `customer` table was included under a purge policy, both `customer.*` columns and `sale.customerID` must be included. During the restore process, a purged column value will be restored as a `NULL`. Thus, non-primary-key columns subject to a purge policy must not prohibit `NULL`s, including any foreign key columns. When all columns are purged from a row, the entire tuple will not be restored (i.e., ignored on restore).

Consider a policy for a company (Fig. 2) that requires purging all customer data where the Super Save enrollment date is over twenty years old:

```
CREATE PURGE customerPurge AS SELECT customer.*, sale.customerID
FROM customer LEFT JOIN sale ON customer.customerID = sale.customerID
WHERE datediff(year, customer.superSaverEnroll,
                date_part('year', CURRENT_DATE)) > 20;
```

In this example, the `superSaveEnroll` column will not contain `NULL`; therefore, at least one column can be used to determine the purge expiration date, satisfying our definition requirements.

3.2 Encryption Process

When a new record is inserted, we use triggers to determine if any of the columns fall under a purge policy; if so, the trigger computes the relevant policies and when the business records must be purged. For example, consider a new employee record inserted into the employee table:

```
INSERT INTO customer
   (customerID,customerName,dob,address,superSaveEnroll,memberLevel)
VALUES (1,'Johnson,Isabel','2/1/1990','Chicago','1/1/2021','Premium');
```

Under the previously defined `customerPurge` policy, Isabel Johnson's data would have a purge date of January 1, 2041. We first check if an encryption

key for this date bucket and policy already exists in the `encryptionOverview` table. If an encryption key already exists, we use it to encrypt the values covered by the purge policy; if not, a new key is generated and stored in the `encryptionOverview` table. The encrypted row and the matching encryption key ID is inserted into the `customerShadow` table. If a column is not covered by a purge policy, a value of –1 is inserted into the corresponding `EncryptionID` column. The value of –1 signals that the column has not been encrypted and contains the original value. In this example, each column in the shadow table is encrypted with the same key, but our proposed framework allows policies to be applied on a per-column basis. Therefore, our framework tracks each column independently in cases where a row is either partially covered or covered by different policies.

To support multiple purge policies, we must determine which policies apply to the new data. A record in a table may fall under multiple policies (potentially with different purge periods). Furthermore, a single value may belong to different business records with different purge period lengths. In data retention the longest retention period has priority; on the other hand, in data purging, the shortest period has priority. Therefore, we encrypt each column using the encryption key corresponding to the shortest purge period policy.

It is always possible to shorten the purging period of a policy by purging the data earlier. However, our approach does not support extending the purge period since lengthening a purge period risks violating another existing policy. Thus, if a policy is dropped, data already encrypted under that policy will maintain the original expiration date.

Continuing with our example, another policy dictates a purge of all "Premium+" customer address information ten years after their enrollment date. Because this policy applies to a subset of columns on the `customer` table, some columns are encrypted using the encryption key for `customerPurge` policy while other columns are encrypted using the `premiumPlusPurge` policy. For example, if a new Premium+ member were enrolled, the `premiumPlusPurge` policy would take priority on the address field, with remaining fields encrypted using the `customerPurge` policy key.

3.3 Encryption on Update

Similarly to `INSERT`, we encrypt all data subject to purge policy during an `UPDATE`. Normally, the updated value would simply be re-encrypted and stored in the shadow table. However, if an update changes the date and alters the applicable purge policy (e.g., changing the start or the end date of the employee), the record may have to be re-encrypted with a different key or decrypted (if purge policy no longer applies) and stored unencrypted in the shadow table. Our prototype system decrypts the primary key columns in the shadow table to identify the updated row. This is a PostgreSQL-specific implementation requirement, which may not be needed in other databases (see Sect. 4). Our system automatically deletes the original row from the shadow table and inserts the new record (with encryption applied as necessary), emulating `UPDATE` by `DELETE`+`INSERT`.

Continuing with our example, let's say Isabel Johnson is promoted to the "Premium+" level, changing the purge policies for her records. We can identify her row in the shadow table using the `customerID` primary key combined with the previously used `customerIDEncryptionID`. We would then apply the corresponding updates to encrypt the fields covered by the policy, based on the new policy's encryption key.

3.4 Purging Process

Purging is automated through a cron-like DBMS job ([3] in Postgres) that removes expired encryption keys from `encryptionOverview` with a simple delete. Our framework is designed to support purge policies and not for support of retention policies (i.e., prevent deletions before the retention period expires). Retention requires a separate mechanism, similar to work in [1,13]. Moreover, key deletion will need to be supplemented by a secure deletion of the encryption keys on the underlying hardware [2,12], guaranteeing the encryption keys are permanently irrecoverable (which is outside the scope of this paper).

