DF-Toolkit: Interacting with Low-Level Database Storage

James Wagner
University of New Orleans
jwagner4@uno.edu

Alexander Rasin
DePaul University
arasin@cdm.depaul.edu

Karen Heart
DePaul University
kheart@depaul.edu

Tanu Malik
DePaul University
tanu@depaul.edu

Jonathan Grier
Grier Forensics
jdgrier@grierforensics.com

ABSTRACT

Applications in several areas, such as privacy, security, and integrity validation, require direct access to database management system (DBMS) storage. However, relational DBMSes are designed for physical data independence, and thus limit internal storage exposure. Consequently, applications either cannot be enabled or access storage with ad-hoc solutions, such as querying the ROWID (which can expose physical record location within DBMS storage but not within OS storage) or using DBMS "page repair" tools that read and write DBMS data pages directly. Such ad-hoc methods are limited in their capabilities and difficult to program, maintain, and port across various DBMSes.

In this demonstration, we showcase DF-Toolkit – a set of tools that provide an abstracted access to the DBMS storage layer. Users will be able to view DBMS storage not accessible through other applications. Examples include unallocated (e.g., deleted) data, index value-pointer pairs, and cached DBMS pages in RAM. Users will also be able to interact with several special-purpose security applications that audit DBMS storage beyond what DBMS vendors support.

PVLDB Reference Format:
James Wagner, Alexander Rasin, Karen Heart, Tanu Malik, and Jonathan Grier. DF-Toolkit: Interacting with Low-Level Database Storage. *PVLDB*, 13(12): 2845-2848, 2020.
DOI: https://doi.org/10.14778/3415478.3415490

1. INTRODUCTION

Relational DBMSes adhere to the principle of physical data independence – DBMSes expose a logical schema of the data while hiding its physical representation. A logical schema of the DBMS consists only of a set of relations (i.e., the data). A physical view of the DBMS, however, consists of several objects, such as pages, records, directory headers, etc. Hiding physical representation is fundamental to the design of relational DBMSes: DBMSes transparently control physical data layout and manage auxiliary objects in order to provide efficient query execution. This data independence, however, impedes several security and performance related applications requiring low-level storage access. Consider the following example, Example 1.

Example 1. A bank or a hospital manages sensitive customer data using a commercial database but for audit purposes must sanitize deleted customer data to ensure that it *cannot* be recovered and stolen. Very few DBMSes offer support for explicit sanitization of deleted data (e.g., `secure delete` in SQLite provides no guarantees or feedback to the user)[1]. In order to programmatically verify that deleted data cannot be reconstructed, a DBA must inspect *all* storage ever used by a DBMS where such data may still reside. This includes DBMS auxiliary objects such as indexes, unallocated fragments in DBMS storage, as well as any DBMS storage released to the OS.

Enabling comprehensive storage-level access is an inherent DBMS challenge because of the way DBMSes control storage. DBMSes control *allocated* storage objects such as a) physical byte representation of the relations, b) metadata that annotates physical storage of relation data, and c) auxiliary objects associated with relations (e.g., indexes, materialized views). The allocated objects are also the ones that the database user can manipulate using the SQL language. However, as illustrated in Example 1, the DBA often also needs access to *unallocated* storage not tracked by a DBMS such as deleted data that lingers in DBMS-controlled files, and DBMS-formatted pages that are released back to the OS and no longer under DBMS control (e.g., files deleted by the DBMS, OS paging files). These objects are certainly part of the physical view and required for a comprehensive storage access, but currently not exposed by any DBMS.

Some DBMSes support built-in tools and interfaces to provide physical storage information at different granularities, but none provide a complete or a standardized view of the storage. The ROWID pseudo-column represents the physical location of a record within allocated DBMS storage, and is one of the simplest examples of storage-based metadata users can access in almost all RDBMSes. Most commercial DBMSes offer custom utilities to inspect and fix page level corruption. Examples include Oracle's `DBMS_REPAIR` [3] (page repair tool allowing users to manually fix or skip corrupt blocks in Oracle storage), Oracle's `BBED` (page editing tool available from Oracle 7 to Oracle 10g), and SQL Server's

[1]DBMS encryption is similar in not providing any feedback. Furthermore, encrypted values should still be destroyed on deletion.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415490

