Forget About It: Batched Database Sanitization

James Wagner
University of New Orleans
New Orleans, LA, USA
jwagner4@uno.edu

Alexander Rasin
DePaul University
Chicago, IL, USA
arasin@cdm.depaul.edu

ABSTRACT

In file systems and database management systems (DBMSes), deleting data marks it as unallocated storage rather than explicitly erasing data. This data can be reconstructed from raw storage, making it vulnerable to data theft and exposing organizations to liability and compliance risks, violating data retention and destruction policies. The problem is further magnified in DBMSes because (unlike in file systems) DBMS backups are performed in pages and will include such deleted records. Data erasure (or sanitization) is a process that eliminates this vulnerability, providing users with "the right to be forgotten". However, most of the work in data sanitization is only relevant to erasing data at the file system level, and not in DBMSes. Limited existing work in database sanitization takes an erase-on-commit approach, which can introduce significant I/O bottlenecks.

In this paper, we describe a novel data sanitization method, `DBSanitizer`, that 1) is DBMS agnostic, 2) can batch value erasure, and 3) targets specific data to erase. `DBSanitizer` is designed as a template for DBMS vendors to support backup sanitization and ensure that no undesirable data is retained in backups. In this paper, we demonstrate how our approach can be used in any row-store relational DBMS (including Oracle, PostgreSQL, MySQL, and SQLite). As there are no backup sanitization tools available on the market or in research literature, we evaluate `DBSanitizer`, in a live database that supports erase-on-commit sanitization approach.

CCS CONCEPTS

• **Security and privacy → Data anonymization and sanitization**; **Information accountability and usage control**.

KEYWORDS

Data erasure, sanitization, secure deletion, data wiping, database forensics, anti-forensics

ACM Reference Format:
James Wagner and Alexander Rasin. 2024. Forget About It: Batched Database Sanitization. In *Proceedings of ACM SAC Conference (SAC'24).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3605098.3636054

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04…$15.00
https://doi.org/10.1145/3605098.3636054

1 INTRODUCTION

The ultimate goal of security systems is to prevent and detect unwanted data access. Accessible data includes not only the obvious stand-alone files (e.g., PDFs, JPEGs) or active records in a database, but also data that resides in unallocated storage. File systems and DBMSes do not explicitly overwrite data upon deletion. Instead, data becomes unallocated storage that is free-listed and eventually overwritten. The deleted data that resides in unallocated storage is retrievable with plain-text searches (e.g., `GREP`), or it can be reconstructed with digital forensics methods (e.g., file carving [26] or database page carving [39]).

DBMSes are used to manage data both in a corporate and in personal setting. A lightweight DBMS, such as SQLite, is commonly used to manage personal data stored on mobile phones or web browsers. Whereas, a DBMS that supports more robust access control and storage management, such as Oracle, PostgreSQL, MySQL, or Microsoft SQL Server, is better suited to manage corporate data. Post-deletion data can become a long term liability in a DBMS. Lenard et al. [14] demonstrated that a significant number of deleted records can be copied into (read-only) DBMS backups where it will be retained in perpetuity, until the backup is destroyed.

Data erasure is an anti-forensics method that overwrites data in a storage medium so that it is no longer reconstructable. Data erasure offers "the right to be forgotten" [44], and its importance is evident in the many laws and regulations that prohibit the disclosure of sensitive data including HIPAA [34], FERPA [35], Canada's PIPEDA [21], and GDPR [3]. Furthermore, many government agencies have procedures for the proper deletion and destruction of sensitive data [17, 18, 36]. Failing to comply with data governance rules and policies creates real liability risks. For example, in February 2022, EyeMed Vision Care was fined $600,000 for failing to comply with New York's Stop Hacks and Improve Electronic Data Security Act [31]. Among the violations, EyeMed failed to comply with the data retention requirements. Data erasure approaches are also referred to as data wiping [11, 13], sanitization [8, 17, 18], and secure deletion [16, 24].

Traditional sanitization techniques (e.g., [8, 15]) overwrite entire sectors on disk to eliminate deleted files. This approach cannot be used for DBMS records because a deleted record is marked "deleted" in a live DBMS file (which is not deleted from the OS perspective). Current DBMSes do not offer a secure delete feature (except SQLite) because a typical implementation of erase-on-commit is considered too costly. We demonstrate that our system, `DBSanitizer`, which uses a batched erasure approach, can be efficiently implemented by any DBMS vendor. This paper evaluates `DBSanitizer` against PostgreSQL and SQLite. We expect that a DBMS vendor (with a better understanding of their own storage) can implement a more

§	Summary
2	Database storage concepts and terminology.
3	Related work in database forensics and erasure.
4	An overview of deleted data behavior within DBMS storage.
5	Considerations that must be taken when directly modifying DBMS storage.
6	An overview of DBSanitizer. Deleted records and objects, stale auxiliary data, and unallocated pages are evaluated for erasure.
7	Considerations for data erasure in DBSanitizer approach.
8	Overview of all storage areas where additional data copies may be found.
9	Experiments that demonstrate the capabilities and advantages of the batched approach used by DBSanitizer.

Table 1: Summary of the remaining paper.

efficient native version of DBSanitizer. This approach can be applied to backups or live DBMS storage. Table 1 summarizes the contributions of this paper.

1.1 DBSanitizer Approach

DBMSes were traditionally designed to favor performance, with security as a secondary consideration. Secure deletion (i.e., erase the data immediately when it is deleted) increases I/O costs which will negatively affect performance. DBMSes also have to consider auxiliary structures (indexes, materialized views) that contain extra copies of data. Furthermore, cloud based DBMSes pay a high cost for increased I/O demands – a cost that applies both to magnetic and SSD drives. Because of the high overhead, DBMS vendors have been unwilling to support sanitization. In fact, SQLite is the only DBMS to support secure deletion, but it is disabled by default because it negatively impacts performance [32].

In a file system, users can use third party tools (surveyed in [8]) to perform data sanitization. However, due to the tight control of DBMSes over their internal storage, they must instead rely on the DBMS vendor to supply such a feature. This paper proposes a data erasure system, DBSanitizer, which mitigates most of the performance concerns. DBSanitizer validates a batched "search and destroy" approach rather than the immediate erase-on-commit approach used by the SQLite `secure deletion` and related work (see Section 3). DBSanitizer has the following properties:

(1) **Database Agnostic.** It can be adopted by any (open or closed-source) relational row-store DBMS.
(2) **Batched Execution.** Data is erased in batches at a user-controlled frequency. Information about deleted data does not need to be maintained.
(3) **Targeted Erasure.** Users can specify the data to erase (i.e., erase only sensitive data values).