3.5 Restore Process

Our framework restores the backup with shadow tables that contain encrypted as well as unencrypted values. Recall that the shadow tables include additional columns with encryption ID for each value. A –1 entry in the `encryptionID` column indicates that the column is not encrypted and, therefore, does not require decryption and would be restored as-is. Our system decrypts all values with non-expired encryption keys into the corresponding active table. For any encrypted value associated with a purged `encryptionID` our system restores the value as a `NULL` in the active table. If the entire row has been purged, the tuples would not be restored into the active table.

4 Experiments

We implemented a prototype system in PostgreSQL 12.6 database to demonstrate how our method supplements backup process with purge rules and effectively purges data from backups. The database VM server consists of 8 GB of RAM, 4 vCPUs, 1 × vNIC and a 25 GB VMDK file. The VMDK file was partitioned into: 350 MB/boot, 2 GB swap, and the remaining storage was used for the/partition; this was done with standard partitioning and ext4 filesystem running CentOS 7 on VMware Workstation 16 Pro. We demonstrate the viability of our approach by showing that it can be implemented without changing the original schema or standard backup procedures, while guaranteeing data purging compliance.

We use two tables, `Alpha` and `Beta`, with `Beta` containing children rows of `Alpha`. As shown in Fig. 3, shadow tables contain the encrypted value for each attribute and the encryption key used. Shadow tables use the datatype bytea

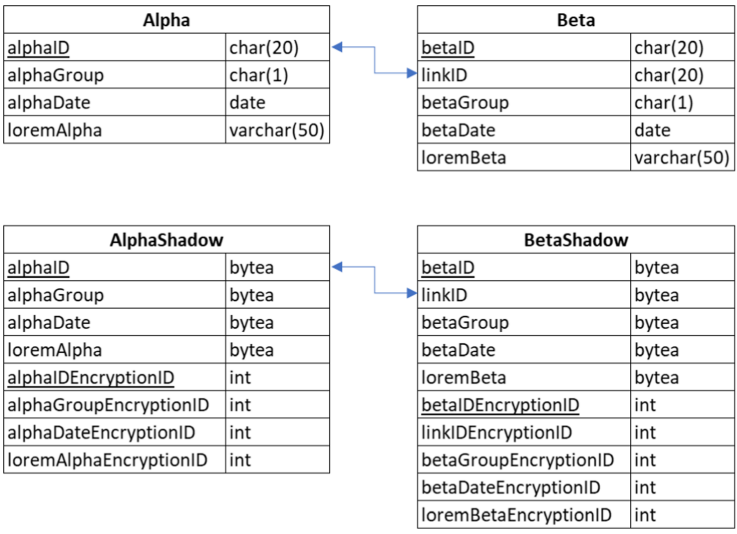

Fig. 3. Tables used in our experiments

(binary array) to store the encrypted value regardless of the underlying data type as well as an integer field that contains the encryption key ID used to encrypt the field. We tested the most common datatypes such as char, varchar and date.

This experiment used two different purge policies. The first policy requires purging data from both tables where the `alphaDate` is older than five years old and `alphaGroup`='a' (randomly generated value occurring in approximately 25% of the rows). The second policy requires purging only from the `Beta` table where `betaDate` (generated independently from `alphaDate`) is older than five years old and `betaGroup`='a' (separately generated with the same probabilities).

Our trigger on each table fires upon `INSERT`, `UPDATE`, or `DELETE` to propagate the change into the shadow table(s). When the insertion trigger fires, it first checks for an encryption key in the `encryptionOverview` table for the given policy and expiration date; if one does not exist, the key is created and stored automatically.

We pre-populated `Alpha` table with 1,000 rows and `Beta` table with 1,490 rows. We also generated a random workload of inserts (25), deletes (25), and updates (25) for the time period between 1/1/2014 to 2/1/2019. Because we used two different policies, we generated the data so that some of the business records were subject to one of the purge policies and some records were subject to both purge policies. Roughly 75% of the data generated was subject to a purge policy. Finally, not all records requiring encryption will be purged during this experiment due to the purge policy date not having passed. Records generated with dates from 2017–2019 would have not expired in running of this experiment. We then perform updates and deletes on the tables to verify that our implementation is accurately enforcing compliance.

Using a randomly generated string of alphanumeric characters with a length of 50, our process uses the function PGP_SYM_ENCRYPT to generate encryption keys to encrypt the input values. `alphaID` is the primary key of `Alpha` and (`alphaID`, `alphaIDEncryptionID`) is the primary key of `AlphaShadow`. If `alphaID` is not encrypted, the column `alphaIDEncryptionID` is set to –1 to maintain uniqueness and primary key constraint.

