Journal of Data Intelligence, Vol. 3, No. 1 (2021) 149–168

PURGING COMPLIANCE FROM DATABASE BACKUPS BY ENCRYPTION

NICK SCOPE
DePaul University, 243 S Wabash Ave
Chicago, Illinois 60604, USA
nscope52884@gmail.com

ALEXANDER RASIN
DePaul University, 243 S Wabash Ave
Chicago, Illinois 60604, USA
arasin@cdm.depaul.edu

BEN LENARD
DePaul University, 243 S Wabash Ave
Chicago, Illinois 60604, USA
blenard@anl.gov

JAMES WAGNER
University of New Orleans, 2000 Lakeshore Dr
New Orleans, Louisiana 70148, USA
jwagner4@uno.edu

KAREN HEART
DePaul University, 243 S Wabash Ave
Chicago, Illinois 60604, USA
kheart@cdm.depaul.edu

Data compliance 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 must 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 across database backups. We demonstrate how different purge policies can be defined through views and enforced 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) [10] and California Consumer Privacy Act (CCPA) [23]. With the increased emphasis on supporting data gov-

ernance rules, many organizations are seeking 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 [12]), 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). Organizations may furthermore establish their own retention and purging policies or have to implement custom data governance control (e.g., purge data due to the owner's request).

In this paper, we consider the problem of data purging in a database. We analyze this problem and propose solutions for both relational and JSON NoSQL database (JSON ND) logical layouts. Prior research has only focused on the challenges of retention in active (live) databases [3]. Nevertheless, to fully comply with the policies mandating data purging, a database 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. The problem of purging policies and backups has been studied in the context of relational databases [28]. In this paper, we extended this approach's framework to support backup purging for both relational and JSON databases.

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 Section 2.3) 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 [4]. Intuitively, a cryptographic erasure 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. This process does not invalidate partial backups.

Our framework creates *shadow collections* which contain an encrypted copy of all data subject to purging policies. In a context of a relational database, a collection is simply a table. With JSON ND databases, data is stored in this manner would be a collection. Throughout this paper, we use the term *collection* referring to both relational database tables and collections in JSON ND. In our approach, the shadow collections are backed up instead of the original collections; 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. In order for any solution to be implemented in industry, it must not come with a high cost for adaption. Furthermore, it must be compatible with current processes. If an organization is unable to use major features (e.g., incremental backups), they are unlikely to implement a purging framework. One challenge of implementing cryptographic erasure is in balancing different policy requirements across

a database logical layout. A single row in a relational table may have columns subject to different retention and purging requirements; in JSON ND collections, different key-values may also have different 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 address the problem of 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. In sum, 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 relational and JSON databases during backup and restore
- We design a key selection mechanism that balances multiple policies and retention period requirements
- We provide experiments illustrating the purging process in a relational and JSON database

2. Background

2.1. *Database Terminology*

Relational Databases: A relational database (RDBMS) represents data in two dimensional tables. Each table represents a set of records (or tuples) where each record is uniquely identified by a primary key. Data stored stored across multiple tables (e.g., customer and purchase information) is connected by foreign key references. RDBMSes enforce referential integrity, which ensures that all foreign key references are valid (referencing a valid record in the remote table). Relational database functionality is mostly standardized, consistent across vendors such as Oracle, PostgreSQL, and MySQL (although each DBMS comes with their own proprietary features and supported add-ons).

NoSQL: NoSQL databases have been designed to be more flexible and scalable by relaxing some of the RDMBS constraints (such as referential integrity). There are multiple loosely defined NoSQL database categories; in this paper, we focus on the broad category of document stores that store JSON style documents identified by a unique key. A document roughly corresponds to a tuple in an RDBMS, although it is more loosely defined (e.g., each document can have a varying number of values or columns). Thus, a NoSQL document corresponds to an RDBMS table; in MongoDB, documents are named collections. In relational databases, foreign keys maintain a relationship between tables, data in each column must belong to the declared data type, and many additional custom constraints can be introduced. NoSQL constraints are less standardized and not required but can still be introduced. For example, in MongoDB, schema validations can enforce custom restrictions on values and data types

within a document. In Elasticsearch, document validation can be implemented with either mappings and REST filters or on the client application itself.

ACID: Transactions help manage concurrent access to the DBMS and are used for crash recovery. RDBMSes guarantee that transactions are atomic, consistent, isolated, and durable (ACID). For example, if a customer transfers $10 from account A($50) to account B($5), transactions ensure that the *transient* account state (account A is already at $40 but account B is still at $5) cannot be observed. Should the transfer fail mid-flight (after subtracting $10 from A, but before adding $10 to B), transactional mechanism restores account A back to $50. Changes performed by transactions are stored in the transaction log (e.g., <A, $50, $40>, <B, $5, $15>), which can undo or reapply the changes, depending on whether the transaction successfully executed the COMMIT (i.e., was "finalized").