Database Agnostic. DBSanitizer operates independently of the DBMS (i.e., source code modification or vendor support is not required). However, we believe users would be skeptical to trust a third-party data erasure tool that has the potential to corrupt DBMS storage. Therefore, this system is intended to provide a template for DBMS vendors to introduce their own erasure functionality. DBSanitizer is meant to demonstrate that vendors can realistically introduce this functionality in a DBMS.

Batched Execution. DBSanitizer's batched approach to Erasure is consistent with the majority of related work in data erasure at the file system level [8]. To operate independently of the DBMS, DBSanitizer uses page carving (Section 3) to interpret and modify storage (Section 5). Rather than maintaining information about deleted data, page carving is used to find and reconstruct this data.

Target Erasure. Deleted data is scattered throughout DBMS storage, but users can specify which data are sensitive (e.g., SSN versus publicly available office phone #) to erase. Focused erasure further limits the overhead associated with the secure delete solution. DBSanitizer illustrates targeted erasure of sensitive data in DBMS storage and cost savings that it achieves.

2 DATABASE STORAGE CONCEPTS

This paper incorporates proof-of-concept experiments that modify internal DBMS storage to erase data. Internal storage is, by design, hidden from users. Thus, it is helpful to have an understanding of some internal details specific to DBMS storage. This section provides a generalized description of page-level storage for all (relational) DBMSes and terminology used in the paper. The concepts in this section apply (but are not limited) to IBM Db2, Microsoft SQL Server, Oracle, PostgreSQL, MySQL, Apache Derby, MariaDB, SQLite, and Firebird.

2.1 Database Pages

A DBMS storage layer partitions all physical structures (e.g., tables, indexes, and system catalogs) into fixed-size pages (typically 4, 8, or 16 KB). Fixed-size pages across an entire instance also simplify storage and cache management.

When data is inserted or modified, the DBMS controls data placement within pages and internally maintains additional metadata. Across all DBMSes on the market, many commonalities exist for how data is stored and maintained at the page level. Every row-store DBMS uses pages with three main structures: header, row directory, and row data. Figure 1.A displays a high-level breakdown of a page with all three of these structures.

The page header stores metadata describing the user records stored in the page. The page header metadata of interest to this paper are the checksum, object identifier, page identifier, and row count. Figure 1.B demonstrates how this metadata could be positioned in a page header. The checksum detects page corruption; whenever a page is modified, the checksum is updated. The object identifier represents the object to which the page belongs. The plaintext object name (e.g., table name) is not stored in a page, but the object identifier can be mapped to the system catalog data to

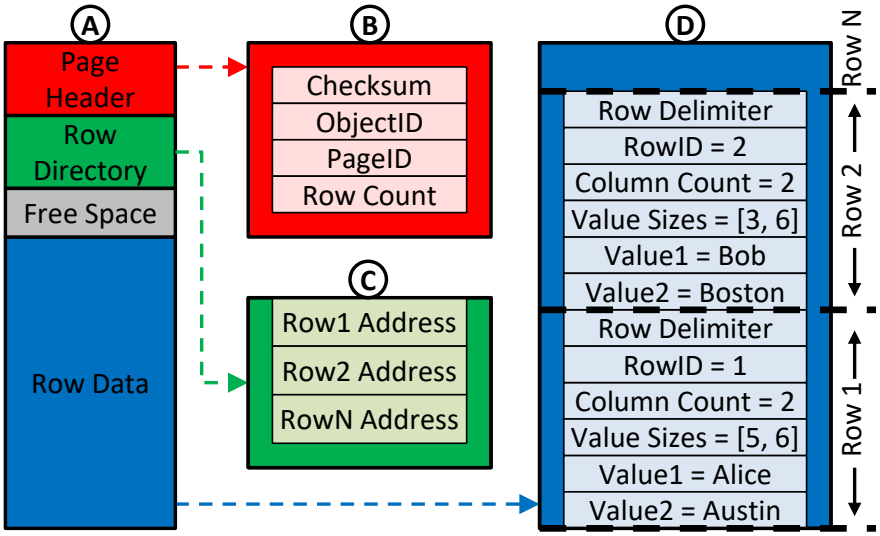

Figure 1: Page examples: A) high-level, B) header, C) row directory, and D) row data.

retrieve it. Depending on a DBMS, the page identifier is unique to each page for either each object, within a file, or across all DBMS files. The row count refers to the number of *active* records within a page. If a record is deleted in a page, the row count is decremented by one; if a record is added, it is incremented.

The row directory stores pointers to each record. When a record is added to a page, a corresponding pointer is added to the row directory. Figure 1.C shows an example row directory. The row data segment stores user data along with metadata, describing record layout. Figure 1.D shows an example row data structure (with minor DBMS-specific variations). In this example, each record stores a row delimiter, row identifier, column count, value sizes, and user data values. The row delimiter marks the start of a record. The row identifier is an internal DBMS pseudo-column. The column count represents the number of record columns. Value sizes are typically stored for strings, but not other data types (e.g., integers).

System Catalog. The system catalog refers to the data and metadata maintained by the DBMS. The system catalog is stored in tables and pages similar to user data. Sometimes the system catalog tables use domain datatypes that are not available to the user (e.g., the Object Name datatype in PostgreSQL). Examples of data and metadata stored in the system catalog are object types (e.g., table or index), object plaintext name (e.g., customer or employee), and object identifier, which is a unique identifier for each object (also stored in the user data page headers).

2.2 Database Auxiliary Objects

Users interact with DBMS tables; however, multiple copies of user data are stored in many other *internal* objects. Copies are stored in auxiliary objects (e.g., indexes, materialized views) that improve query performance or enforce constraints. Note: indexes are created both by explicit user commands or automatically by the DBMS itself (e.g., primary key or unique constraints).

Index value-pointer pairs are stored in pages just as table data in Figure 1. Index storage is structurally similar to a table that stores (value, pointer) records. Figure 2 displays an example of an index page, and how it references a table page. A table record pointer is stored with each city value. In this example, the pointer stores the table page identifier, 8, and the respective row identifier, 25.

Index Organized Tables. MySQL and SQLite create index organized tables (IOTs) by default, and IOTs are often used in other DBMSes under different names (e.g., IOT table in Oracle or index included columns in Microsoft SQL Server), so we incorporated IOTs into our data erasure method. An IOT is structured as a traditional B-Tree index on the primary key, and all remaining columns are included columns (or not used for ordering).

