



SUNIMOD FIELD NOTES

Deleted Does Not Mean Gone: The Customer Data Still Hiding in Your Database

Deleted records can leave readable data in database files, transaction logs, and backups. Learn how to test the gap and build a verified deletion workflow.

BY Christopher Jacobson

PUBLISHED October 15, 2026

TOPIC CISSP, Domain 2

<https://sunimod.com/deleted-customer-data-sqlite-remanence/>

The business problem: Your organization needs to know not just whether information disappeared from a screen, but which copies still exist, who can access them, and when they should expire.

The solvable project: Build a verified data-retention and deletion workflow that connects application behavior, database configuration, backups, support exports, and evidence of completion. The goal is an understandable, testable process—not a stronger-sounding delete button.

- [Why deleted data can remain readable](#)
- [How the exposure happens](#)
- [Reproduce the deletion gap safely](#)
- [Fix the database layer](#)
- [Manage the entire data lifecycle](#)
- [Potential business repercussions](#)
- [Verify the controls](#)
- [Turn the findings into a Sunimod project](#)

Why deleted data can remain readable

Imagine a fictional service company whose scheduling application stores customer email addresses and private job notes in a SQLite database. A developer removes an old customer, confirms that the customer search returns no results, and sends a supposedly cleaned database to an outside support contractor.

The important question is what “cleaned” means. Removing a row from the current database view is not necessarily the same as removing its previous bytes from the file. SQLite’s documentation explicitly describes this distinction and the possibility of recovering deleted content. [SQLite: VACUUM](#).

Data remanence is information that remains after an operation you expected to remove it. In this example, the security weakness is the company’s assumption that a successful application deletion makes the entire database safe to share. This is not a claim that SQLite is broken or that every SQLite deployment exposes deleted records.

The application view is not the storage view

A soft delete usually changes application state, such as marking an account inactive. A hard delete removes a row from normal query results. Neither label, by itself, establishes what happened to every storage representation or independent copy.

SQLite organizes a database into pages and can retain unused space for reuse. A database file is therefore more than the rows returned by your current queries: it also contains structural information and potentially unused areas. [SQLite: database file format](#).

Think of a hotel deleting a guest’s name from its booking index while an old registration card remains in a drawer. The current lookup is accurate; the broader claim that no guest information remains is not. The demonstration below shows that distinction with actual database operations.

Transaction logs can retain earlier content

In write-ahead logging mode, SQLite appends changes to a separate file ending in `-wal`. Multiple transactions can occupy that file before their changes are transferred into the main database through a checkpoint. The current logical state and earlier page images can therefore coexist across these files. [SQLite: write-ahead logging](#).

That creates a second question: even when the current page has been scrubbed, does an earlier log entry still contain the information? A check of only the main database file cannot answer that question.

Backups are a separate retention problem

An earlier backup can contain an ordinary, fully queryable customer record. That is different from recovering leftover bytes: the backup preserves an earlier database state. SQLite's backup API is designed to produce a consistent database snapshot. [SQLite: backup API](#).

Consequently, a clean live database does not establish that an older backup, exported report, or support attachment is clean. Each copy needs its own handling decision.

How the exposure happens

In our fictional company, the failure develops through four ordinary-looking decisions:

1. **Sensitive information enters the application.** Customer details and job notes are written to a database. The team treats the application as the only important location.
2. **A deletion is verified only through the interface.** The customer disappears from searches, so the company treats the data as gone without checking the database's storage behavior.
3. **A retained artifact crosses an access boundary.** A database is attached to a support ticket, copied to a less-protected development machine, or included in an exposed archive.
4. **The recipient examines the artifact rather than using the application.** They inspect remaining bytes, transaction-log content, or a historical backup and find information the company thought it had removed.

This chain is a threat scenario, not a report of a particular breach. Its key condition is access to an artifact containing readable information. The recipient might be malicious, or they might be an authorized support provider who was never supposed to receive that customer's data.

This is not a remote login bypass. The example requires access to the relevant unencrypted files, or equivalent access to their decrypted contents. Someone who can only use a properly restricted customer portal does not automatically gain raw database access. Preventing artifact exposure and minimizing what artifacts contain are separate controls.

For this scenario, encryption needs a precise threat model. Protecting a powered-off device does not necessarily protect a database copied through an already-authorized, running system. OWASP distinguishes encryption layers and explains why the appropriate layer depends on the access an attacker might obtain. [OWASP: cryptographic storage guidance](#).