DBCC CHECKDB. However, even for accessible metadata such as ROWID, built-in tools do not help interpret its meaning; a DBA must manually make such interpretations. Moreover, no RDBMS offers access to unallocated storage. Finally, existing tools only support analysis of persistent storage only, not volatile storage. DF-Toolkit is designed to offer a universal standardized access to storage of many DBMSes including (but not limited to) IBM DB2, Microsoft SQL Server, Oracle, MySQL, PostgreSQL, SQLite, Firebird, and Apache Derby. It supports querying and analytics of both persistent and volatile storage including user data, DBMS metadata, DBMS-controlled unallocated storage (e.g., deleted records), OS-controlled unallocated storage (e.g., free listed pages or deleted files), and DBMS memory areas (e.g., buffer cache).

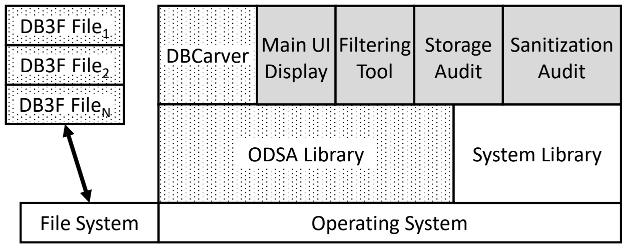


Figure 1: DF-Toolkit Architecture.

2. DF-TOOLKIT OVERVIEW

Figure 1 shows the overall architecture of DF-Toolkit. We divide the components into two categories: 1) storage carving & storage abstraction components and 2) end-user applications. The storage carving & abstraction components are represented by a dotted background. For the purposes of this demonstration the user does not interact with these components. The end-user applications, which are the focus of this demonstration, are represented by a solid gray background.

2.1 Storage Carving & Abstraction Components

DB3F Output. DB3F is a file format for representing DBMS storage artifacts, previously proposed in [7] by Wagner et al. One of the goals of DB3F representation is to abstract DBMS storage engine specifics. For example, the *object identifier* concept exists in every RDBMS. An object identifier is unique for each object in a DBMS; it further maps to a system table for the object's plaintext name (e.g., *Employee* table). Most DBMSes store the object identifier in the page header. Alternatively, PostgreSQL stores the object identifier with each individual record (even though it is redundant, as a database page can only contain data belonging to one object). Although it is represented and stored differently, the function of the object identifier remains the same. DB3F representation abstracts the specifics of how such metadata is stored. In this demonstration, we provide users with pre-generated DB3F files (which were generated by DBCarver).

DBCarver. The term *carving* refers to interpreting data at the byte-level, e.g., reconstructing deleted files without the help of the file system. We previously extended the idea of carving to interpret DBMS storage with DBCarver [9], retrieving both allocated and unallocated data and metadata without relying on the DBMS. DBCarver reads individual files or disk/RAM snapshots and extracts data, including user data and system metadata; it then writes the data to a DB3F formatted file.

ODSA API. We previously proposed a development API to access DB3F files called Open Database Storage Access (ODSA) [10]. ODSA supports both Python and SQL interfaces. For the Python interface, library functions were defined based on general concepts and terminology used across DBMS documentation. For the SQL interface, ODSA loads the DB3F file into a user-specified DBMS, allowing queries to be executed against the carved data. This process, therefore, eliminates the use of the original DBMS and the restrictions imposed by its interface. For purposes of this demonstration, we loaded DB3F files into an SQLite DBMS. All of the DF-Toolkit end-user applications in Section 2.2 were implemented using ODSA; however, the end user does not need to know ODSA.