NoSQL databases offer a more limited ad-hoc support of ACID guarantees. Although outside of RDBMS world ACID support is optional, it is a desired feature. ACID was introduced to NoSQL databases recently: for example, MongoDB for multi-document transactions [17] was added in version 4.0 (circa 2018). In Elasticsearch, transactions and ACID guarantees are not yet implemented within the database engine. If this functionality is required, a two-phased commit approach would need to be implemented [16].

Triggers: Triggers are a mechanism designed to enforce database-wide rules by firing (executing) in response to a database event. For example, a trigger can act before an UPDATE operation and either log the operation before it happens or prevent the update from happening. RDBMS triggers use an extension of SQL (e.g., PL/SQL, SQL/PL, or T-SQL) and are ACID-compliant. In NoSQL databases, trigger support has been implemented more recently. For example, MongoDB Atlas implemented triggers for their cloud solution [19] and change streams for on-premise deployments as of version 3.6 [18] (circa 2017). Other NoSQL databases have implemented functionality similar to triggers but with different system-specific limitations. For example, in Elasticsearch, a Watcher is a process that fires at a fixed interval [9] instead of responding to specific events.

2.2. *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" [6]. A business record may consist of a single document for an organization (e.g., an email message). In a relational 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).

Regardless of database schema, a business record is constructed from a database using a query. With relational databases, database business record queries are defined using Select-Project-Join operations. With respect to JSON databases, this is accomplished using find queries. Aggregations are the result of combining business records; they are not used to define business records. Therefore throughout this paper, we define a business record as the result of a non-aggregate query.

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.

Policies can use a combination of time and external events as the criteria for data purging. Policies must be mapped to the underlying business records to know which records must either be retained or purged. For example, retaining employee data until employee termination plus 5 years illustrates a policy criteria that is based on a combination of an external event (employee termination) and time (5 years). United States Department of Defense (DoD) "DoD 5015-02-STD" documentation [2] outlines the minimum requirements and guidance for any record system related to the DoD (although multiple US government agencies, such as the National Archives, use the same standards). These DoD guidelines state that any storage system must support retention thresholds such as time or event (Section C2.2.2.7 of [2]).

Purging: In data retention, purging is the permanent and irreversible destruction of data in a business record [11]. 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.

Some retention policies require an organization to completely purge business records either after the passage of time, at the expiration of a contract, or purely when the data is no longer needed. Additionally, there are an increasing number of regulations (such as the European Union's GDPR) which require organizations to purge business records at the request a customer. Therefore, organizations must be prepared to comply with regular policies as well as ad-hoc requests.

2.3. *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 [7] 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 or documents).

Even for a relatively simple task of coping database files at the file system level, there are various methods ranging from a simple copy command with an offline database to a more complex copy of the storage system in an active database [25]. Moreover, one can replicate database changes from one database to another to provide a live backup or replica by using tools such as Oracle Data Guard [24], in addition to taking traditional backups of the database.

The type of backup depends on two application metrics: Recovery Point Objective (RPO) and Recovery Time Objective (RTO); as your RPO and RTO shorten, the complex solutions such as Oracle Data Guard [24] emerge. RTO is defined as the time it takes to recover the database after a crash and RPO is defined as how close to the crash or error can you restore

the database. A backup solution could be a simple RDBMS backup utility to snapshot filesystem contents, replication of the database and then backing up the database as well as other options [24]. The criticality of the application and data as well as RTO and RPO determines the backup solution and its complexity; in other words, how much would downtime or the financial and reputation loss of data could cost your organization. In addition to data, one might also backup transaction logs, archievelogs (Oracle), logarchive (Db2), binary log (MySQL), or write ahead log (WAL in Postgres), file backups, to replay transactions to meet the RPO goal of the application. For example, one would restore last backup before the failure and then replay the transaction logs to meet the RPO set by the organization's needs.

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 [14].

For JSON databases (e.g., MongoDB), a backup can be categorized into on-premise or cloud methodologies. For cloud deployments, one has the choice of taking snapshots of the database, akin taking a full backup of a relational database, or replicating changes to another target, which is similar Oracle's Data Guard in that changes are shipped from a primary to a standby. In terms of on-premise (or legacy deployments), you can backup the database by copying the underlying files, snap shotting the filesystem, or dumping the database (depending on your needs) [21]. For MongoDB, the backup stratigies are similar to relational databases and are driven by the business requirements. Overall, our framework is designed to work across backup categories.