Reproduce the deletion gap safely

The following Python program creates three temporary SQLite databases, inserts entirely fictitious information, deletes it, and checks the resulting files. It makes no network connections, accepts no production database path, and removes its temporary directory when finished. The operations are real; only the customer information is synthetic.

On Ubuntu, save the complete program as `deletion_lab.py`. It uses Python's standard-library SQLite interface, including its database backup method. [Python: sqlite3 documentation](#).

Controlled conditions matter: Case A explicitly turns secure deletion off to make the behavior visible. Case B turns it on. Case C enables write-ahead logging and disables automatic checkpoints only inside the temporary lab. These are demonstration settings, not a recommendation to disable protections in your application. SQLite build options can change defaults. [SQLite: compile-time options](#).

Python · Reproduce database, backup, and transaction-log retention

```
#!/usr/bin/env python3
"""Observe SQLite deletion using temporary, entirely fictitious records."""

from contextlib import closing
from pathlib import Path
import sqlite3
import tempfile

MARKER = "SUNIMOD-LAB-PRIVATE-NOTE-7D914B2C"

def marker_present(path: Path) -> bool:
    return path.is_file() and MARKER.encode("utf-8") in path.read_bytes()

def row_count(connection: sqlite3.Connection) -> int:
    return connection.execute("SELECT COUNT(*) FROM customers").fetchone()[0]

def seed(connection: sqlite3.Connection) -> None:
    connection.execute(
        "INSERT INTO customers(id, email, private_note) VALUES (?, ?, ?)",
        (1, "alex@example.com", MARKER),
    )
    connection.commit()

def remove_customer(connection: sqlite3.Connection) -> None:
    connection.execute("DELETE FROM customers WHERE id = ?", (1,))
    connection.commit()

def create_database(
    path: Path, *, secure_delete: bool, wal: bool = False
) -> sqlite3.Connection:
    connection = sqlite3.connect(path, timeout=5.0)
    try:
        mode = "WAL" if wal else "DELETE"
        actual = connection.execute(f"PRAGMA journal_mode={mode}").fetchone()[0]
        if actual.lower() != mode.lower():
            raise RuntimeError(f"Requested journal mode {mode}; received {actual}")
        setting = "ON" if secure_delete else "OFF"
        connection.execute(f"PRAGMA secure_delete={setting}")
        observed = connection.execute("PRAGMA secure_delete").fetchone()[0]
        if observed != int(secure_delete):
            raise RuntimeError("SQLite did not apply the secure_delete setting")
        connection.execute("PRAGMA auto_vacuum=NONE")
```

```

    if wal:
        connection.execute("PRAGMA wal_autocheckpoint=0")
    connection.execute(
        "CREATE TABLE customers ( "
        "id INTEGER PRIMARY KEY, "
        "email TEXT NOT NULL, "
        "private_note TEXT NOT NULL)"
    )
    connection.commit()
    return connection
except BaseException:
    connection.close()
    raise

def main() -> None:
    with tempfile.TemporaryDirectory(prefix="sunimod-deletion-lab-") as directory:
        root = Path(directory)

        ordinary = root / "ordinary.db"
        with closing(create_database(ordinary, secure_delete=False)) as connection:
            seed(connection)
            remove_customer(connection)
            print(f"A: live rows after DELETE = {row_count(connection)}")
            print(f"A: marker in database file = {marker_present(ordinary)}")
            connection.execute("VACUUM")
            print(f"A: marker after VACUUM = {marker_present(ordinary)}")

        protected = root / "protected.db"
        backup = root / "before-deletion.db"
        with closing(create_database(protected, secure_delete=True)) as connection:
            seed(connection)
            with closing(sqlite3.connect(backup)) as destination:
                connection.backup(destination)
            remove_customer(connection)
            print(f"B: live rows after DELETE = {row_count(connection)}")
            print(f"B: marker in database file = {marker_present(protected)}")
        with closing(sqlite3.connect(backup.as_uri() + "?mode=ro", uri=True)) as copy:
            print(f"B: rows in earlier backup = {row_count(copy)}")
        print(f"B: marker in earlier backup = {marker_present(backup)}")

        logged = root / "logged.db"
        wal_file = root / "logged.db-wal"
        with closing(create_database(logged, secure_delete=True, wal=True)) as connection:
            seed(connection)
            remove_customer(connection)
            print(f"C: live rows after DELETE = {row_count(connection)}")
            print(f"C: marker in WAL before checkpoint = {marker_present(wal_file)}")
            status = connection.execute("PRAGMA wal_checkpoint(TRUNCATE)").fetchone()
            print(f"C: checkpoint result = {status}")
            if status != (0, 0, 0):
                raise RuntimeError("The isolated WAL checkpoint did not complete")
            print(f"C: marker in WAL after checkpoint = {marker_present(wal_file)}")
            print(f"C: marker in database after checkpoint = {marker_present(logged)}")

        print("Temporary lab files removed; no production data was used.")

if __name__ == "__main__":
    main()

