Digital Investigation 18 (2016) S97–S107

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS USA 2016 — Proceedings of the 16th Annual USA Digital Forensics Research Conference

Database image content explorer: Carving data that does not officially exist

James Wagner [a, *], Alexander Rasin [a], Jonathan Grier [b]

[a] DePaul University, Chicago, IL, USA
[b] Grier Forensics, USA

Keywords:
Database forensics
File carving
Data recovery
Memory analysis
Stochastic analysis

ABSTRACT

When a file is deleted, the storage it occupies is de-allocated but the contents of the file are not erased. An extensive selection of file carving tools and techniques is available to forensic analysts – and yet existing file carving techniques cannot recover database storage because all database storage engines use proprietary and unique storage format. Database systems are widely used to store and process data – both on a large scale (e.g., enterprise networks) and for personal use (e.g., SQLite in mobile devices or Firefox). For some databases, users can purchase specialized recovery tools capable of discovering valid rows in storage and yet there are no tools that can recover deleted rows or make analysts aware of the “unseen” database content.

Deletion is just one of the many operations that create de-allocated data in database storage. We use our Database Image Content Explorer tool, based on a universal database storage model, to recover a variety of phantom data: a) data that was actually deleted by a user, b) data that is marked as deleted, but was never explicitly deleted by any user and c) data that is not marked as deleted and had been de-allocated without anyone's knowledge. Data persists in active database tables, in memory, in auxiliary structures or in discarded pages. Strikingly, our tool can even recover data from inserts that were canceled, and thus never officially existed in a data table, which may be of immeasurable value to investigation of financial crimes. In this paper, we describe many recoverable database storage artifacts, investigate survival of data and empirically demonstrate across different databases what our universal, multi-database tool can recover.

Introduction

Deleted files can be restored from disk, even if the storage is corrupted. File carving techniques look for data patterns representative of particular file type and can effectively restore a destroyed file. For numerous reasons (e.g., recovery, query optimization), databases hold a lot of the valuable data and yet standard file carving techniques do not apply to database files. Work by Wagner et al. (2015) described that, primarily due to the unique storage assumptions, database carving solutions must take an entirely new approach compared to traditional solutions.

Our motivating philosophy is that a comprehensive analytic tool should recover *everything* from *all* databases. Beyond simple recovery, forensic analysts will benefit from seeing the “hidden” content, including artifacts whose existence is a mystery. In this paper, we deconstruct database storage and present techniques for recovering database content that does not officially exist. We use our Database Image Content Explorer (DICE) tool to restore deleted and de-allocated data across a variety of different Database Management Systems (DBMSes).

[*] Corresponding author.
E-mail addresses: jwagne32@mail.depaul.edu (J. Wagner), arasin@cdm.depaul.edu (A. Rasin), jdgrier@grierforensics.com (J. Grier).

http://dx.doi.org/10.1016/j.diin.2016.04.015

Our contributions

We present forensic analysis and recovery techniques tailored to de-allocated database storage. We define storage strategies of many relational databases, with in-depth analysis of what happens "under the hood" of a database:

- We define similarities and differences in how different databases handle deletion, explaining why deleted values often remain recoverable for a long duration of time.
- We also show how non-delete user actions create deleted values in a database.
- We explain why databases create and keep many additional copies of the data. Copies that are often created without user's knowledge and sometimes without any human action at all.
- We demonstrate how to recover a surprising amount of content from auxiliary structures used in databases.
- We prove the value of our tool, recovering nonexistent data (de-allocated and/or surviving past expectations) by testing DICE against many DBMSes.

This paper is structured as follows: Section *Background* starts with a review of different database storage elements and the side-effects of caching, Section *The life cycle of a row* deals with row recovery, Section *The life cycle of a page* addresses entire "lost" pages and Section *The life cycle of a value* traces different places where table columns can hide.

A thorough evaluation with different DBMSes in Section *Experiments* shows what can be recovered. Our tests show that DICE can recover 40–100% of deleted rows, 14% of rows overwritten by updates, "invisible" column values in auxiliary structures, and 100% of canceled inserts. We believe that the power of seeing forgotten or non-existent data and transactions, and its potential role in investigation of data-centric crimes, such as embezzlement and fraud, is self-evident. For example, suppose that company X is suspected of falsifying financial information in preparation for an audit. Investigator Y is would want to determine if some financial transactions in their Oracle database have been deleted and, if so, restore the evidence of falsification. Section *Related work* summarizes related work in the area and Section *Conclusions and future work* points towards some of the promising future directions.