2.4. *Related Work*

Kamara and Lauter's research has shown that using cryptography can increase storage protections [13]. Furthermore, their research has shown that erasing an encryption key can fulfill purging requirements. Our system expands on their research by using policy definitions to assign different encryption keys relative to their policy and expiration date.

Reardon et al. [26] provided a comprehensive overview of secure deletion. 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. [4] used cryptographic erasure, but each physical device had unique encryption key. Therefore, they did not develop a key-assignment system at the granularity required by purging policies. 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. [26] 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. [3] described some of the challenges associated with record retention implementation in relational databases. 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 retaining data in an active database; they did not consider how their approach would interact with backups.

Currently, there has been limited academic research on implementing purging policy requirements functionality in any NoSQL database. In the private sector, Amazon S3 does offer an object life-cycle management tool [1]. S3's object life-cycle management is limited to time criteria only. Moreover, S3 is file-based and therefore lacks sufficient granularity. All purging would be done at the file level without the ability to purge specific underlying key-value pairs.

3. Our Process

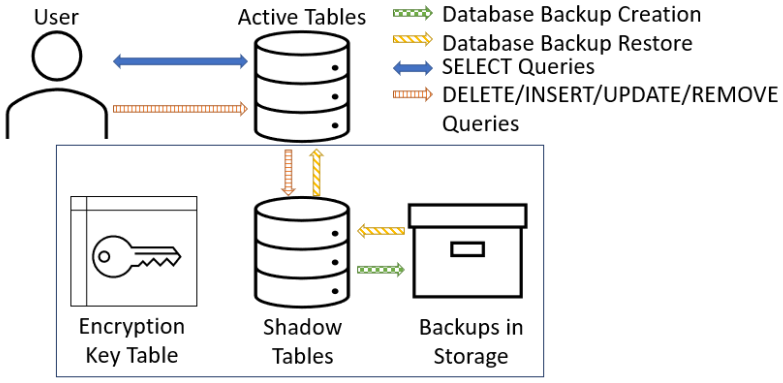

Fig. 1. Framework Overview

Our proposed framework automatically applies encryption to data that is subject to purge policy requirements whenever values subject a purging policy are inserted or updated. An overview of this process is presented in Figure 1. We maintain and backup a shadow (encrypted) copy of the collections; other collections not subject to purging rules are not affected. SELECT and FIND queries always interact with the non-encrypted database collections (rather than shadow collections) and are not impacted by our approach. We translate (using triggers) DELETE, INSERT, REMOVE, and UPDATE queries into a corresponding operation on the encrypted shadow copy of the collection.

Our framework relies on stored procedures (e.g., triggers) in order to operate transparently from the user's prospective. Using ACID compliant triggers, we make the necessary substitutions, applying encryption and updating the shadow collections in the background. If the trigger mechanism is not available or it does not support ACID guarantees, the functionality has to be moved to the application level. Ensuring data purging in the application breaks transparency, requiring some changes to the operations that modify the database (read-only queries are not affected). All major relational databases support trigger functionality. On the other hand, triggers are not always available with JSON databases. For example, in

MongoDB, triggers exist but only with MongoDB Atlas (the cloud version). Moreover, the implementation of triggers in Atlas is different from relational databases as they respond only after an event has happened (in a relational database, the trigger can act before or after an event). Adding to the complexity, triggers may be delayed further since the trigger application may be operating in a different cloud region [20, 22]. In terms of on-premise MongoDB, the implementation by developers also involves writing a "side-car" application to watch for events in Mongo after they have occurred. Another popular NoSQL database is ElasticSearch, and while triggers have been on their roadmap for a few years now, the only functionality that is akin to triggers is the watcher functionality. [9, 8] The watcher functionality fires on a set interval and can look for changes within the data.

Our framework is designed to remain transparent to the user. For example, one can use client-side encryption without affecting our data purging approach. A change in purge policy has to be manually triggered to encrypt existing data.

In our system, shadow collections are backed up instead of the corresponding user-facing collections; collections that are not subject to purging policies are backed up normally. When the shadow collections are restored from a backup, our system decrypts all data except for purged values. In relational databases, 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" NULLs in the database). In cases where the entire row must be purged (with all values having been eliminated), the tuple would not be restored. In JSON databases, once all values in a document are purged, the key is also removed.