```

Bash · Run the isolated demonstration on Ubuntu

```
python3 deletion_lab.py
```

Read the results

A run of this program produced the following output:

Plaintext · Observed results from the demonstration

```
A: live rows after DELETE = 0
A: marker in database file = True
A: marker after VACUUM = False
B: live rows after DELETE = 0
B: marker in database file = False
B: rows in earlier backup = 1
B: marker in earlier backup = True
C: live rows after DELETE = 0
C: marker in WAL before checkpoint = True
C: checkpoint result = (0, 0, 0)
C: marker in WAL after checkpoint = False
C: marker in database after checkpoint = False
Temporary lab files removed; no production data was used.
```

Case A demonstrates residual bytes. The current table contains zero rows, but the unique marker is still readable in the database file. Rebuilding this ordinary-table database with `VACUUM` removes the marker from the file examined by the lab.

Case B demonstrates an independent copy. Secure deletion removes the marker from the current database in this test, while the earlier backup still contains a normal customer row. Nothing about deleting from the live database retroactively modifies that separate file.

Case C demonstrates a transaction-log gap. The customer is absent from normal queries, but an earlier WAL frame retains the marker. After the isolated truncate checkpoint completes, the marker is absent from both files checked by the program.

The marker search models the recipient's simplest opportunity: reading file contents without consulting the application's current record list. It is not a full forensic recovery tool, and it does not need to reconstruct an entire customer table to demonstrate an information disclosure.

The useful result is evidence, not a blanket guarantee. The test proves where this marker was present or absent under these conditions. It does not prove that every possible record, encoding, database extension, storage snapshot, or physical storage block has been sanitized.

Keep the raw-file inspection inside this isolated lab. It is not a procedure for examining a concurrently used production database or preserving incident evidence. Ordinary removal of the temporary directory is also not a media-sanitization claim.

Fix the database layer without creating a recovery problem

Configure the actual writing connections

For ordinary SQLite tables, `secure_delete=ON` instructs SQLite to overwrite deleted content with zeros. Set it before relevant deletes or updates; enabling it afterward is not a retroactive cleanup. The `FAST` setting is not equivalent to `ON` and may retain traces on freelist pages. [SQLite: secure_delete documentation](#).

The control belongs in the application's connection setup—not merely in a separate database console used during an audit. The following complete helper opens an existing database, requests secure deletion, verifies the setting, and closes the connection if initialization fails:

Python · Open an existing database with verified secure-deletion configuration

```
from pathlib import Path
import sqlite3

def connect_for_private_data(database: str | Path) -> sqlite3.Connection:
    """Open an existing SQLite file and enable scrubbing on this connection."""
    path = Path(database).expanduser().resolve(strict=True)
    if not path.is_file():
        raise ValueError("The database path must identify an existing file")

    connection = sqlite3.connect(
        path.as_uri() + "?mode=rw", uri=True, timeout=10.0
    )
    try:
        connection.execute("PRAGMA secure_delete=ON")
        if connection.execute("PRAGMA main.secure_delete").fetchone() != (1,):
            raise RuntimeError("Required secure_delete setting was not applied")
        return connection
    except BaseException:
        connection.close()
        raise
```

Use this pattern in every application process that performs relevant writes, including background workers and maintenance jobs. Verify attached databases separately where applicable. The caller still owns transaction boundaries and must close the returned connection. This helper intentionally does not alter journal mode, delete customer records, or pretend to enforce authorization and retention policy.

Measure the effect on your real workload before rollout. Scrubbing can require additional work; SQLite documents the associated performance tradeoff. [SQLite: secure-deletion build option](#).

Coordinate transaction-log cleanup

A checkpoint normally transfers WAL changes into the database; it does not necessarily truncate the log. Treat the existence and accessibility of transaction logs as part of the data-handling design. [SQLite: checkpointing and WAL reuse](#).

During an approved maintenance procedure, after coordinating readers and writers, SQLite provides a truncate checkpoint:

SQL · Request a truncate checkpoint through SQLite

```
PRAGMA main.wal_checkpoint(TRUNCATE);
```

For a WAL database, successful completion truncates the log. Active connections can prevent completion, so inspect the returned status instead of treating “the command ran” as success. In the lab, `(0, 0, 0)` reports successful completion with no remaining WAL frames. [SQLite: wal_checkpoint return values](#). The [checkpoint API documentation](#) explains locking and busy conditions.