Background

Page structure

Relational database pages share the same component structure: the header, the row directory and the row data. Other database-specific components also exist, e.g., PostgreSQL pages have a "special space" for index access. The page header stores variables such as *unique page identifier* or *structure identifier* – this component is always located at the beginning of the page. The row directory tracks individual row addresses within the page, maintained when rows are modified. The row directory is positioned either between the page header and the row data or at the end of the page. Row data structure contains the actual page content along with some additional overhead. Fig. 1 shows how these structures interact within a page.

Some of the parameters used in this paper are summarized in Table 1 for the six DBMSes used in Section *Experiments*. DICE supports many more databases – we only use six different databases due to space considerations. All databases use a unique page identifier, which distinguishes page types. All databases except PostgreSQL store a *structure identifier* in the header, e.g., table *supplier* or an *Index*$_{EmpID}$. Structure information can be recovered from database system tables (see Section *System tables*); PostgreSQL and MySQL use a dedicated file for each database structure and thus establish a more direct link between pages and *structure identifier.*

PostgreSQL, SQLite, Oracle, and DB2 add row directory addresses from top to bottom, with row insertion from bottom to top. SQL Server and MySQL instead add row directory addresses from bottom to top, with row data appended from top to bottom. The order of adding newly inserted rows can affect the order in which deleted values are overwritten. PostgreSQL, SQLite, SQL Server, and MySQL create a *row identifier* – an internal column created by the database that is sometimes accessible to users. PostgreSQL, SQLite, Oracle, and SQL Server explicitly store the *column count* for each row, while DB2 and MySQL do not. PostgreSQL, SQLite, Oracle, and MySQL store the size of each string in the row. SQL Server and DB2 instead create a column directory with pointers to each string column. Oracle *percent used* parameter controls page storage utilization – e.g., setting *percent used* to 50% means that once a page is half-full, new inserts will start replacing deleted rows. In other DBMSes, users have less control over deleted data fragmentation in a page. SQL Server and DB2 mitigate fragmentation by using special storage to shuffle rows and accommodate newly inserted rows in a page (*auto row reclamation* in Table 1). A more comprehensive list of parameters and a description of how to reconstruct pages is described by Wagner et al. (2015).

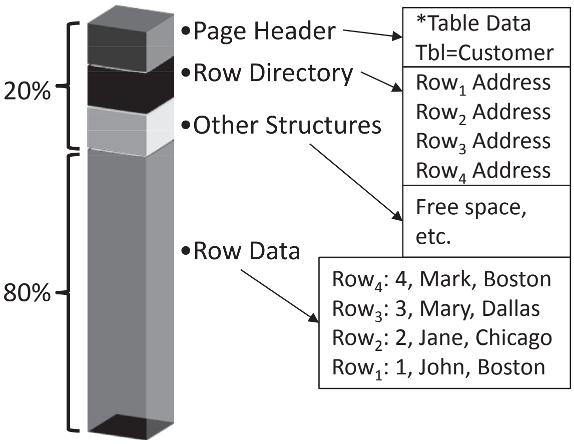

Fig. 1. An overview of database page structure.

Table 1
A list of some of the parameters used to parse a database page or to predict DBMS storage behavior.

Parameter type	Parameter	PostgreSQL	SQLite	Oracle	DB2	SQLServer	MySQL
Header	Unique Identifier	Yes					
	Structure Identifier	No	Yes				
Row Directory	Sequence	Top-to-bottom insertion				Bottom-to-top insertion	
Row Data	Row Delimiter	No		Yes			
	Row Identifier	Yes		No		Yes	
	Column Count	Yes			No	Yes	No
	Raw Data Delimiter	Yes	No				
General	Percent Used	No		Yes	No		
	File-Per-Object Storage	Yes	No				Yes
	Auto Row Reclamation	No			Yes		No

Database storage structures

Table. Each table in a database consists of a collection of pages (as in Fig. 1), with each page holding a subset of table rows. The *database catalog* is stored in *system tables* (similar to a regular table, Section *System tables*). In most databases, tables represent only a part of the database content.