3 RELATED WORK

Retention of Deleted Database Data. Stahlberg et al. [33] demonstrated how deleted data remains in storage after table defragmentation and SQL `DELETE`, `UPDATE`, and `INSERT` commands. Their work was tested on PostgreSQL, MySQL, DB2, and SQLite. Lenard et al. [14] have investigated how deleted data propagates into database backups, showing that depending on a workload, a significant fraction of already-deleted rows can be backed up and permanently retained. Scope et al. [27, 30] have designed a data purging framework which relied on cryptographic erasure to destroy data across all backups. Their system generated encryption keys based on retention and purging policies; these keys could then be independently managed [28] in order to "remotely" erase associated data in backups by making it unrecoverable. Although Scope et al. approach was not limited to relational databases [29], it would was only destroy data that was covered by a purging policy (e.g., "erase after 3 years") and only after the expiration time. Wagner et al. [38, 40] analyzed what operations cause deleted data (or data abandoned

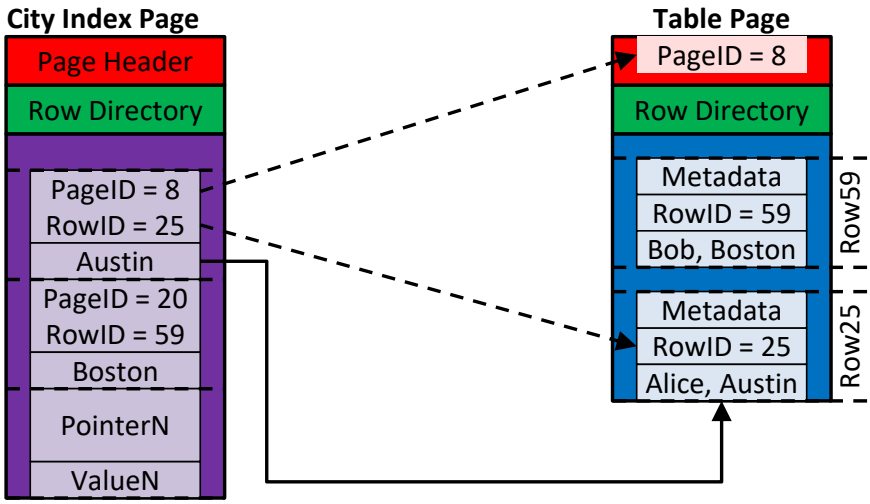

Figure 2: An example of how an index value references a table record.

in unallocated storage) including non-delete user actions, maintenance of auxiliary structures, and object rebuilds. Their work was evaluated across multiple relational databases.

Database Artifact Reconstruction. Related work explored the reconstruction of (deleted) database data. Forensic tools, such as Sleuth Kit [1] and EnCASE Forensic [6], are commonly used by digital investigators, but they reconstruct file system data and cannot parse DBMS files. While a multitude of built-in and 3^{rd} party recovery tools (e.g., [19, 22, 23]) can extract database storage, these tools only recover table records that are "active" (non-deleted). Frühwirt et al. presented a reconstruction method specifically for MySQL in [5]. Reconstruction of database content was later generalized across relational DBMSes with page carving [39, 41–43]. Page carving is similar to traditional file carving [7, 26] in that data, including deleted data, is reconstructed from disk images or RAM snapshots without using a live system. In this paper, we implement database-specific page carving steps to carve deleted data.

Data Erasure. Many solutions for disk-level data erasure exist [2, 8, 10, 25], but these cannot erase values or records in DBMS files. A few DBMS-specific erasure solutions were previously proposed. SQLite is the only DBMS that supports data erasure with the `secure_delete` setting [32], which is disabled by default. If enabled, `secure_delete` explicitly overwrites deleted data with zeros. Stahlberg et al. presented a well-founded method of data erasure for MySQL [33]. Their method modified MySQL source code to overwrite records as soon as they are free-listed. Both [33] and SQLite `secure_delete` use an erase-on-commit approach. Alternatively, `DBSanitizer` performs a batched erasure at a user-specified time. We evaluate both approaches in Section 9.

Grebhahn et al. proposed a set of SQL commands that, if supported, could overwrite unallocated storage [9]. Miklau et al. described the challenges facing a DBMS that supports targeted overwrites, including the lack of storage transparency [16]. Furthermore, Wagner et al. [40] emphasized the importance of transparency for data erasure by demonstrating deleted data that is created unbeknown to the user. `DBSanitizer` provides storage transparency giving users the knowledge of unallocated storage. Thus, users can be selective of which data is erased.

4 DELETED DATA

This section describes types of deleted data that we seek to erase: records, values, and pages. Deleted database data has been explored in more detail by related work (e.g., [5, 37, 40]); this section provides a brief overview. Section 2 discusses some of the terms and concepts fundamental to storage principles in a DBMS.

Records are the minimum deletion unit for a `DELETE` command; deleted records also arise from other operations, such as an `UPDATE` or a table rebuild. When a record is deleted, a DBMS either erases the record's directory pointer on the page or marks the record as "deleted". Figure 3 shows examples of a record, *Row1*, marked as deleted in the row data or in the row directory. In each scenario, modified page parts are labeled with a star; deletion always updates the row count and the checksum.

Deleted records are eventually overwritten based on a particular DBMS and its settings. For example, Oracle uses a percent page utilization threshold (user-configured) to determine when unallocated storage is reclaimed. Alternatively, PostgreSQL reclaims storage sooner if a new record fits into the unallocated space.

In practice, index values are not marked as unallocated space when a corresponding table record is deleted. Stale index values persist in storage, until the B-Tree index is explicitly rebuilt (with a special user command(s)); thus index copies survive long after the table record is overwritten. To identify stale index values, table records can be mapped back to index value-pointer pairs.