This is not a command to append blindly to every web request. A production design needs workload-specific coordination, timeout handling, and a plan for delayed completion. Nor does a successful checkpoint erase independent backups or scrub every preexisting free area of the main database.

Never substitute manual deletion of a live `-wal` or `-journal` file. These files can contain information required for consistency and crash recovery. Removing or mismatching them can corrupt a database. Use supported database operations and a coordinated maintenance process. [SQLite: how database corruption can occur](#).

Address old content and specialized indexes

For historical remnants in ordinary tables, evaluate a controlled `VACUUM` operation after the intended logical deletions. Plan for locking, recovery, and temporary disk space: SQLite documents that the operation can require as much as twice the original database size in additional free space. Its scope is the database being rebuilt, not every historical copy. [SQLite: VACUUM behavior and requirements](#).

Full-text search needs separate review. FTS5 can retain older index entries until merge operations remove them. Its own `secure-delete` option is distinct from the core setting, and the documentation describes both the additional control and compatibility implications. Do not apply an ordinary-table test result to a full-text index without testing that index's behavior. [SQLite: FTS5 secure-delete configuration](#).

The practical lesson is to inventory how your application represents information before selecting a cleanup method. A customer note might exist as a row, an index entry, an attachment, and an exported report. Fixing one representation does not establish that the others were handled.

Manage the entire data lifecycle, not just one file

Assign ownership and define retention

Start with a record category that matters to the business: completed customer jobs, abandoned applications, support attachments, or former employee records. Identify a business owner who can approve why the information is needed and an operator responsible for implementing the handling rules.

For each category, document its purpose, sensitivity, storage locations, permitted recipients, retention trigger, and disposal method. Distinguish a period measured from collection from one measured from the end of a customer relationship. Those are different rules and can produce different deletion dates.

This approach follows the FTC's guidance to understand your information, retain only what is needed, and establish a written retention policy where information must be kept. There is no single retention period appropriate for every kind of business record. Applicable obligations need to be assessed for your situation. [FTC: protecting personal information](#).

As a design rule, require an explicit, approved exception when a record must be preserved. Record who approved it, why, and when it will be reviewed. Do not let a deletion job override a preservation requirement merely because a generic age threshold has passed.

Treat backups as governed copies

Do not “solve” retained data by casually destroying your recovery capability. Instead, give backup sets a documented retention window, access policy, encryption approach, and owner. OWASP specifically recommends protecting database backups with appropriate permissions and, ideally, encryption.

[OWASP: database security guidance](#).

A proposed workflow should distinguish live-system deletion from backup expiration. Where selective removal from a backup is not supported, record that limitation, restrict use of the retained copy, and obtain approval for the retention approach. Do not describe a pending expiration as completed destruction.

Build restoration into the design. Suppose Monday’s backup contains a customer who was deleted on Tuesday, and the business restores that backup on Wednesday. Without a reconciliation step, the old record can return to service. That follows directly from restoring the earlier state demonstrated in Case B.

One practical design is a separately protected deletion ledger containing minimal internal identifiers, decision timestamps, affected systems, and approved exceptions. Restore into an isolated environment, reconcile later deletions and preservation decisions, and verify the result before reopening access or restarting outbound communications. The ledger itself needs access controls and a retention rule; it should not become a second customer database.

Replace casual support copies with approved exports

For the fictional scheduling company, the best support workflow might avoid sending its working database at all. Start with synthetic reproduction data or an export restricted to the fields and records needed for the support task. Treat that export as a new asset with an owner, an approved recipient, and an expiration decision.

A project-specific acceptance test should ask whether the export includes only approved live records, whether attachments and indexes are included, and whether the receiving party needs the information. Do not label an export anonymous merely because obvious names have been removed; review what the remaining combinations reveal.

Separate file cleanup from media sanitization

Before retiring storage or handing a device to another organization, apply a media-appropriate sanitization process. NIST SP 800-88 Revision 2 addresses sanitization as making access to target data infeasible for a defined level of effort, with decisions based on information sensitivity. That is a broader objective than obtaining an empty SQL result. [NIST: Guidelines for Media Sanitization](#).

A marker absent from one accessible file says nothing, by itself, about a separately retained storage snapshot, an older device image, or the physical medium. Match the verification evidence to the claim being made: application deletion, file-level cleanup, backup expiration, or media sanitization.

What could happen to your business or infrastructure?