2.2 End-User Applications

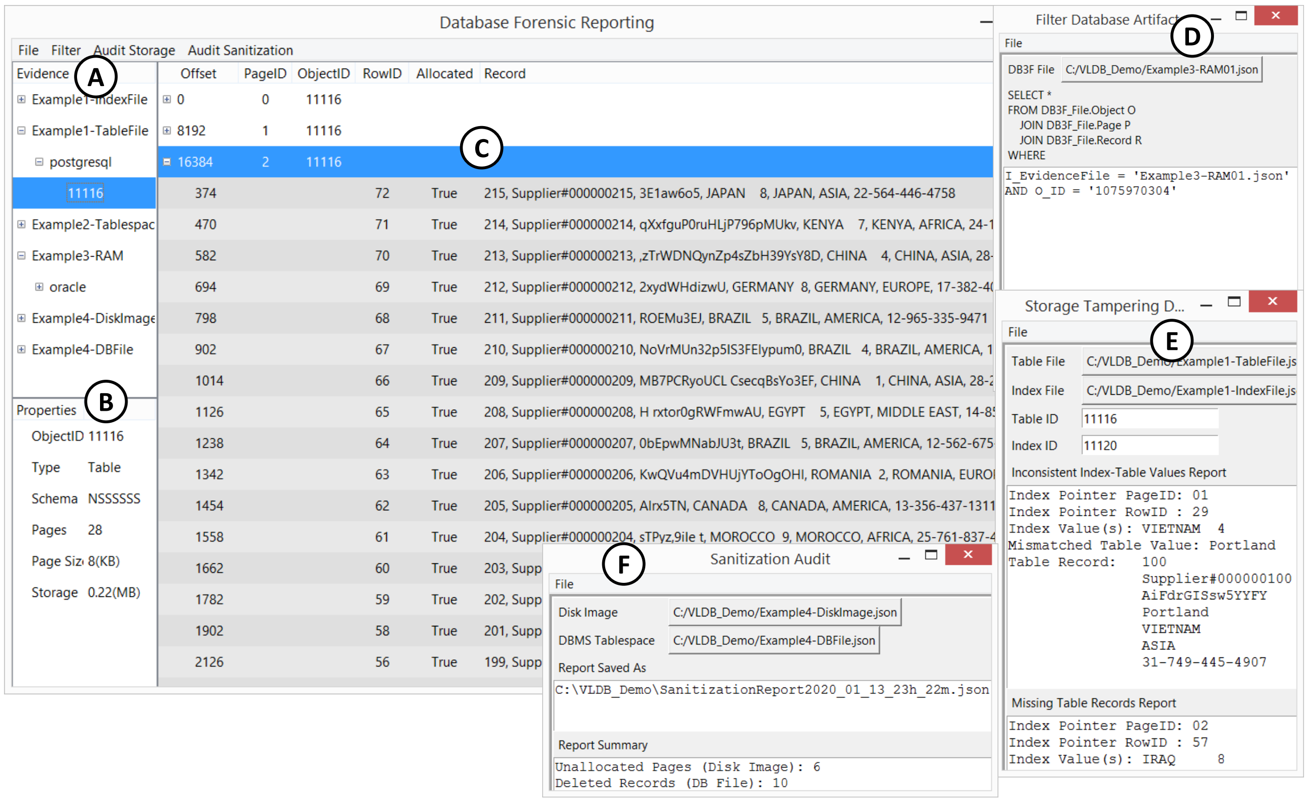
Main UI. Figure 2 shows a screenshot of the main UI screen for DF-Toolkit. This window contains three major panels: 1) evidence tree, 2) current object properties, and 3) the page display panel.

Figure 2 Ⓐ displays the evidence tree. In this example, there are six evidence files, e.g., Example1-TableFile and Example3-RAM; their contents are explained in Section 3. Within each evidence file, data is partitioned by the DBMS vendor since it is possible for more than one DBMS to be present on a disk image or in a RAM snapshot. Example1-TableFile contains PostgreSQL data and Example3-RAM contains Oracle data. For each DBMS, data is further partitioned by object ID (the object name is stored separately in the system tables). The PostgreSQL data contains the object with an ID 11116.

Figure 2 Ⓑ displays properties for the currently selected object. This information includes the object type (e.g., table or index), object schema (in this example, 'N' corresponds to a number and 'S' corresponds to a string), the number of pages associated with the object, the page size, and the total amount of storage occupied by the object.

Figure 2 Ⓒ displays the data for the selected object (11116 in this example). The page display panel displays the list of pages, including the byte offset within the evidence file and the DBMS page ID. The page at offset 16384 (page ID 2) is selected to display the records within the page. The following is displayed for each record: the byte offset within the page, the DBMS internal row ID, the 'allocated' status (True for active and False for deleted), and user data values.