The `UPDATE` trigger for the `Alpha` table is similar to the `INSERT` trigger, but it first deletes the existing row in the shadow table. Next, we determine the current applicable encryption key and insert an encrypted updated row into the shadow table. The `DELETE` trigger removes the row from the `AlphaShadow` table upon deletion of the row in `Alpha`. When `alphaDate` in a row from `Alpha` changes, the corresponding rows in `Beta` table may fall under a different policy and must be re-encrypted accordingly. Furthermore, when a `Alpha` row is deleted, the child `Beta` row must be deleted as well along with the shadow table entries. Note that PGP_SYM_ENCRYPT may generate several different ciphertext values given the same value and the same encryption key. Therefore, we cannot encrypt the value from `Alpha` and compare the encrypted values. Instead, we must scan the table and match the decrypted value in the predicate (assuming the key is encrypted):

```
DELETE FROM alphaShadow
WHERE PGP_SYM_DECRYPT(alphaID, v_encryption_key)=old.alphaID
   AND alphaIDKey=v_key_id;
```

Changes to `Beta` table are a little more interesting since there is a foreign key relationship between `Beta` rows and `Alpha` rows. When a row is inserted or updated in the `Beta` table, in addition to the `Alpha` trigger processes, the `Beta` table triggers must compare the expiration date of the `Beta` row to the expiration date of the `Alpha` parent row and select the encryption bucket with the shorter of the two periods.

Initialization: We first import data into the `Alpha` and `Beta` tables. We then ran loadAlphaShadow() and loadBetaShadow() to populate the shadow tables using the corresponding key; the dates in the `encryptionOverview` table are initialized based on our expiration dates. Next, we enabled the triggers and incremented dates in `encryptionOverview` by five years to simulate the policy's expiration at a later time.

Validation: We wrote a procedure, RestoreTables(), to restore `Alpha` and `Beta` tables after shadow tables were restored from backup. In a production database, the backup method would depend on the Recovery Time Objective (RTO) and Recovery Point Objective (RPO) which would determine the backup methodology implemented, such as with PostgreSQL's pg_dump and excluding the tables with sensitive data. We tested the basic backup and restore process by exporting and importing the shadow tables, then truncating `Alpha` and `Beta`, and finally invoking our RestoreTables() procedure. We then modified the procedure to restore the tables to (temporarily created) `Alpha`' and `Beta`' so that we could compare restored tables to `Alpha` and `Beta`. We then verified that the values for the restored tables match the original tables' non-purged records.

Evaluation: We have verified that by deleting encryption keys to simulate the expiration of data, the restore process correctly handled the absence of a key to eliminate purged data. In total, there were 61 rows purged from `Alpha` and 182 rows purged from `Beta`, as well as the same rows purged from `AlphaShadow` and `BetaShadow`. Therefore, we have demonstrated that our framework achieves purging compliance in a relational database without altering tables in the existing schema or modifying the standard backup procedures.

Encrypting and maintaining a shadow copy of sensitive data to support purging incurs processing overheads for every operation that changes database content (read operations are not affected). Optimizing the performance of this approach is going to be considered in our future work. During an `INSERT` on the `Alpha` table, our system opens a cursor to check if an encryption key is available in the `encryptionOverview` table. If the applicable key exists we fetch it, otherwise we create a new one. Once a key is retrieved or a new key is generated, the values that are under a purge policy are encrypted with PGP_SYM_ENCRYPT. Next, we insert encrypted data into the shadow table as part of the transaction. For an `UPDATE`, we follow the same steps but also delete the prior version of the row from the shadow table (and may have to take additional steps if the update to the row changes the applicable purge policy). If the policy condition changes, we insert the shadow row into `AlphaShadow` and then evaluate the data in the `BetaShadow` table to see if the encryption key needs to change on the encrypted rows of the `BetaShadow` where the `linkID` refers to the `Alpha` row that changed.

The restore process is subject to decryption overheads. For example, in Postgres, in addition to the normal restore operation that restores the shadow table, we recreate the unencrypted (active) version of the table. For each encrypted column, we look up the key, then apply PGP_SYM_DECRYPT, and finally insert the row into the active table (unless the row already expired). Because the restore process creates an additional insert for every decrypted row, this also increases the space used for the transaction logs. The performance overhead for a restore will be correlated with doubling the size of each encrypted table (due to the shadow copy addition) plus the decryption costs. During deletion, each time we decrypt a row, the process of executing PGP_SYM_ENCRYPT and evaluating each row of the table incurs a CPU cost in addition to the I/O cost of deleting an additional row for each deleted row. The performance for an update statement incurs a higher overhead since an update is effectively a delete plus insert. Some of these I/O costs, such as fetching the key, can be mitigated with caching.