Evaluation of possible conflicts between schema and purging policy requirements (e.g., purge policy on a column that is restricted to NOT NULL) are resolved during the policy definition step. When a policy is defined, the framework validates that the purging of data will not be in-conflict with the underlying schema requirements. MongoDB collections leveraging schema validation can require specific underlying keys during key-validations. For example, you can require a field to be present and of a given data type (e.g., a date field must be a date and within a given range or a field must be one of n options).

Our default implementation uses a collection called `encryptionOverview` (with column definition shown in Table 1) to manage encryption keys. This collection is backed up separately to avoid the problem of having the encryption keys stored with the backup. The key backups can be more readily purged of expired contents because they represent a small fraction of the overall database. Figure 2 illustrates an example of a JSON `encryptionOverview`.

Field	**PostgreSQL**	**MongoDB**
encryptionID	Int	String
policy	Varchar(50)	String
expirationDate	Date	String
encryptionKey	Varchar(50)	String

Table 1. encryptionOverview Collection

In our proof-of-concept experiments, the `encryptionOverview` collection is stored in the database. However, in a production system the key management collections will be stored in a separate database. Access to these collections could be established via a database link or in a federated fashion, allowing the keys to be kept completely separate from the actual data.

```
_id: ObjectId("61c242ec964ebb38273595e3")
encryptionID: "1"
policy: "1"
encryptionDate: "2014-01-01"
encryptionKey: "RGDVmmaAAzFQsiZIUCNwbEvjxRjkIGlybaFqXfBIQTrHeREpHW"
```


Fig. 2. MongoDB JSON ND encryptionOverview Example

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, a daily purging satisfies policy requirements to where further time granularity is not required (e.g., GDPR: Article 25 [10]).

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


```
_id: ObjectId("61be275e3e13c6b191c662ba")
alphaIDKeyID: 1
alphaIDEnc: Binary('c2MAAmU224ttMMgxBfSBzs/rIEmX+QA5tIZST3zqzuHLLUP8/xddFX3pJXK9yxdhQDCzeYu03rJJUQqlxfe4Z1mdtteDDNlqWsQ1...', 0)
alphaGroupKeyID: 1
alphaGroupEnc: Binary('c2MAAjTs+VeG6zPa426qWspAW81ZR7iqTBByRQtPp1Vt03iTbkrdlrkSrqlVngyPZtUeWsIgSeIoiSCmbFId4gbGAz0l', 0)
alphaDateKeyID: 1
alphaDateEnc: Binary('c2MAAuoGap1xh5jx/+aMKD0xqCske3hrrK5yIO5JDmGItbQriP3D6zSB+MIiaOOCSKUgBUYmz5BJ8zB4fCrFkXr8J01FsOkjm28w...', 0)
alphaLoremKeyID: 1
alphaLoremEnc: Binary('c2MAAq4DkXeH3Cw6cbwyTNnfV30Jyw2W/yc3aprF/yn+HEJL/Cirjy4OEFwvmgtO2xonveWxCqlGgsxC3lvX9fjqaTRbU4fE1uv6...', 0)
alphaDocID: ObjectId("61be275e3e13c6b191c662b9")
```


Fig. 3. MongoDB JSON ND `AlphaShadow` Example

With JSON ND, because each item may have different underlying key-values, JSON ND does not require a `[column name]EncryptionID` column for non-encrypted values. Our framework only adds a key-value pair for the `[column name]EncryptionID` if encryption has been applied. Therefore, with the values shown in Figure 3, any non-encrypted values would not have a corresponding `[column name]EncryptionID`. In this particular example, all values were subject to a purge policy and were encrypted.

3.1. *Defining Policies*

Our method of defining purge policies uses queries 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 time value).

With relational databases, 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 collection to maintain referential integrity. Because foreign keys can point to a unique column that is not a primary key. One of our framework's requirement for defining business records is that joins must uniquely identify each corresponding row between the collections.

As an example, using the schema in Figure 4, if the `customerID` in the `customer` collection

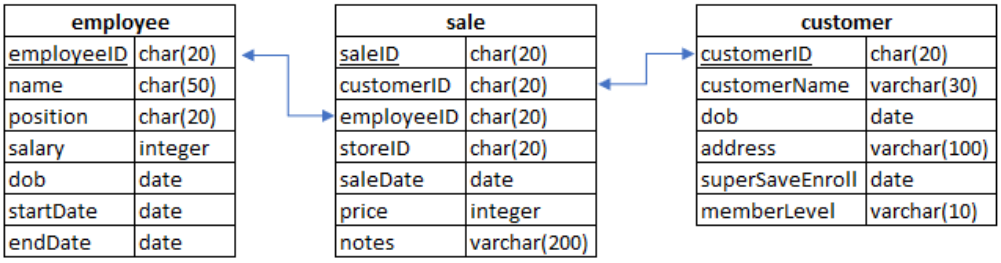

Fig. 4. Sample company schema

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 NULLs, 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 (Figure 4) that requires purging all customer data in their relational database where the Super Save enrollment date is over twenty years old. Using the keyword PURGE, our framework would recognize the VIEW below as a purging policy definition:

```
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. For a JSON ND, this would be accomplished using the purge creation definition below using the new function createPurge.