Indexes. An index is an auxiliary structure that helps locate rows within a database table. For example, consider a *customer* table from the SSBM (Neil et al., 2009) benchmark that is sorted by customer ID. A query searching for a particular city (`city = 'Boston'`) would scan the entire table, but an index on *city* expedites the search by storing value-pointer pairs (e.g., [Boston, Row#2], [Boston, Row#17], [Chicago, Row#12]). Indexes are stored in pages that follow the structure in Fig. 1 (in many ways, an index is a 2-column table), but some of the row metadata may be omitted – e.g., indexes do not need to store *number of columns*, since there are exactly two.

Database administrators (DBA) create indexes to tune database performance, but we emphasize that many indexes are created automatically. Some users know that declaring a primary key constraint causes databases to build an index, but few would know that so does the creation of a `UNIQUE` constraint (e.g., *SSN* column).

Materialized views (MVs). MVs are pre-computed queries – unlike views that are "memorized" but not physically stored. For example, if SQL query `SELECT * FROM Customer WHERE City = 'Boston'` is executed often, DBA may choose to construct a `BostonCustomers` MV that pre-computes the answer in order to speed up that query. MVs are not created automatically, but some indirect actions can cause MV to become materialized – e.g., indexing a view in SQL Server makes it a materialized view.

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

Caching behavior in databases

DBMSes maintain a cached set of data pages in RAM (*buffer cache*) to reduce disk access cost during query execution. However, when database users update tables (e.g., inserts or deletes), pages are modified in-place, creating so-called *dirty* (modified from original) pages in memory. The dirty page stored in memory replaces the old page on disk only after it is evicted from cache.

Inserts, deletes and updates present opportunities for recovering old, new or even *tentative* data – transactions can be aborted and thus "undone", but data content will linger in storage. We discuss the recoverable artifacts for a variety of different DBMSes in Sections *The life cycle of a row and auxiliary structure: indexes*.

The following sections define recoverable extraneous data at different levels (rows, pages and values). All of our analysis specifically addresses non-recoverable values, that cannot be directly accessed by the database users or by third-party recovery tools (see Section *Related work*).

The life cycle of a row

Relational database store tables (relations) and therefore the smallest entity that can be deleted or inserted is a row (tuple). An update changes specific columns, but in practice updates will sometimes manipulate an entire row (delete + insert) at the storage layer. In the rest of this section we explain why data-altering operations leave recoverable copies behind and how such data can be restored.

Page structure

Deleted rows can both be recovered and explicitly identified as "deleted" by DICE. In contrast, a discarded page (see Section *The life cycle of a page*) looks like any other page and requires additional steps to identify as "unused". There are three types of deleted row alterations that may be used by a database: 1) the row header is updated, 2) the address in row directory is deleted, 3) the metadata within the row is modified.

Row header. Every database we investigated updates the row header in the affected page. This helps us determine when a page was last altered but not what specific data was updated. For example, if the page header changes compared to previous version, we know that the page was altered at some point in-between – page checksum update is one of the alteration causes.

Row directory. Only two databases, DB2 and SQL Server, change the page row directory when a row is deleted. When a row is deleted in SQL Server and DB2 the row directory address is overwritten with a NULL value. SQL Server overwrites each byte of an address with decimal value 0, and DB2 overwrites each byte of an address with decimal value 255. Deleted rows can be identified and restored by searching for and parsing the row pattern between the preceding and following valid row entries for each NULL address in row directory. SQL Server and DB2 only use the row directory to reflect the specific row that has been deleted, and do not alter row metadata at all.

Row metadata. Oracle, PostgreSQL, SQLite, and MySQL update row metadata to mark deleted rows. We found that some of the same parameters for the row data in a page can also be used to distinguish an active row from a deleted row. We summarize our findings and parameter decimal values in Table 2. MySQL and Oracle mark the row delimiter at the position stored in the row directory address – a deleted row can be identified using Table 2 values. PostgreSQL marks the raw data delimiter, identifying the start of individual values within the row. When a row is deleted in PostgreSQL, the second byte of raw data delimiter is updated to a new value. SQLite marks the *row identifier*, a unique ID created by the database for each row. In SQLite deleted rows all share a common row identifier value allowing us to detect a deleted row.

Fig. 2 contains examples of what a deleted row looks like in different DBMSes. In each example, the Row2 containing (*Customer2, Jane*) has been deleted while Row1 and Row2 containing (*Customer1, Joe*) and (*Customer3, Jim*) are active. The first example page shows how the row delimiter is marked in a database such as MySQL or Oracle, the second example page shows how the raw data delimiter is marked in PostgreSQL, and the third example show how the row identifier is marked in SQLite. Fig. 2 omits DB2 and SQL Server as they only alter the row directory on deletion.