SQL Filtering. Figure 2 Ⓓ displays the filtering application that can be selected from the main UI menu. This application allows querying the DB3F evidence files using SQL conditions. All queries are executed using ODSA (in SQL or Python) against the evidence file; queries are not executed against the original DBMS from which the data was carved. After selecting the DB3F evidence file, the user can enter custom filtering conditions. For example, the Example3-RAM file is a snapshot of RAM containing an Oracle buffer cache. The user may want to see what data was cached following the execution of a specific query. However, the RAM snapshot contains a significant amount of other system table data as well as data cached by other queries. The condition `AND O_ID = '1075970304'` filters the file, retrieving only the records from the table of interest in


Database Forensic Reporting
File Filter Audit Storage Audit Sanitization
A
B
C
D
E
F

Evidence
- Example1-IndexFile
- Example1-TableFile
  - postgresql
    - 11116
- Example2-Tablespac
- Example3-RAM
  - oracle
- Example4-DiskImage
- Example4-DBFile

Properties

ObjectID	11116
Type	Table
Schema	NSSSSSS
Pages	28
Page Siz	8(KB)
Storage	0.22(MB)

Offset	PageID	ObjectID	RowID	Allocated	Record
0	0	11116			
8192	1	11116			
16384	2	11116			
374			72	True	215, Supplier#000000215, 3E1aw6o5, JAPAN 8, JAPAN, ASIA, 22-564-446-4758
470			71	True	214, Supplier#000000214, qXxfguP0ruHLjP796pMUkv, KENYA 7, KENYA, AFRICA, 24-1
582			70	True	213, Supplier#000000213, ,zTrWDNQynZp4sZbH39YsY8D, CHINA 4, CHINA, ASIA, 28-
694			69	True	212, Supplier#000000212, 2xydWHdizwU, GERMANY 8, GERMANY, EUROPE, 17-382-4(
798			68	True	211, Supplier#000000211, ROEMu3EJ, BRAZIL 5, BRAZIL, AMERICA, 12-965-335-9471
902			67	True	210, Supplier#000000210, NoVrMUn32p5IS3FElypum0, BRAZIL 4, BRAZIL, AMERICA, 1
1014			66	True	209, Supplier#000000209, MB7PCRyoUCL CsecqBsYo3EF, CHINA 1, CHINA, ASIA, 28-2
1126			65	True	208, Supplier#000000208, H rxtor0gRWFmwAU, EGYPT 5, EGYPT, MIDDLE EAST, 14-85
1238			64	True	207, Supplier#000000207, 0bEpwMNabJU3t, BRAZIL 5, BRAZIL, AMERICA, 12-562-675
1342			63	True	206, Supplier#000000206, KwQVu4mDVHUjYToOgOHI, ROMANIA 2, ROMANIA, EUROI
1454			62	True	205, Supplier#000000205, Alrx5TN, CANADA 8, CANADA, AMERICA, 13-356-437-1311
1558			61	True	204, Supplier#000000204, sTPyz,9ile t, MOROCCO 9, MOROCCO, AFRICA, 25-761-837-4
1662			60	True	203, Supp
1782			59	True	202, Supp
1902			58	True	201, Supp
2126			56	True	199, Supp

Filter Database Artifact

File

DB3F File C:/VLDB_Demo/Example3-RAM01.json

SELECT *
FROM DB3F_File.Object O
  JOIN DB3F_File.Page P
  JOIN DB3F_File.Record R
WHERE

```
I_EvidenceFile = 'Example3-RAM01.json'
AND O_ID = '1075970304'
```

Storage Tampering D...

File

Table File	C:/VLDB_Demo/Example1-TableFile.js
Index File	C:/VLDB_Demo/Example1-IndexFile.js
Table ID	11116
Index ID	11120

Inconsistent Index-Table Values Report

```
Index Pointer PageID: 01
Index Pointer RowID : 29
Index Value(s): VIETNAM  4
Mismatched Table Value: Portland
Table Record:   100
                Supplier#000000100
                AiFdrGISsw5YYFY
                Portland
                VIETNAM
                ASIA
                31-749-445-4907
```

Missing Table Records Report

```
Index Pointer PageID: 02
Index Pointer RowID : 57
Index Value(s): IRAQ     8
```

Sanitization Audit

File

Disk Image	C:/VLDB_Demo/Example4-DiskImage.json
DBMS Tablespace	C:/VLDB_Demo/Example4-DBFile.json

Report Saved As

```
C:\VLDB_Demo\SanitizationReport2020_01_13_23h_22m.json
```

Report Summary

```
Unallocated Pages (Disk Image): 6
Deleted Records (DB File): 10
```

Figure 2: The main DF-Toolkit UI (A, B, C) and the other end-user applications (D, E, F).

this snapshot. Currently, the newly created filtered file is automatically added to the evidence tree in Ⓐ.