Unallocated pages occur when an object is deleted (`DROP`), rebuilt (e.g., `ALTER INDEX Name REBUILD ONLINE` in Oracle, `REINDEX TABLE Customer` in PostgreSQL), or defragmented (e.g., the PostgreSQL `VACUUM` command). DBMS storage architecture and operations dictate when unallocated pages are disassociated from DBMS files or

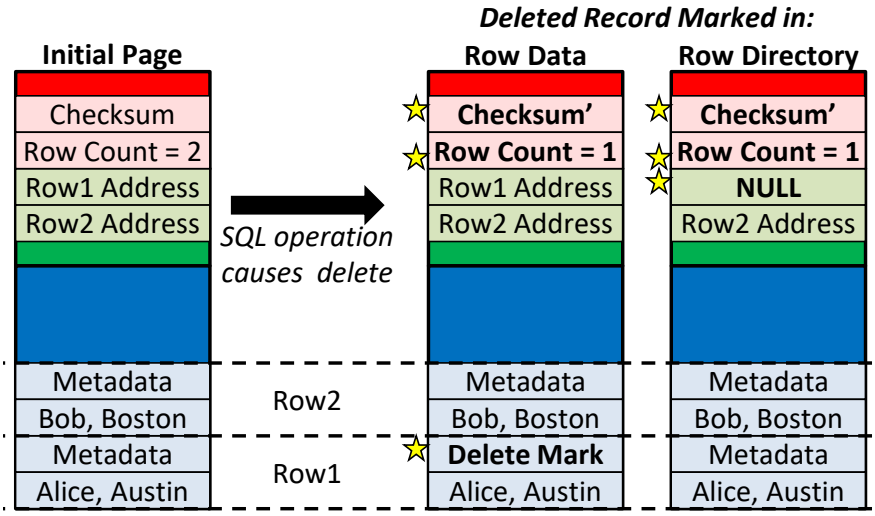

Figure 3: An example of a deleted record marked in either the row data or row directory.

reclaimed and overwritten by new objects. The page itself is typically not marked as unallocated. Instead, the table's record in the system catalog is marked as deleted, and the pages containing the table records are either free-listed in DBMS storage or reclaimed by the file system. Therefore, a page's object identifiers can be compared to the object identifier in the system catalog to determine if a page is unallocated.

Example: Unallocated Pages. An index rebuild results in a different representative storage-level outcomes for Oracle, MySQL, and PostgreSQL. All three build a new index structure. In Oracle, both the old and the new index structures remain in storage, leaving many old unallocated pages. In MySQL, the new index structure immediately overwrites the old index structure. If the new index needs less storage (which is very likely, because index is compacted), some old, "deleted" index pages remain. In PostgreSQL, the new index version is stored in a new file and the old index file is reclaimed by the file system. In all cases, old index pages have a different object identifier than the new index version pages.

5 DBMS FILE MODIFICATION

`DBSanitizer` directly modifies DBMS files without using the DBMS API (e.g., SQL). DBMSes do not provide an API to modify or even directly inspect the storage at the page level. When a DBMS file is modified, correct format and all relevant metadata must be considered to avoid corrupting the page (or the entire DBMS instance). Three things must be considered to perform live DBMS file modification correctly: 1) page checksum, 2) committed transactions, and 3) dirty pages. Taking into consideration committed transactions and dirty pages, we recommend that our erasure approach is applied to quiesced (i.e., not actively used by users) data structures (e.g., [4] in Oracle).

Page Checksum. Section 2 discussed that any page modification requires updating the corresponding checksum, even if DBMS is shut down. Figure 3 demonstrated that the checksum is updated when a record is deleted.

Transactions. Transactions help manage concurrent access to the DBMS and are used for recovery. All relational DBMSes guarantee that transactions are atomic, consistent, isolated, and durable (ACID). If page modifications are performed on a live DBMS structure, logs can identify any uncommitted transactions. Note that SQL DDL commands (e.g., `CREATE`, `DROP`, and `ALTER`) are automatically committed.

Dirty Pages. DBMSes do not immediately write modified pages from RAM to disk; a page that contains pending changes in RAM is known as dirty page. This is significant because a (manually) modified page can be overwritten when a dirty page is flushed to disk. This does not prevent our approach from working or pose any corruption risk, but may undo some of the work by `DBSanitizer`; Section 6 discusses how to address this possibility.

Example: Transactional Rollback Effect. A transaction can fail due to a conflict with another transaction or because user explicitly issued an `ABORT` command. This causes a `ROLLBACK`, restoring the database to the pre-transaction state. We loaded the `Supplier` table from SSBM benchmark (see Section 9) into an Oracle DBMS, started, and aborted a transaction that inserted 1000 rows. For all intents and purposes, an aborted insert transaction *never happened*. However, *all* 1000 records had their values copied into the indexes. Furthermore, after the DBMS was stopped, a (deleted) copy of all 1000 records could be recovered from the table files as well. This exemplifies just one of several non-obvious operations that create many additional (inaccessible) copies of user data.

6 DBSANITIZER

The remainder of the paper describes `DBSanitizer` in Algorithm 1, followed by an experimental evaluation in Section 9.

Algorithm 1 DBSanitizer Overview

```
1: ActiveTables ← a list of active user/system tables queried from the system catalog.
2: ActiveIndexes ← a list of active indexes queried from the system catalog.
3: N ← the number of table pages to be read
4: ErasedRecords ← an empty list for erased table record pointers and hash values
5: SortedIndex ← an empty dictionary to store approximately sorted indexes.
6: for each NPages ∈ DatabaseTableFiles do
7:     Carved ← NPages page carving output: PageIDs, Slot#s, and Records
8:     for each Page ∈ Carved do
9:         if Page.ObjectID ∈ ActiveTables then
10:            for each Record ∈ Page do
11:                if RecordStatus = Deleted then
12:                    Pointer ← ReconstructPointer()
13:                    ErasedRecords.append(Pointer, HashFunction(Record.Values))
14:                    EraseRecord()                                          ▷ Section 7.1
15:        else
16:            ErasePage()                                                    ▷ Section 7.3
17: for each IndexPage ∈ DatabaseIndexFiles do
18:    Carved ← page carving output of IndexPage.                             ▷ Section 3
19:    if Carved.ObjectID ∈ IndexTables then
20:        for each Value ∈ Carved do
21:            HashValue ← HashFunction(Value)
22:            SortedIndex.update(Value.Pointer, HashValue)
23:    else
24:        ErasePage()                                                        ▷ Section 7.3
25: for each Bucket ∈ SortedIndex do
26:    NPages ← carved pages where PageID ∈ Bucket
27:    for each IndexValue ∈ Bucket do
28:        Pointer ← IndexValue.PageID + IndexValue.Slot#
29:        if (Pointer ∈ NPages ∧ IndexValue ≠ NPages.Pointer.Value) ∨ (Pointer ∈ ErasedRecords) ∨ (Pointer ∉ NPages) then
30:            EraseIndexValue()                                              ▷ Section 7.2
```

Deployment Considerations. DBMSes support a restricted state (e.g., quiesced state in Oracle [4]), preventing ongoing transactions. For example, a quiesced state is enabled during backup to ensure backup integrity. However, as we demonstrate, file modifications can also be performed against a live DBMS if desired. Since dirty pages in a live system can overwrite changes made by `DBSanitizer`, some data may survive in RAM until it is evicted from cache.