Updated rows

When a row is updated, it can be updated in-place (see Section *The life cycle of a value*) or by a sequence of `DELETE` and `INSERT`. For all of the databases we studied, when a row is updated to a new row of equal or lesser size, old row storage is overwritten with new content (old value remainder can still be recovered). When a row is updated to size greater than the size of the old row, the old row is marked as deleted (same as regular delete) and the new row is either appended to the end of the table, or overwrites other deleted rows if an empty slot is available (subject to exact DBMS policy). Since the row is marked exactly like a regular delete, a deleted row caused by an `UPDATE` operation and a deleted row caused by a `DELETE` operation can't be distinguished.

Table 2
Row data parsing parameters used to identify deleted rows.

DBMS	Parameter	Active	Deleted
MySQL	Row Delimiter	0	32
Oracle	Row Delimiter	44	60
PostgreSQL	Data Delimiter	2, 9, 24	2, x, 24
SQLite	Row Identifier	4 uniq bytes	0,226,0,57

*This table excludes DB2 and SQL Server because these DBMSes mark deletion in row directory but not in metadata.

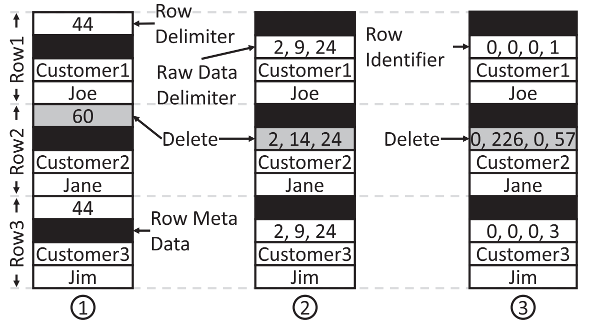

Fig. 2. Examples of how deleted rows are marked in different databases: 1) MySQL or Oracle 2) PostgreSQL 3) SQLite.

Transactional effects

A transaction can fail because it conflicted with another running transaction or because it was canceled. Failed transactions are undone from user perspective but the page storage is still altered in the database: 1) inserted row can still be recovered from page storage (marked as deleted), 2) an old copy of the updated can be recovered from page storage (looking similar to a deleted row) and 3) a deleted row is reinserted to cancel out deletion. Thus, *every possible canceled operation will leave recoverable rows in storage* – database logs could determine whether the “deleted” row is actually a side-effect of `INSERT` or `UPDATE`.

The life cycle of a page

Data pages

In this section we discuss causes for de-allocation of multiple pages. When a user drops a table, all pages become unallocated – such pages are fully recoverable until overwritten. Table deletion is only one of the operations that de-allocate data pages. A more interesting example is *structure reorganization* that compacts table storage (fragmented by deletion and other operations from Section *The life cycle of a row*).

Few databases (DB2 and PostgreSQL) permit explicit reorganization of table storage. Oracle and SQLite require that a new table be built to compact an existing table. Both DB2 and SQL Server reclaim deleted tuple space with new row inserts (*auto row reclamation* in Table 1). However, SQL Server may require a *cleantable* command to reclaim space wasted from a dropped column. MySQL uses OPTIMIZE

TABLE command, which is very similar to the rebuild operation expected by Oracle and SQLite.

A DBMS may choose to perform a compacting operation automatically – DB2 even provides control over automatic reorganization (DB2). Rebuild operation (with or without user's knowledge) will typically leave behind recoverable table pages just as the `DROP TABLE` command. Recovering a discarded page is trivial for DICE (discarded and active pages are usually identical), but to identify whether a page is discarded we need to look at system tables.

System tables

A deleted table page is not usually identified as deleted in storage, unlike deleted rows which are explicitly marked. In order to identify de-allocated (i.e., old) recovered pages, we reconstruct the system table that stores table name and structure (or object) identifier. *Structure identifier* is one of the page parameters stored in the page header. System tables are typically stored in regular pages on disk, but require additional parsing parameters and use different existing parameter values for parsing. System tables may also contain unique data types that are not accessible to the user. Since determining the structure of system tables and new datatypes with synthetic data may not be feasible, manual analysis was typically performed to create new parameters or parameter values.