Storage Auditing. Figure 2 Ⓔ displays the storage auditing application that can be selected from the main UI menu. This application is an implementation of our algorithms presented in [8]. We briefly summarize the motivation of our work in [8]. System administrators (and attackers that gained privileged access) have the ability to modify DBMS files at the file system level. Such actions bypass all DBMS access control, constraints, and logging (and could further be hidden by disabling `tune2fs` file system journaling). Since such an attack is performed manually and not by the DBMS, it creates inconsistencies within storage. In window Ⓔ, two different reports are shown indicating storage inconsistencies between a table and the index on the *city* column. The first report shows mismatches between index values and table records. The index value 'VIETNAM 4' has a pointer with page ID 01 and row ID 29. However, the table record at that location contains a 'Portland' city listing instead. Therefore, this is possible evidence that an attacker changed the city value at the byte level (to confirm the inconsistency, our algorithm also checks whether it is a legitimate result of an `UPDATE` operation). The second report shows an index pointer that does not match a valid record in storage. This is abnormal because storage is never zeroed out by the DBMS; deletes and updates can only mark the record as deleted or overwrite it. Therefore, a zeroed out record represents possible evidence that the system administrator replaced each of the record bytes with `NULL`. The algorithm also checks whether the index structure itself was tampered with by the attacker.

Sanitization Audit. Figure 2 Ⓕ displays the sanitization audit application. To understand the goal of this application consider the following. Organizations possessing sensitive data are concerned with data theft and compliance with government regulations (e.g., EU General Data Protection Regulation). This includes data that was deleted, but not destroyed. Data sanitization tools are designed to destroy data so that it can no longer be read at the byte level. Currently, no data sanitization standards exist for DBMSes beyond complete file storage destruction [1]. DBMS sanitization approaches were proposed for MySQL [5, 2]; SQLite currently supports secure delete [4] to destroy deleted data. However, none of the sanitization tools can identify or destroy versions of deleted records stored in backup. Therefore, organizations need tests and guarantees that their sanitization approach complies with organizational requirements and government regulations. DF-Toolkit offers a solution to identify deleted data in any storage medium. When data that should have been destroyed is found, sanitization tool developers must know details about the data to remedy the problem. Example details include the database object (e.g., table of B-Tree index) and the physical address of the data (e.g., byte position within the database file, an OS paging file, or the specific disk sector). To use this application the user provides as input a disk image and the DBMS file(s). The application reports 1) pages found on the disk image outside of the DBMS files and thus outside of DBMS control, and 2) deleted records within the DBMS files. Similar to the filtering application output, the DB3F report file is automatically added to the evidence tree Ⓐ of the DF-Toolkit UI.

3. DEMONSTRATION

For this demonstration, we will present the DF-Toolkit end-user applications (gray background in Figure 1). To supply sample carved data, we loaded Star Schema Benchmark data into several DBMSes and captured and carved data at various storage levels. We provide the following DB3F files for the user to interact with:

- **Example1-TableFile**: a carved PostgreSQL file containing the *Supplier* table.
- **Example1-IndexFile**: a carved PostgreSQL file containing an index on *Supplier*(*S_City*).
- **Example2-Tablespace**: a carved MySQL file containing both the *Supplier* table and an index on *Supplier*(*S_City*).
- **Example3-RAM**: a carved RAM snapshot containing an Oracle buffer cache. The following four queries were executed prior to capturing the snapshot:
  - `SELECT * FROM Customer; --full table scan`
  - `SELECT * FROM Supplier; --full table scan`
  - `SELECT * FROM Supplier WHERE S_Suppkey = 1000;`
  - `SELECT * FROM LINEORDER; --full table scan`
- **Example4-DBFile**: a carved SQLite file with secure delete enabled between record updates.
- **Example4-DiskImage**: a carved disk image which includes the SQLite file from Example4-DBFile.