```
db.createPurge(
    "customerPurge",
    "customer",
    [{"superSaveEnroll":
        {$lte: new Date(ISODate().getTime() - 1000 * 3600 * 24 * 365 * 20 }}])
```

Our framework recognizes createPurge as VIEW designed specifically for our purging process. With JSON databases, if a key is included in a purge, all corresponding values in the collection must also be included. Because of the flexibility of the JSON ND schemas, instead of restoring purged values with NULL, those key-value pairs are simply ignored during the restoration process. Unlike with relational databases, in JSON databases each document can have different underlying key-values; our framework does not have to restore purged values to satisfy schema requirements. Additionally, referential integrity checks are not required during the business record definitions with JSON databases. Purging values from collections are handled independently unless they are defined under a single policy and business record.

3.2. *Encryption Process*

When a new record is inserted, we use triggers to determine if any of the values fall under a purge policy; if so, the trigger determines the relevant policies and their corresponding purge date. For example, consider a new employee record inserted into the employee collection:

```
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` collection. 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` collection. The encrypted row and the matching encryption key ID is inserted into the `customerShadow` collection. With relational databases, 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 collection 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.

This process would be similar using insertion of Isabel's data into a JSON database.

```
db.customer.insert([
{ customerID:1, customerName:"Johnson,Isabel", dob:"2/1/1990", address:"Chicago",
    superSaveEnroll:"1/1/2021", memberLevel:"Premium" }])
```

With a JSON database, because key values are more flexible, we do not need to worry about inserting a value of -1. Encryption keys are only stored for values that are encrypted, otherwise, they key-value is not added to the database. For example, using Figure 3, if any of the values did not require encryption, the original value would be shown in the shadow collection with no encryption key data stored.

To support multiple purge policies, we must determine which policies apply to the new data. A record in a collection 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 value 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. This is done to guarantee that a separate policy will not be violated by failing to drop a value when the another purge policy is removed. For example, let's say two different policies requiring purg-

ing. The first policy requires purging at t_1 while the second policy requiring purging at t_2 (where t_2 is after t_1). Because the policy at t_1 comes first, all overlapping values would be assigned t_1's encryption key. Once a backup has been created, if the policy with t_1 no longer requires purging, if we simply were to never purge those values, all values requiring purging at t_1 and t_2 would be in violation of the policy requirements after t_2. When our framework applies encryption, we only use the Our framework only tracks the earliest purging policy time when encrypting the values. Furthermore we are unable to apply a different encryption key post-backup. Therefore, to fully guarantee compliance, we must purge all data in backups regardless of if a policy is still active.

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` collection, 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 collection. 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 collection. Our prototype system decrypts the document identfier columns in the shadow collection to identify the updated row. This is a PostgreSQL-specific implementation requirement, which may not be needed in other databases (see Section). Our system automatically deletes the original row from the shadow collection and inserts the new record (with encryption applied as necessary), emulating UPDATE by DELETE+INSERT.

In our MongoDB example, we implemented this at the application level. Specifically, we developed a Python script which performed similar to a BEFORE trigger to execute the necessary encryption functionality. When a value was subject to a policy, the Python determines which encryption key to use (potentially generating a new key), apply the encryption key, and insert the data into the original and shadow collections. Specifically with updates, the Python script determines which value was being updated to remove the old encryption key from the shadow collection and insert the new updated encrypted value.

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 collection using the `customerID` document identifer combined with the previously used column `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*

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 [3, 27].

The encryption keys can be deleted by a cron-like scheduler, available in most DBMSes. Purging is automated through a cron-like DBMS job ([5] in Postgres) that removes expired encryption keys from `encryptionOverview` with a simple delete. Moreover, key deletion will need to be supplemented by a secure deletion of the encryption keys on the underlying hardware [26, 4], guaranteeing the encryption keys are permanently irrecoverable (which is outside the scope of this paper). While Mongo Atlas has scheduled triggers, akin to CRON, the on-premise version does not, so the purging would need to be dependent on CRON or something akin to this.