In Oracle, system table page is similar to regular data page and uses standard data types. When a table is dropped, data pages are not altered, but the corresponding system table row is marked as a regular deleted row. SQLite system tables contain extra metadata within the row, but still use standard data types. When a table is dropped, metadata in the row data is marked and the row header of the *first page that belongs to the table* is marked.

PostgreSQL system table pages use regular structures, but *raw data delimiter* (used to locate raw data, see Section *Page structure*) uses a different value. PostgreSQL system tables also use several data types not available to the user. Some of these data types are listed in Table 3. Object Identifier (*OID*) is an object (or structure) identifier, and stored like any other 4-byte number in PostgreSQL. The *Name* data type stores object name string in a special reserved 64 byte slot. *Aclitem* is an array that stores user access privileges. *XID* is a transaction identifier, also stored like a 4-byte number in PostgreSQL. When a table is dropped, the corresponding row in the system table is overwritten. The single, dedicated data file for the table still exists, but all references to database pages are removed and file size is reduced to 0. Discarded pages from the dropped table can still be recovered from unallocated file system storage.

Table 3
MV refresh options for each database.

Datatype	Size	Description
OID	4 bytes	Identifier to represent objects.
Name	64 bytes	Name of objects.
Aclitem	Variable	An array of access privileges.
XID	4 bytes	Transaction identifier.

MySQL stores database catalog tables in RAM, and no system table pages were found on disk. This is a direct consequence of MySQL implementation – in addition to the newer InnoDB (InnoDB Engine), MySQL still uses an older MyISAM (MyISAM Storage Engine) storage layer, which does not use pages (to our knowledge MySQL is the only row-store database to do so and MyISAM is being retired). DICE was built to parse pages across different databases, and thus special-case parsing is required for parts of MySQL stored in MyISAM. When a MySQL table is dropped, the files containing table data and metadata are deleted in file system. In DB2 there were no notable differences between system table pages and data pages, nor did we observe special data types in DB2 system tables. When a DB2 table is deleted, data pages are not changed but the corresponding system table row is deleted (using same deletion mark as rows).

SQL Server was the only database to successfully hide its system tables (so far). According to SQL Server documentation, system tables are not accessible to users but only to developers. We were not able to find where system tables are stored, but we have ascertained that they are not in the default instance storage. Table data pages in SQL Server are not altered in any way when a table is dropped (and thus can be recovered by DICE).

Fig. 3 shows an example of how pages belonging to a deleted table can be detected for databases such as Oracle, PostgreSQL or SQLite that update system tables when a table is dropped. In this figure, the row for the deleted table is marked in the system table as previously described for each database in Section *The life cycle of a row*. Table *supplier* has been dropped while table *customer* remains active. The pages for *customer* and *supplier* table use structure identifiers 125 and 126. In order to determine if these pages are deleted or active, we check the relevant page of the catalog system table. This system page shows the row meta data contains a deleted mark for the row with structure identifier 126. The table catalog page also stores the table name (*supplier*) for this deleted structure. This allows us to identify all parsed pages with the *structure identifier* 126 as discarded pages belonging to the *supplier* table.

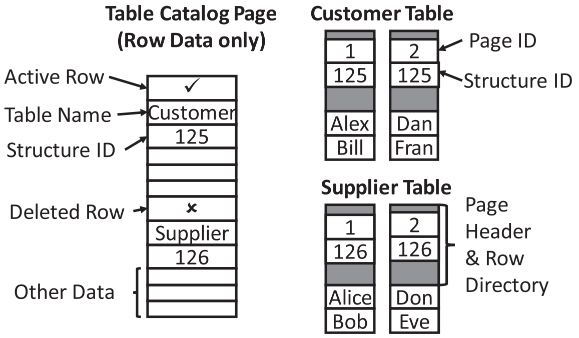

Fig. 3. Example of how a deleted page is marked.

The life cycle of a value

Database tables are stored and processed as a collection of individual rows that comprise them. In Section *The life cycle of a page* we described scenarios where an entire table (or at least a collection of pages) can be discarded by a single operation. We now discuss scenarios that create individual de-allocated values (i.e., columns) in database storage.