5 Discussion

5.1 Implementation

In our experiments we exported and imported the shadow tables to show that the system worked as expected; in practice, backup methodology would depend on the RTO and RPO on the application [10]. There are a plethora of options that can be implemented depending on the needs of the application. One could use `pg_dump` and exclude the tables containing sensitive data, so that these tables

are excluded from the backup file. If the size of the database is too large for a periodic `pg_dump`, or if the RTO and RPO warrant a faster backup, one could replicate the database to another database, and exclude the tables with sensitive data from replication. Using the clone of the database, one could do filesystem level backups or a traditional `pg_dump`. While the clone is a copy, a clone is not versioned in time like backups would be. For example, if someone dropped a table, the drop would replicate to the clone and not protect data against this change, whereas a backup would allow restoring a dropped table.

5.2 ACID Guarantees

If a trigger abends at any point, the transaction is rolled back. Since we attach triggers to the base tables, we are able to provide ACID guarantees. These guarantees are also extended to the shadow tables because all retention triggers execute within the same transaction. Overall, for any table dependencies (either between the active tables or with the shadow tables), our framework executes all steps in a single transaction, fully guaranteeing ACID compliance. This guarantee requires additional steps if we replicate the changes outside of the database since the database is no longer in control of the transaction.

For example, if the remote database disconnects due to a failure (network or server), the implementation would have to choose the correct business logic for the primary database. If the primary database goes into a read-only mode, the primary can keep accepting transactions or keep a journal to replay on the remote database. If the implementation kept a journal to replay, organizations must determine if is it acceptable to break ACID guarantees. Oracle DataGuard and IBM Db2 HADR provide varying levels of replication guaranties; similar guaranties would need to be built into our framework and verbosely explained as to the implications. Similarly, supporting asynchronous propagation and encryption of data into shadow tables would require additional investigation.

5.3 Future Work

We plan to consider asynchronous propagation (instead of triggers) to shadow tables; although that would require additional synchronization mechanisms, it has the potential to reduce overheads for user queries. Because scalability is a concern, tools such as Oracle Goldengate or IBM Change Data Capture, provide a framework to replicate changes, apply business logic, and replicate the changes to the same database or other heterogeneous databases. We also intend to explore developing our framework to replicate changes outside of a single database.

Our approach can easily incorporate new policies without requiring any changes to the already defined policies. However, when a policy is removed, all data in the shadow tables will stay bucketed under the previous policy. Further research is needed to automatically re-map all data points to the newest policy after a policy has been replaced or altered, to facilitate up-to-date compliance.

6 Conclusion

Organizations are increasingly subject to new requirements for data retention and purging. Destroying an entire backup violates retention policies and prevents the backup from being used to restore data. Encrypting the active database directly (instead of creating shadow encrypted tables) would interfere with (commonly used) incremental backups and introduce additional query overheads. In this paper we have shown how a framework using cryptographic erasure is able to facilitate compliance with data purging requirements in relational database backups.

Our approach does not change the active tables and maintains support for incremental backups while providing an intuitive method for data curators to define purge policies. This framework balances multiple overlapping policies and maintains database integrity constraints (checking policy definitions for entity and referential integrity). We demonstrate that cryptographic erasure supports the ability to destroy individual values at the desired granularity across all existing backups.

Overall, our framework provides a clear foundation for how organizations can implement purging into their backup processes without disrupting the organization's business continuity processes. This is also accomplished without adding any restrictions to existing databases. Our purging framework is able to guarantee purging compliance while being easily integrated into existing databases.

References

1. 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)
2. Boneh, D., Lipton, R.J.: A revocable backup system. In: USENIX Security Symposium, pp. 91–96 (1996)
3. Citus Data: pg_cron, https://github.com/citusdata/pg_cron
4. Congress, U.S.: 28 u.s. code §1732 (1948)
5. Dudjak, M., Lukić, I., Köhler, M.: Survey of database backup management. In: 27th International Scientific and Professional Conference"Organization and Maintenance Technology (2017)
6. European Parliament: Regulation (EU) 2016/679 of the European parliament and of the council (2020). https://gdpr.eu/tag/gdpr/
7. International Data Sanitization Consortium: Data sanitization terminology and definitions, September 2017. https://www.datasanitization.org/data-sanitization-terminology/
8. IRS: How long should i keep records? https://www.irs.gov/businesses/small-businesses-self-employed/how-long-should-i-keep-records
9. Kamara, S., Lauter, K.: Cryptographic cloud storage. In: Sion, R., Curtmola, R., Dietrich, S., Kiayias, A., Miret, J.M., Sako, K., Sebé, F. (eds.) FC 2010. LNCS, vol. 6054, pp. 136–149. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14992-4_13
10. 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

11. Office of the Attorney General: California consumer privacy act (CCPA), July 2020. https://oag.ca.gov/privacy/ccpa
12. Reardon, J., Basin, D., Capkun, S.: SoK Secure data deletion. In: 2013 IEEE Symposium on Security and Privacy, pp. 301–315. IEEE (2013)
13. Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.): DEXA 2010. LNCS, vol. 6261. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15364-8