3.5. *Restore Process*

When creating the backups, our framework only creates the backups of collections without shadow copies or the shadow collections of tables. The `encryptionOverview` collection is ignored during this step.

Our framework restores the backup with shadow collections that contain encrypted as well as unencrypted values. Recall that with the relational databases, the shadow collections 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 collection. For any encrypted value associated with a purged `encryptionID` our system restores the value as a NULL in the active collection. If the entire row has been purged, the tuples would not be restored into the active collection.

With JSON databases, we are able simply to decrypt all values with using their related `[column name]EncryptionID` value. If an encryption key has been purged, the encrypted value is ignored during the restoration process. If a document key does not have a `[column name]EncryptionID` available, it and all underlying values are ignored (regardless of if the underlying values still have an available encryption key).

4. Experiments

4.1. *Relational Database Implementation*

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 in a relational database. The database VM server consists of 8GB of RAM, 4 vCPUs, 1 x vNIC and a 25GB VMDK file. The VMDK file was partitioned into: 350MB/boot, 2GB 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 collections, `Alpha` and `Beta`, with `Beta` containing children rows of `Alpha`. As shown in Figure 5, shadow collections contain the encrypted value for each attribute and the encryption key used. Shadow collections use the datatype bytea (binary array) to store the

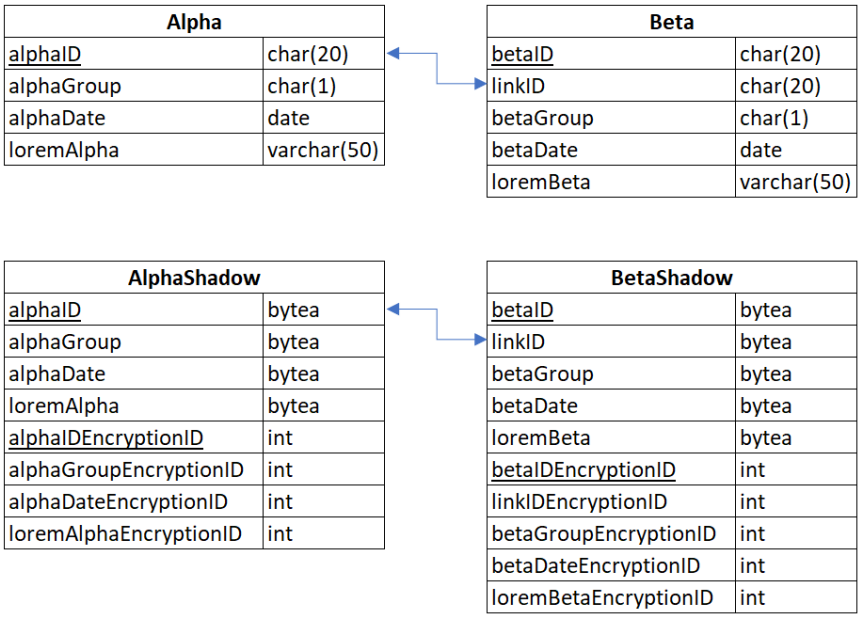

Fig. 5. Collections used in our experiments

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

We pre-populated `Alpha` collection with 1,000 rows and `Beta` collection 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 collections 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` collection is similar to the INSERT trigger, but it first deletes the existing row in the shadow collection. Next, we determine the current applicable encryption key and insert an encrypted updated row into the shadow collection. The DELETE trigger removes the row from the `AlphaShadow` collection upon deletion of the row in `Alpha`. When `alphaDate` in a row from `Alpha` changes, the corresponding rows in `Beta` collection 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 collection 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 collection 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` collection 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` collection, in addition to the `Alpha` trigger processes, the `Beta` collection 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` collections. We then ran loadAlphaShadow() and loadBetaShadow() to populate the shadow collections using the corresponding key; the dates in the `encryptionOverview` collection 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` collections after shadow collections 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 collections with sensitive data. We tested the basic backup and restore process by exporting and importing the shadow collections, then truncating `Alpha` and `Beta`, and finally invoking our RestoreTables() procedure. We then modified the procedure to restore the collections to (temporarily created) `Alpha`' and `Beta`' so that we could compare restored collections to `Alpha` and `Beta`. We then verified that the values for the restored collections match the original collections' 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

collections 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` collection, our system opens a cursor to check if an encryption key is available in the `encryptionOverview` collection. 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 collection 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 collection (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` collection 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 PostgreSQL, in addition to the normal restore operation that restores the shadow collection, we recreate the unencrypted (active) version of the collection. For each encrypted column, we look up the key, then apply PGP_SYM_DECRYPT, and finally insert the row into the active collection (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 collection (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 collection 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.