Auxiliary structure: indexes

Section *Database storage structures* defines a variety of common non-table structures that contain copies of data. When a row is deleted, DBMS does not delete the corresponding index values – nor are such index values marked deleted. Although indexes were designed to be dynamically maintained (Comer, 1979), it is easier to leave the index entry for a deleted row. For example, if an employee is erased from a table, Index$_{EmployeeID}$ would keep this ID value, relying on row deletion mark to ensure query correctness (i.e., queries *should not* access deleted employee records). This holds true for all table indexes; such not-really-deleted values will exist in indexes for a long time, either until the index is rebuilt or until massive storage changes occur (see Experiment 3).

While deletion does not remove values from the index, inserting a new row does create a value *even if that insert is canceled* (i.e., transaction ABORT). The nature of database transactions (Section *Database storage structures*) means that it is easier to include every possible value in the index and rely on other metadata to determine if the row is relevant to query lookup. Therefore every indexed value, including never-inserted values will find its way into the index. Fig. 4 contains one example: student records Carl and Greg have been deleted (and marked as such), but the ID values for these students (*035* and *143*) still persist in the index.

Row from an aborted insert is treated as if it was inserted and then deleted (transaction logs can differentiate between the two options). An update that has been canceled would also be treated as a combination of an insert and a delete. The pre-update value would be marked as deleted (if the new value is larger and cannot change in-place) and the post-update value of the canceled update will be marked as deleted too.

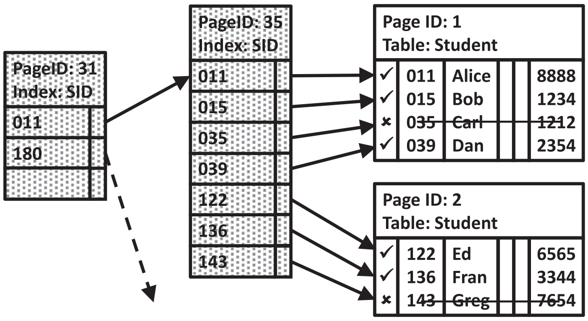

Fig. 4. Example of how a deleted value can still live in an index.

Auxiliary structure: materialized views

The amount of extraneous values in an MV depends on update options configured for this MV (which, in turn, depends by update settings available in a DBMS). There are three types of MV refresh options that databases can offer: 1) use a custom refresh function, 2) refresh on each transaction commit, 3) refresh on demand. Table 4 summarizes which refresh options are available for each database. A custom refresh function can be created using *trigger* mechanism to refresh an MV based on certain events (e.g., UPDATE). Refresh on commit will refresh the MV when a COMMIT is issued, reducing the maintenance overhead somewhat. Refresh on demand refreshes the MV only when manually requested (that is the cheapest option).

We discuss MVs in this section (dedicated to recoverable values) because MVs have fewer columns compared to source tables and can include new pre-computed columns. Even when an entire MV row is affected by data changes, this row is still a subset of columns from the original table. MV maintenance works similarly to table maintenance, both support a rebuild directive.

When a row is deleted from a table, but the MV is not refreshed, all table values stored in MV can still be recovered from disk. Such old MV values may or may not be accessible to the user (depending on database policies). When an MV is refreshed, deleted data may either be overwritten by active data or marked as deleted (similar to table rows). Note that SQLite, MySQL, and PostgreSQL (prior to PostgreSQL 9.3) do not offer materialized views – but since MV-like functionality is desirable, database documentation recommends building a "derived" table instead (CREATE NewTable AS SELECT …). In that case, MV rows follow the same rules discussed in Sections *The life cycle of a row and The life cycle of a page* because this MV is really a table.

Experiments

Our current implementation of DICE forensic tool supports at least *ten different* RDBMS systems under both Windows and Linux OS. We present results using six representative databases (Oracle, SQL Server, PostgreSQL, DB2, MySQL and SQLite) due to space limitations. Other supported DBMSes are less widely used (e.g., Firebird and ApacheDerby); yet others are supported by the virtue of sharing the same storage layer: e.g., MariaDB (same as MySQL) and Greenplum (same as PostgreSQL). Our

Table 4
MV refresh options available in each database.

DBMS	Function	Commit	On demand
DB2	✓	✓	✓
MySQL	✓	–	✓
Oracle	✓	✓	✓
PostgreSQL	✓	–	✓
SQLite	✓	–	✓
SQL Server[a]	–	–	–

[a] Indexed views are immediately refreshed. The user cannot change this setting.

experiments were carried out on servers with Intel X3470 2.93 GHz processor and 8 GB of RAM; Windows servers run Windows Server 2008 R2 Enterprise SP1 and Linux experiments used CentOS 6.5. We either read the database storage files or the raw hard drive image since DICE does not rely on file system structure. Algorithm 1 describes the overall parsing process. A file and a database parameter file are passed as an input. For every *general page identifier* found in the image file, DICE records the page header and the row directory parameters. Next, a list of addresses from the row directory are recorded. For each row directory address, the row data parameters are recorded, and the row is parsed. Page parameters and a list of rows are recorded for each *general page identifier*. Finally, the DICE parser returns the list of all pages.