Initialization: Alg 1, Lines 1 - 5. `DBSanitizer` first retrieves a list of tables and indexes from the system catalog (Section 2), and stores their object identifiers in the lists *ActiveTables* and *ActiveIndexes*. A number of table pages (parametrized by *N*) is read into memory at a given time. An empty list (*ErasedRecords*) for pointers and hash values from tables and an empty dictionary (*SortedIndex*) for approximately sorted indexes are initialized.

Table Data Erasure: Alg 1, Lines 6 - 16. The DBMS files are first evaluated for table data to erase. *N* number of pages are reconstructed using our implementation of page carving (Section 3) and stored as *Carved*. The object identifier, *ObjectID*, from each page in *Carved* determines if the page belongs to a table. If *ObjectID* is in *ActiveTables*, then the page is evaluated for deleted records. When a deleted record is found, the index pointer is reconstructed. The value and its pointer are then appended to *ErasedRecords*. We store a hash of the value instead of the value itself to avoid the risk of leaking data while erasing it. Next, the deleted record is erased (described in Section 7.1). If *ObjectID* is not in *ActiveTables*, then the page is an unallocated page, and the entire page is erased as described in Section 7.3.

Index Sorting: Alg 1, Lines 17 - 24. Next, the index value-pointer pairs are reconstructed, collected, sorted, and stored in *Carved*. The *ObjectID* from *Carved* determines if the page belongs to an index. If *ObjectID* is in *ActiveIndexes*, then index value-pointer pairs are parsed and sorted; If the *ObjectID* is not in *ActiveTables*, then the page is assumed to be unallocated page and erased (Section 7.3).

Index Data Erasure: Alg 1, Lines 25 - 30. The DBMS files are next evaluated for index data to erase. For each *Bucket* from *SortedIndex* the relevant table pages are carved. For each *IndexValue* within the *Bucket*, the index value is erased if the value-pointer pairs do not match between the table and index, its pointer is in *ErasedRecords*, or its pointer does not point to a valid record structure.

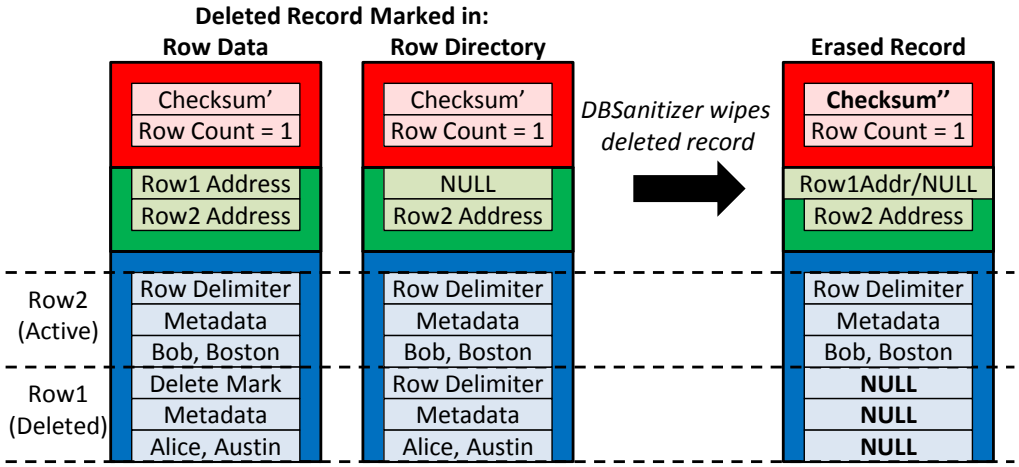

Figure 4: An example erasing a deleted record.

7 DATA ERASURE

7.1 Table Records

Once a page's ObjectID is associated with an active table, the page is evaluated for deleted records marked in the carved page output, *Carved*. Section 4 discussed deleted record identification.

To erase a record (regardless of how DBMS signifies row deletion in page storage), `DBSanitizer` overwrites the entire record (including the metadata) with `NULL` (decimal value 0). To avoid data corruption, `DBSanitizer` also updates the page checksum to reflect the change. Since the record was already marked deleted by a SQL command, no other page metadata must be updated. This operation is demonstrated in Figure 4. In this example, the record (`Alice, Austin`) is deleted (similar to the deleted record in Figure 3). Regardless of deleted record markings, `DBSanitizer` overwrites the record and its metadata with `NULL` and updates the checksum.

7.2 Value Erasure

After index values are sorted and deleted record pointers are collected, index pages are considered for erasure. If the page belongs to an active index, then it is evaluated for stale values (presented in Section 4). `DBSanitizer` finds stale index values by mapping the reconstructed pointer from the table data to the index value-pointer pairs. Similar to a record erasure, `DBSanitizer` overwrites a stale value, its pointer, and any metadata with `NULL`. Finally, the page checksum is updated.

7.3 Unallocated Pages Erasure

If a page cannot be associated with an active object (table or index), then the entire page is assumed to be unallocated storage. An unallocated page may either have an ObjectID that is `NULL` or not in the system catalog. Unallocated pages are the result of a deleted object (i.e., `DROP`), object rebuild, or object defragmentation. To erase a page, `DBSanitizer` overwrites the entire page row data with `NULL`.

For an unallocated table page, the index pointers do not need to be reconstructed, unlike for deleted records. This is because the page either belongs to a deleted or rebuilt table. If the table was deleted, then all indexes for that table were also deleted creating unallocated index pages. If the table was rebuilt, then the index was rebuilt, again, causing unallocated index pages.

8 ADDITIONAL COPIES OF DATA

Additional data copies may exist in storage outside of DBMS control, caused by activity that causes DBMS storage to be released to the OS, such as a deleted file from dropping a table. Since these pieces of data are not at risk of corruption, the entire page can be overwritten.

Transaction Logs. Write-ahead logs (WAL) record DBMS modifications in order to support transactional (ACID) guarantees, maintaining a history of transactions. WAL files do not store data in pages as other database objects. WALs cannot normally be disabled or easily modified, and require a special-purpose tool to be read (e.g., Oracle LogMiner or PostgreSQL pg_xlogdump). DBMSes (including Oracle, MySQL, PostgreSQL, and SQL Server) allow the administrator to switch to a new WAL file and delete old WAL files. These deleted WAL files can then be erased using methods of file erasure at the OS level without concern of corrupting storage. For example, an administrator can switch from log file A to log file B. To implement this operation in Oracle:

1) `ALTER DATABASE ADD LOGFILE (`path/logB.rdo')`
2) Executed transaction logs are placed in `logB.rdo` file
3) `ALTER DATABASE DROP LOGFILE MEMBER `path/logA.rdo'`

The file `path/logA.rdo` can then be erased from storage using standard OS-level sanitization methods.