4.2. *JSON ND Implementation*

We conducted a similar experiment in MongoDB 4.4 on-premise, with the same VM setup as described for PostgreSQL, to demonstrate and confirm we are able to implement our process in a JSON database for purging data from backups. We used MongoDB since in the fall of 2021, the company was worth around $30 billion dollars, which is indicative of it commercial use and success [15]. From a logical perspective, the code for two experiments followed a similar workflow. Because BEFORE triggers are not currently supported in MongoDB, we implemented our functionality within the MongoDB client driver and our Python application to ensure that `Alpha`, `Beta`, and their shadow collections remain consistent (i.e., the insert into `Alpha` and its shadow happen in one transaction, similarly with `Beta`). MongoDB has implemented multi-document ACID in version 4.0.

Therefore, instead of using PL/SQL like with our PostgreSQL experiments, we utilized Python 3.6 and the PyMongo driver. Additionally, we changed the process to find, update, and delete the shadow documents. With JSON databases, we are able to use a universally unique identifier (UUID) to determine which row to decrypt, update, and re-encrypt. With

PostgreSQL, we had to decrypt and scan every row to find the row we need to delete or update; however with JSON, including Mongo, every document had a _id field containing a UUID. Although many relational databases do have a row identifier, this may change during a collection reconstruction. By having a consistent UUID, we are able to streamline the process by minimizing the amount of values we are required to decrypt.

We utilized a different symmetric encryption package to encrypt and decrypt the data within the shadow document. With our PostgreSQL experiment, we were able to leverage the included encryption functionality. On the other hand, MongoDB does not currently have encryption functionality built in. Therefore, we used the Python package Simple Crypt.

MongoDB backup are able to be partitioned at the collection level. Therefore, we were able to generate backups using only the shadow collections containing the encrypted data. This experiment used collections that mirrored the collections from our relational database experiment. Furthermore, the same data, synthetic time lapse, and purging of encryption keys was executed. Therefore, the values and records purged would match between the two experiments.

The results mirrored the results from our previous experiment with relational databases. Our system was able to successfully backup the data, destroy the encryption keys, purge the necessary records, and restore the database (while ignoring encrypted values where the encryption key has been purged). Therefore, this experiment proved our framework can successfully purge inaccessible records in an inaccessible JSON ND backup.

5. Discussion

5.1. *Implementation*

In our experiments we exported and imported the shadow collections to show that the system worked as expected; in practice, backup methodology would depend on the RTO and RPO on the application [14]. 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 collections containing sensitive data, so that these collections 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 collections with sensitive data from replication. Using the clone of the database, one could do filesystem level backups or a traditional `pg_dump`. Similarly in MongoDB there is a `mongodump` to backup the database in a similar fashion. While the clone is a copy, a clone is not versioned in time like backups would be. For example, if someone dropped a collection, the drop would replicate to the clone and not protect data against this change, whereas a backup would allow restoring a dropped collection. We intend to study the performance and granularity trade off (by changing bucket size) in future work.

We intend to separate the `encryptionOverview` collection from the database in production. Currently, we did not evaluate that functionality in our experiments for this paper. Because the `encryptionOverview` is relatively small, it must be separately backed up. Otherwise, if the `encryptionOverview` collection were to be lost, all backups would be rendered void. To guarantee compliance, the backups of `encryptionOverview` must regularly be purged to prevent multiple copies of encryption keys from remaining in perpetuity.

5.2. *JSON ND Functionality Limitations*

Our framework is not limited by forcing the solutions we implemented in relational databases into a JSON databases. Because JSON databases have different limitations compared to a relational databases, some changes were required. We used BEFORE triggers in the relational databases. Currently there is no native support for BEFORE triggers in JSON databases. Therefore, for the solutions to be completely consistent, this either needs to be added to the native support, or users must build this into the application level. Unfortunately, building this into the application lessens the transparency to the user.

5.3. *ACID Guarantees*

If a trigger abends at any point, the transaction is rolled back. With relational databases, since we attach triggers to the base collections, we are able to provide ACID guarantees. These guarantees are also extended to the shadow collections because all retention triggers execute within the same transaction. Overall, for any collection dependencies (either between the active collections or with the shadow collections), 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 collections would require additional investigation.

MongoDB does currently guarantee ACID (as of version 4.0) if implemented in the application and the client driver for Mongo. Specifically, in our MongoDB experiment, we leverage the ACID functionality in the client driver to implement our encryption framework. With MongoDB's locking functionality, we are able to maintain ACID at the cost of transparency.