The consequences depend on the information, the recipient's access, and what actually occurred. For the scenario in this article, plausible repercussions include:

- **A larger disclosure than the live system suggests.** An incident involving an old database or backup could include former customers whom your current dashboard no longer lists. An investigation limited to active records could underestimate the affected population.
- **Misuse of confidential business details.** Recovered job notes, pricing, customer correspondence, or site information could reveal commercial relationships or operational details. The harm would depend on the contents; an email address and a building access instruction do not carry the same risk.
- **Recovery work that reintroduces unwanted data.** Restoring a historical backup could reactivate a removed record, republish an attachment, or feed an old customer into an automated workflow unless the restoration process reconciles later decisions.
- **Operational interruption and unplanned work.** The business might need to restrict a support integration, suspend exports, inventory historical copies, rebuild affected artifacts, or involve outside specialists while staff continue serving customers.
- **Privacy, contractual, and trust consequences.** A mismatch between promised deletion and actual handling could require legal review and difficult customer conversations. Notification duties or other legal consequences are not automatic; they depend on the data, circumstances, contracts, and applicable law.

These are scenario-based risks, not a prediction of a fine or an assertion that every residual byte constitutes a reportable breach. The FTC's response guidance emphasizes investigating what happened, involving appropriate professionals, and determining notification obligations. [FTC: data breach response guide](#).

When exposure is suspected, restrict further distribution and preserve relevant evidence under an incident-response plan. Do not start overwriting artifacts simply to make a search come back empty: remediation and evidence preservation need to be coordinated.

Verify the controls with evidence you can explain

A deletion project should finish with more than a configuration screenshot. Define acceptance tests around the data's actual journey:

- **Application behavior:** Use synthetic records to confirm the approved deletion reaches the relevant screens, APIs, related tables, attachments, and outbound workflows—not just one search page.
- **Storage behavior:** Test the actual application's writing connections, ordinary tables, specialized indexes, and coordinated transaction-log handling. Record exactly what was inspected and what remains outside the test's scope.

- **Copy and restore behavior:** Identify the treatment of support exports and retained backups, then demonstrate that a restore cannot return a previously deleted test record to service without the approved reconciliation process.
- **Governance and failure handling:** Verify that exceptions have owners, failed jobs are visible, retries are safe, and the evidence distinguishes completed deletion from pending work or approved retention.

For your project, track the share of data categories with an owner, unreviewed retained copies, deletion tasks past their approved deadline, and successful restore-reconciliation tests. These are proposed operating measures, not industry benchmarks or promises of a particular risk reduction.

Keep the evidence useful without retaining the original sensitive content. A task identifier, system name, timestamps, counts, outcome, and exception reference may establish what happened without copying a private customer note into the audit log.

Turn the findings into a focused Sunimod project

The business opportunity is straightforward: replace uncertain, manual data handling with a repeatable workflow that your team can operate and explain. Start with one valuable process instead of attempting to redesign every system at once.

For the fictional service company, a sensible first scope would cover completed customer jobs: map the database and its copies, define approved retention rules, correct application-side deletion behavior, replace casual support attachments with controlled exports, and test a restore against later deletion decisions.

Hire Sunimod to scope and implement that improvement. Depending on your environment, the project can combine custom application changes, database configuration review, workflow automation, access-control improvements, and documented verification. The deliverables should name the systems covered, the controls implemented, the tests performed, and any remaining limitations.

The value is not a promise that one setting makes information disappear everywhere. It is a system in which your staff can answer: What was removed? What is still retained? Why? Who approved it? What happens if we restore a backup?

Sources and further reading

- [SQLite database file format](#) and [compile-time options](#).
- [SQLite PRAGMA reference](#) and [VACUUM documentation](#).
- [SQLite write-ahead logging](#) and [checkpoint API behavior](#).
- [SQLite backup API](#), [FTS5 documentation](#), and [database corruption hazards](#).
- [Python sqlite3 documentation](#).
- [NIST SP 800-88 Revision 2: Guidelines for Media Sanitization](#).
- [OWASP Database Security Cheat Sheet](#) and [Cryptographic Storage Cheat Sheet](#).
- [FTC: Protecting Personal Information](#) and [Data Breach Response](#).

Build a deletion workflow you can verify

Does your business depend on customer databases, internal tools, support exports, or backup restores? Bring Sunimod the workflow that needs improvement. We can define a practical project to reduce unnecessary retention, improve data handling, and verify what happens when information is removed.

[Request a Sunimod data-protection project quote](#)

Describe the system, the type of information involved, and the outcome you need. Do not submit customer records, database files, passwords, or access tokens through the quote form.