9 EXPERIMENTS

This section evaluates `DBSanitizer` with three experiments. Section 9.1 uses PostgreSQL 9.6 to demonstrate that DBMS files can be directly modified without corrupting storage. Section 9.2 uses several SQLite instances to compare `DBSanitizer` to `secure delete`. Section 9.3 uses PostgreSQL 9.6 to measure the costs associated with `DBSanitizer`. Tables `Lineorder` and `Customer` at Scale 4 (2.4M and 120K rows, respectively) were generated from the Star Schema

Benchmark SSBM [20]. SSBM combines a realistic distributed data (maintaining data types and cross-column correlations) with a synthetic data generator that creates datasets at different scale.

9.1 DBSanitizer Demonstration

This experiment demonstrates that DBSanitizer can effectively erase deleted data without corruption. Although we anticipate the storage to be quiesced during erasure, it is also possible to erase contents of a live DBMS. If DBSanitizer can alter live storage, we expect a DBMS vendor (with access to source code) to have no difficulty applying the same techniques.

We created a database file containing the Customer table. We overwrote the values from a deleted record with NULL and updated the page checksum. The modification was performed directly in the PostgreSQL file using a Python script with sudo privileges. Note that this change did not remove the record from storage but just made it blank (our goal is to erase deleted data, not to alter records). As a result, the values were completely removed from the file. The erased values were not found by reading the file with Python or using grep. In order to stress-test DBSanitizer, we performed this experiment on a live (rather than a read-only) database instance. Thus, it retained the cached page in RAM and SELECT queries initially still returned the original record values based on cached pages. We simulated a page refresh by loading a large new table into the DBMS. When the cached copy of the page was discarded, the SQL queries started to return blank values consistent with disk contents.

This experiment illustrated that erasure is possible on a live system without corrupting storage. Moreover, performing this operation on a quiesced (i.e., read-only) DBMS is simpler. Data sanitization on disk is reliable and instantaneous – as soon as we modified the page, the values were no longer recoverable from the file.

9.2 Effectiveness

This experiment compares the erasure capabilities of DBSanitizer's batched approach and the erase-on-commit approach used in SQLite's secure delete. Both of these approaches are intended to only erase data within DBMS storage. Section 4 discussed actions that can create copies of (erased) records. Note that copies of table records may also exist outside of DBMS storage in paging files (i.e., copies of RAM written to disk) or deleted file storage released back to the file system (e.g., dropping table in a DBMS such as PostgreSQL deletes the corresponding file). Both in-DBMS and outside of DBMS control value copies are accounted for in our experiment.

Setup & Procedure. We created a total of three SQLite DBMS instances: I_1 used DBSanitizer, and I_2 & I_3 enabled secure delete at different points in time. The secure delete feature was initially turned off upon instance creation since this is the default for SQLite. Each instance ran on a separate 100MB partition on a external hard drive previously never used. We note that since SQLite is a lightweight DBMS, it uses a single file to store all objects - other transaction log files exist only when the instance is open. Finally, table CUSTOMER (Scale 4, 120K rows) was loaded into each instance with indexes on the Name and City columns.

We next describe experimental steps. For each step, the database instance was opened, the described operation(s) was performed, the instance connection was closed to flush the DBMS buffer cache, the partition was imaged using the dd command, and the SQLite file was copied to a separate storage device.

- T_1 Initial instance setup.
  → **Secure delete enabled for I_3.**
- T_2 Table CUSTOMER was created with indexes on the Name & City columns. The data was loaded.
- T_3 24K (20%) rows updated on a column with no index.
  → **Secure delete enabled for I_2.**
- T_4 455 City records deleted:
  `DELETE FROM CUSTOMER WHERE City = 'CANADA␣5';`
- T_5 The same records updated in T_3 updated again on a column with no index.
- T_6 A SELECT query (using a full table scan) was ran.
  → **DBSanitizer was ran against I_1**
- T_7 The instance was opened with no further activity.

Results. Table 2 summarizes the distribution of data copies for two sample deleted values found in the SQLite file (i.e., under DBMS control) and across the disk image (released by the DBMS to the OS). *City* refers to the value 'CANADA 5', the delete predicate condition at T_4. *Name* refers to a value, 'CUSTOMER#000000434', which was in a record deleted at T_4.

T_2. I_3 resulted in a different number of *City* and *Name* values versus I_1 and I_2. Since data was loaded after the indexes were created, the B-Tree was forced to rebuild as data was loaded. Therefore, we conclude that secure delete in I_3 successfully erased the stale index values left behind by the index rebuilds.

T_3. We observed two interesting changes at T_3. First, copies of updated records were generated in the SQLite file for I_1 and I_2. This is explained by an UPDATE that writes the new version of the record to a new location in storage and marks the old version as deleted. The secure delete in I_3 successfully erased these old record versions within the SQLite file. Second, we found a large number of value copies on the disk image. This is explained by the fact that in most cases, the new version of the page is not written to the same disk sector (thereby creating a new copy of the page). A new page version written to a new sector was often compacted and no longer included the deleted records. The old version of the page is no longer part of the SQLite file. In this case, the deleted values in old pages were not erased by secure delete for I_3 until the next instance startup (likely due to file system caching). Note that the active records in an old discarded page (outside of the SQLite file) were not erased by secure delete, remaining vulnerable to theft.

T_4. Following the DELETEs, secure delete successfully erased all of the deleted data in the I_3 SQLite file. Secure delete did not erase all of the deleted data in I_2. This is because deleted values were created at T_3, before secure delete was enabled, and secure delete does not retroactively erase data. Some deleted data was overwritten in I_1 by the page compaction we observed, which was previously mentioned in the discussion of T_3. An increase in deleted data was found in the disk image, which is similar to what was observed at T_3; some pages were written to different sectors on disk, and secure delete did not erase some of the deleted data until the next instance startup.