Main UI. In the main UI window, the user can traverse the provided DB3F files. One example we highlight is the *Supplier* table and *S_City* index data for PostgreSQL in Example1-TableFile and Example1-IndexFile and for MySQL in Example2-Tablespace. Even though PostgreSQL uses heap tables and MySQL uses index-organized tables, the table data and the index data is abstracted and represented similarly by DF-Toolkit. Furthermore, note that indexes cannot be queried through DBMS APIs, but index pages can be viewed and queried in DF-Toolkit via ODSA API.

Evidence Filtering. In this demonstration, the user can perform filtering on any of the provided DB3F files. We provide three sample questions and corresponding queries:

1. Using Example1-TableFile, find supplier records from the nation Germany. This is achieved with the following condition: `WHERE R_Values LIKE '%GERMANY%'`.
2. Using Example3-RAM, find the data cached by the `SELECT * FROM LINEORDER;` query. This is achieved with the following condition: `AND O_ID = '1075970304'`. We note that this query would return all copies of this data in RAM. The work in [6] can be used to assign context to each copy of the data.
3. Using Example4-DBFile, find all of the deleted records, This is achieved by using the following condition: `AND R_Allocated = 'False'`

Storage-Level Audit. In Example1-TableFile and Example1-IndexFile we created a scenario where a system administrator tampered with the PostgreSQL DBMS files. She performed two malicious operations: 1) changed the city for supplier #100 from 'VIETNAM 4' to 'Portland' and 2) replaced all of the record bytes for (200, Supplier#000000200, fdAQLE5VY6hwvxG, IRAQ 8, IRAQ, MIDDLE EAST, 21-472-302-4189) with `NULL`s. In this scenario, the user can detect DBMS file tampering using the storage auditing application.

Sanitization Audit. In Example4-DBFile and Example4-DiskImage, we present the user with a scenario where secure delete was enabled for a SQLite DBMS between a set of SQL operations. By using the sanitization audit application, the user can observe in Example4-DBFile that secure delete does not retroactively destroy records deleted prior to the enabling of secure delete. Similarly, the user can observe that secure delete does not always destroy pages released back to the OS (e.g., modified pages that are written to a new disk sector).

Acknowledgments

This work was partially funded by the US National Science Foundation Grant CNF-1656268. We would also like to thank Kevin Lyon and Shannon Lynott for helping record the video.

4. REFERENCES

[1] Intl. Data Sanitization Consortium. Data sanitization terminology. https://www.datasanitization.org/data-sanitization-terminology/, 2019.

[2] G. Miklau, B. N. Levine, and P. Stahlberg. Securing history: Privacy and accountability in database systems. In *CIDR*, pages 387–396. Citeseer, 2007.

[3] Oracle. Using DBMS_REPAIR. https://docs.oracle.com/cd/B19306_01/server.102/b14231/repair.htm, 2019.

[4] SQLite. PRAGMA statements. https://www.sqlite.org/pragma.html#pragma_secure_delete, 2019.

[5] P. Stahlberg, G. Miklau, and B. N. Levine. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 91–102. ACM, 2007.

[6] J. Wagner and A. Rasin. A framework to reverse engineer database memory by abstracting memory areas. In *International Conference on Database and Expert Systems Applications*, page to appear. Springer, 2020.

[7] J. Wagner, A. Rasin, K. Heart, R. Jacob, and J. Grier. Db3f & df-toolkit: The database forensic file format and the database forensic toolkit. *Digital Investigation*, 29:S42–S50, 2019.

[8] J. Wagner, A. Rasin, K. Heart, T. Malik, J. Furst, and J. Grier. Detecting database file tampering through page carving. In *21st International Conference on Extending Database Technology*, 2018.

[9] J. Wagner, A. Rasin, T. Malik, K. Heart, H. Jehle, and J. Grier. Database forensic analysis with dbcarver. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research*, 2017.

[10] J. Wagner, A. Rasin, D. H. T. That, T. Malik, and J. Grier. Programmable access to relational database storage. In *23rd International Conference on Extending Database Technology*, 2020.