ACID inherently is not at conflict with transparency. Until BEFORE triggers are supported though, our framework's implementation in JSON databases must sacrifice transparency for ACID compliance. In JSON databases where BEFORE triggers are supported, our framework would be implemented with triggers instead of at the application level (facilitating enhanced transaction transparency).

5.4. *Future Work*

We plan to consider asynchronous propagation (instead of triggers) to shadow collections; although that would require additional synchronization mechanisms, it has the potential to reduce overhead 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 collections will stay bucketed under the previous policy. Further research is needed to automatically remap 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 collections) 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 database backups with relational and JSON databases.

Our approach does not require changes to the active collections 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 in relational databases). 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. Databases which either have trigger or application level support can use our purging framework to enforce purging compliance.

References

1. Amazon. AWS S3. `https://aws.amazon.com/s3/`, 2020.
2. Assistant Secretary of Defense for Networks and Information Integration. Electronic records management software applications design criteria standard. `https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodm/501502std.pdf`, Apr 2007.
3. Ahmed A Ataullah, Ashraf Aboulnaga, and Frank Wm Tompa. Records retention in relational database systems. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 873–882, 2008.
4. Dan Boneh and Richard J Lipton. A revocable backup system. In *USENIX Security Symposium*, pages 91–96, 1996.
5. Citus Data. pg_cron. `https://github.com/citusdata/pg_cron`.
6. United States Congress. 28 u.s. code §1732, 1948.
7. Mario Dudjak, Ivica Lukić, and Mirko Köhler. Survey of database backup management. In *27 th International Scientific and Professional Conference"Organization and Maintenance Technology*, 2017.
8. Elastic. Changes api - issue 1242 - elastic/elasticsearch. `https://github.com/elastic/elasticsearch/issues/1242`.
9. elastic. Watcher triggers: Elasticsearch guide [7.16]. `https://www.elastic.co/`.

10. European Parliament. Regulation (eu) 2016/679 of the european parliament and of the council. https://gdpr.eu/tag/gdpr/, 2020.
11. International Data Sanitization Consortium. Data sanitization terminology and definitions. https://www.datasanitization.org/data-sanitization-terminology/, Sep 2017.
12. IRS. How long should i keep records? https://www.irs.gov/businesses/small-businesses-self-employed/how-long-should-i-keep-records.
13. Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In *International Conference on Financial Cryptography and Data Security*, pages 136–149. Springer, 2010.
14. Ben Lenard, Alexander Rasin, Nick Scope, and James Wagner. What is lurking in your backups? In *ICT Systems Security and Privacy Protection IFIP Advances in Information and Communication Technology*, page 401–415. Springer International Publishing, 2021.
15. Ari Levy. Mongodb surge wraps up a massive week for open-source software as a business. https://www.cnbc.com/2021/09/03/mongodb-tops-30-billion-market-cap-in-banner-week-for-open-source.html, Sep 2021.
16. Alexander Marquardt. https://alexmarquardt.com/2019/12/05/emulating-transactional-functionality-in-elasticsearch-with-two-phase-commits/, Jan 2020.
17. MongoDB. Acid transactions basics. https://www.mongodb.com/basics/acid-transactions.
18. MongoDB. Change streams. https://docs.mongodb.com/.
19. MongoDB. Database triggers. https://docs.mongodb.com/realm/triggers/database-triggers/.
20. MongoDB. Deployment models and regions. https://docs.mongodb.com/realm/manage-apps/deploy/deployment-models-and-regions/\#global-deployment.
21. MongoDB. Mongodb backup methods. https://docs.mongodb.com/manual/core/backups/.
22. MongoDB. Mongodb stitch trigger delay. https://stackoverflow.com/questions/57811761/mongodb-stitch-trigger-delay, Oct 1967.
23. Office of the Attorney General. California consumer privacy act (ccpa). https://oag.ca.gov/privacy/ccpa, Jul 2020.
24. Oracle. Oracle maa reference architectures. https://www.oracle.com/.
25. Oracle. https://www.delltechnologies.com, Jan 2018.
26. Joel Reardon, David Basin, and Srdjan Capkun. Sok: Secure data deletion. In *2013 IEEE symposium on security and privacy*, pages 301–315. IEEE, 2013.
27. Nick Scope, Alexander Rasin, James Wagner, Ben Lenard, and Karen Heart. Database framework for supporting retention policies. In *International Conference on Database and Expert Systems Applications*. Springer, 2021.
28. Nick Scope, Alexander Rasin, James Wagner, Ben Lenard, and Karen Heart. Purging data from backups by encryption. In *International Conference on Database and Expert Systems Applications*. Springer, 2021.