T_5. Following the UPDATEs some updated records written to new locations, overwriting the previous unallocated space. These newly written records erased some of the deleted data in the I_1 and I_2

	SQLite File						Disk Image					
	City			Name			City			Name		
Time	I_1	I_2	I_3	I_1	I_2	I_3	I_1	I_2	I_3	I_1	I_2	I_3
T_1	0	0	0	0	0	0	0	0	0	0	0	0
T_2	921	921	910	3	3	2	921	921	910	3	3	2
T_3	944	944	910	3	3	2	1399	1399	1365	4	4	3
T_4	672	25	0	3	1	0	1943	1295	1250	6	4	2
T_5	218	16	0	2	1	0	846	42	3	3	1	0
T_6	218	16	0	2	1	0	846	42	3	3	1	0
T_7	0	16	0	0	1	0	846	42	3	3	1	0

Table 2: Experiment 2 deleted values. I_1 applies DBSanitizer at T_6. I_2 and I_3 enable secure delete at T_3 and T_3, respectively.

SQLite files. Therefore, we observed a drop in number of the detected values in the I_1 and I_2 database files between T_4 and T_5.

T_6. No notable changes for read-only queries.

T_7. Finally, `DBSanitizer` successfully removed all of the deleted data from the I_1 SQLite file. No deleted data remained in I_3, which began with `secure delete` enabled. I_2 still had deleted data in the SQLite file because `secure delete` does not retroactively erase existing deleted data.

Conclusion. SQLite's `secure delete` was effective at erasing data within DBMS storage when enabled at the instance initialization. However, if this feature is not enabled at initialization time (or disabled at any point) deleted data can persist in storage since `secure delete` does not retroactively erase data. `DBSanitizer` can be invoked to erase all currently deleted data at a specified point in time, without synchronizing with other database operations. After executing `DBSanitizer`, the amount of deleted data was at zero, identical to `secure delete` in I_3.

While `DBSanitizer` had the most deleted data remain outside of the SQLite file across the disk partition, all partitions (I_1, I_2, and I_3) had some deleted data persist outside of the DBMS-controlled storage. Therefore, each approach (including a permanently enabled `secure delete` in I_3) requires continuous disk-level sanitization. A software tool that meets the `data erasure` guidelines of the International Data Sanitization Consortium [12] can be used to erase unallocated storage areas on disk. Since it may not be possible to identify released DBMS data in disk storage, we recommend periodically sanitizing the entire unallocated disk storage. This sanitization cost is similar for all three instances, regardless of the amount of deleted data; the entire disk image must be scanned for sectors containing deleted data.

9.3 Feasibility of `DBSanitizer`

This experiment measures `DBSanitizer` costs. Part-A provides runtimes measured against database files containing a different number of deleted table records. Part-B provides index sorting runtimes (in order to sanitize index values, we join indexes and tables they reference – index is sorted to speed up the join performance). This experiment used table `Lineorder` Scale 4 (24M records, 2.4 GB). We created a secondary index on the `Orderdate` and `Revenue` columns.

We did not compare our runtimes to the `secure delete` in SQLite for a few reasons. First, the cost of our approach can be increased or reduced by changing the sanitization batching frequency. Second, SQLite is not representative of a robust DBMS that handles many users and large quantities of data.

Deleted Records	Affected Pages	Runtime(MB/s)
0.0%	0.0%	0.96
0.1%	7.4%	0.98
0.2%	14.3%	1.00
0.5%	32.0%	1.03
1.0%	53.5%	1.06
5.0%	97.2%	1.10

Table 3: `DBSanitizer` Page carving and writing costs.

Part-A.. We created a series of PostgreSQL files containing table `Lineorder`. We deleted different sets of records from each `Lineorder` table. Records were chosen randomly for deletion based on their primary key values. Table 3 summarizes the deletes performed and the results. Our implemented carving processed the SQLite file containing no deletes at .96 MB/s. Each 4 KB page contained about 78 records, thus not every deleted record occurred in a separate page. For example, when 5% of the records were deleted, 97% of the table pages were modified. Our results show that the cost to sanitize the tables is primarily based on the number of pages modified rather than the amount of deleted records in the file. A targeted sensitive data erase operation would access fewer pages (out of all pages with deleted data), reducing the sanitization cost.

Bucket Size	Buckets	Orderdate	Revenue
5K pages	63	1366 sec	1380 sec
10K pages	32	1121 sec	1131 sec
50K pages	7	932 sec	945 sec
100K pages	4	909 sec	926 sec
200K pages	2	903 sec	918 sec

Table 4: Index sorting costs with varying bucket sizes.

Part-B.. To evaluate approximate sorting with respect to bucket size, we used the carved output from PostgreSQL files containing table `Lineorder`, a secondary index on `Revenue` column, and a secondary index on `Orderdate` column. Table 4 summarizes the performance results. As the number of buckets decreases the time to sort the data decreases. However, a bucket must fit into memory, so increasing of bucket sizes is limited by available RAM.

10 CONCLUSION

This paper presented `DBSanitizer`, a batched data erasure approach, as a template for DBMS vendors to support data erasure. `DBSanitizer` is executed at a time specified by the user and does not require any maintenance. Our experiments demonstrated that `DBSanitizer` is a cost effective alternative to the "erase-on-commit" approach. We have also compared the erasure capabilities of both `DBSanitizer` and SQLite's `secure delete`. If `secure delete` is enabled from instance creation, it is effective at erasing data and has the same end result as `DBSanitizer`. However, if `secure delete` is temporarily disabled, any deleted data in storage cannot be retroactively erased with `secure delete`.

While our goal in this paper is to prevent data exposure to theft, it is possible for these methods to be used maliciously. Two specific actions include record removal and erasure of active records. Record removal marks the metadata of a record fooling the DBMS into recognizing it as unallocated storage. Erasure of an active record not only creates unallocated storage, but also explicitly overwrites the record leaving behind little evidence of tampering. Our future work will seek to detect these malicious operations.

ACKNOWLEDGMENTS

This work was partially funded by the Louisiana Board of Regents Grant AWD-10000153 and by US National Science Foundation Grant IIP-2016548.

REFERENCES

[1] Brian Carrier. 2011. The Sleuth Kit. *http://www.sleuthkit.org/sleuthkit/* (2011).
[2] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. 2005. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation.. In *USENIX Security Symposium*. 22–22.
[3] European Union. 2018. The General Data Protection Regulation (GDPR). https://www.eugdpr.org/
[4] Steve Fogel. 2010. Oracle Database Administrator's Guide: Quiescing a Database. \https://docs.oracle.com/cd/E18283_01/server.112/e17120/start004.htm
[5] Peter Frühwirt, Marcus Huber, Martin Mulazzani, and Edgar R Weippl. 2010. InnoDB database forensics. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. IEEE, 1028–1036.
[6] Lee Garber. 2001. Encase: A case study in computer-forensic technology. *IEEE Computer Magazine January* (2001).
[7] Simson L Garfinkel. 2007. Carving contiguous and fragmented files with fast object validation. *digital investigation* 4 (2007), 2–12.
[8] Simson L Garfinkel and Abhi Shelat. 2003. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security & Privacy* 99, 1 (2003), 17–27.
[9] Alexander Grebhahn, Martin Schäler, and Veit Köppen. 2013. Secure Deletion: Towards Tailor-Made Privacy in Database Systems.. In *BTW Workshops*. 99–113.
[10] Peter Gutmann. 1996. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, Vol. 14. 77–89.
[11] Ryan Harris. 2006. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. *digital investigation* 3 (2006), 44–49.
[12] International Data Sanitization Consortium. 2017. Data Sanitization Terminology and Definitions. https://www.datasanitization.org/data-sanitization-terminology/
[13] Gary C Kessler. 2007. Anti-forensics and the digital investigator. In *Australian Digital Forensics Conference*. Citeseer, 1.
[14] Ben Lenard, Alexander Rasin, Nick Scope, and James Wagner. 2021. What is lurking in your backups?. In *ICT Systems Security and Privacy Protection: 36th IFIP TC 11 International Conference, SEC 2021, Oslo, Norway, June 22–24, 2021, Proceedings*. Springer, 401–415.
[15] Ming Di Leom, Kim-Kwang Raymond Choo, and Ray Hunt. 2016. Remote wiping and secure deletion on mobile devices: A review. *Journal of forensic sciences* 61, 6 (2016), 1473–1492.
[16] Gerome Miklau, Brian Neil Levine, and Patrick Stahlberg. 2007. Securing history: Privacy and accountability in database systems.. In *CIDR*. Citeseer, 387–396.
[17] National Institute of Standards and Technology. 2006. Guidelines for Media Sanitization.
[18] National Security Agency Central Security Service. 2014. NSA/CSS Storage Sanitization Manual.
[19] OfficeRecovery. 2017. Recovery for MySQL. \http://www.officerecovery.com/mysql/
[20] Patrick O.Neil, Elizabeth O.Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Performance evaluation and benchmarking*. Springer, 237–252.
[21] Parliament of Canada. 2000. The Personal Information Protection and Electronic Documents Act (PIPEDA). \https://www.priv.gc.ca/en/privacy-topics/privacy-laws-in-canada/
[22] Percona. 2018. Percona Data Recovery Tool for InnoDB. https://launchpad.net/percona-data-recovery-tool-for-innodb.
[23] Stellar Phoenix. 2018. DB2 Recovery Software. http://www.stellarinfo.com/database-recovery/db2-recovery.php.
[24] Joel Reardon, David Basin, and Srdjan Capkun. 2013. Sok: Secure data deletion. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 301–315.
[25] Joel Reardon, Srdjan Capkun, and David Basin. 2012. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 17–17.
[26] Golden G Richard III and Vassil Roussev. 2005. Scalpel: A Frugal, High Performance File Carver.. In *DFRWS*. Citeseer.
[27] Nick Scope, Alexander Rasin, Ben Lenard, Karen Heart, and James Wagner. 2022. Harmonizing Privacy Regarding Data Retention and Purging. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management*. 1–12.
[28] Nick Scope, Alexander Rasin, Ben Lenard, and James Wagner. 2023. Compliance and Data Lifecycle Management in Databases and Backups. In *International Conference on Database and Expert Systems Applications*. Springer, 281–297.
[29] Nick Scope, Alexander Rasin, Ben Lenard, James Wagner, and Karen Heart. 2022. Purging Compliance from Database Backups by Encryption. *Journal of Data Intelligence* 3, 1 (2022).
[30] Nick Scope, Alexander Rasin, James Wagner, Ben Lenard, and Karen Heart. 2021. Purging data from backups by encryption. In *Database and Expert Systems Applications: 32nd International Conference, DEXA 2021, Virtual Event, September 27–30, 2021, Proceedings, Part I 32*. Springer, 245–258.
[31] Damon W Silver and Gregory C Brown. 2022. $600,000 reasons to review your shield act compliance program: NY attorney general announces significant settlement stemming from email data breach. https://www.natlawreview.com/article/600000-reasons-to-review-your-shield-act-compliance-program-ny-attorney-general
[32] SQLite. 2018. PRAGMA statements. \{https://www.sqlite.org/pragma.html#pragma_secure_delete}
[33] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. 2007. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, Citeseer, 91–102.
[34] U.S. Centers for Disease Control and Prevention. 1996. Health Insurance Portability and Accountability Act (HIPAA). \https://www.hhs.gov/hipaa/index.html
[35] U.S. Department of Education. 1974. The Family Educational Rights and Privacy Act (FERPA). \https://www2.ed.gov/policy/gen/guid/fpco/ferpa/index.html
[36] U.S. Internal Revenue Service. 2017. Media Sanitization Methods. \https://www.irs.gov/privacy-disclosure/media-sanitization-methods
[37] James Wagner. 2020. Auditing database systems through forensic analysis. (2020).
[38] James Wagner, Alexander Rasin, Boris Glavic, Karen Heart, Jacob Furst, Lucas Bressan, and Jonathan Grier. 2017. Carving database storage to detect and trace security breaches. *Digital Investigation* 22 (2017), S127–S136.
[39] James Wagner, Alexander Rasin, and Jonathan Grier. 2015. Database forensic analysis through internal structure carving. *Digital Investigation* 14 (2015), S106–S115.
[40] James Wagner, Alexander Rasin, and Jonathan Grier. 2016. Database image content explorer: Carving data that does not officially exist. *Digital Investigation* 18 (2016), S97–S107.
[41] James Wagner, Alexander Rasin, Karen Heart, Rebecca Jacob, and Jonathan Grier. 2019. Db3f & df-toolkit: The database forensic file format and the database forensic toolkit. *Digital Investigation* 29 (2019), S42–S50.
[42] James Wagner, Alexander Rasin, Karen Heart, Tanu Malik, and Jonathan Grier. 2020. DF-toolkit: interacting with low-level database storage. *Proceedings of the VLDB Endowment* 13, 12 (2020).
[43] James Wagner, Alexander Rasin, Tanu Malik, Karen Heart, Hugo Jehle, and Jonathan Grier. 2017. Database forensic analysis with DBCarver. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research*.
[44] Ben Wolford. 2020. Everything you need to know about the "Right to be forgotten". https://gdpr.eu/right-to-be-forgotten/