Table 5 outlines the results for each of the experiments presented here.

Algorithm 1 DICE Parsing Algorithm

```
Input: (Any image file, database parameter file)
for each GeneralPageIdentifier in $imagefile$ do
    set PageHeader and RowDirectory parameters
    for each $ValidAddress$ in RowDirectory do
        append $ValidAddress$ to $Addresses$
    end for
    for each $Address$ in $Addresses$ do
        set RowParameters
        $Row \leftarrow ParseRowData()$
        append $Row$ to $RowList$
    end for
    append ($PageParameters, RowList$) to $PageList$
end for
return $PageList$
```

Experiment 1: Recovering Deleted Rows. In this experiment we demonstrate several properties of deleted row storage behavior: 1) for any database, 100% of deleted rows can be recovered immediately after deletion, 2) over time, we can recover a significant chunk (40%) of deleted rows from most databases and 100% of deleted rows from Oracle, 3) given an atypical workload of deletes specifically designed to be “easy to overwrite”, we can still recover 1% of deleted rows. Our experiments highlight the difference between deletes that result in high and low amount of *deleted row fragmentation*. A sequential range of deleted contiguously-stored rows is more likely to be replaced by new data. Deleted rows that are scattered across pages are less likely to be overwritten.

Table 5
Summary of experimental results in this section.

#1	DICE recovers 40–100% of the deleted rows even after many inserts have been are executed.
#2	For up to 14% of updated rows, the full pre-update version of the row can be recovered.
#3	In addition to #1/#2, old values are recovered from active and deallocated index pages.
#4	Canceled transactions leave just as many recoverable values in storage as regular transactions.
#5	DICE can recover 0.5% deleted rows and duplicate active rows after MV rebuild.
#6	Table rebuild leaves behind 1) 85% deleted rows *or* 2) a large number of duplicate active rows.

We use two databases with different row replacement approach. SQL Server overwrites deleted rows once a row of equal or lesser size is inserted into the table, possibly doing some in-page defragmentation – Oracle will instead wait until page utilization falls below a user-configurable threshold (see Table 1, Oracle default threshold is 39%). For both Oracle and SQL Server, we started with two different tables, each with 20 K random sized rows. Both databases used a page size of 8 KB, and each page contained approximately 85 rows resulting in table sizes of 236 pages. We deleted 1000 rows (more than one per page), inserted 1000 new rows of random size, and inserted another 1000 random rows. At each step we evaluated how many deleted rows are recovered from disk. In table $T1_{rand}$, 1000 deleted rows were randomly distributed across the page storage, while in table $T2_{cont}$ 1000 deleted rows were contiguous (i.e., delete *all* rows from just a few pages).

As Table 6 demonstrates, before new inserts come in, *all* of the deleted rows can be recovered by DICE. None of the deleted rows for $T1_{rand}$ in Oracle were overwritten by inserts executed in the next step. The default threshold in Oracle is 39%, and we only deleted about 5% of the rows in each page, leaving 95% intact. For $T2_{cont}$ in Oracle all but 8 of the deleted rows were overwritten by the first 1000 new inserts (these 8 rows were still recoverable after 1000 more inserts). In $T2_{cont}$ deleted rows correspond to wiping out 19 pages (0% utilization each) – the remaining 8 rows spilled into the 20th page with other active rows with sufficiently high utilization (85%). In SQL Server we saw that in both $T1_{rand}$ and $T2_{cont}$ first 1000 new inserts overwrote 60%–65% of de-allocated rows (due to compaction applied by SQL Server). For the second 1000 inserts, $T2_{cont}$ replaced most of the deleted rows because they are contiguous and easy to overwrite. For $T1_{rand}$, only 20 additional rows were displaced by the second batch of 1000 inserts because remaining $T1_{rand}$ are the smallest surviving rows that are difficult to overwrite.

Fig. 5 shows how an inserted row may overwrite a deleted row in SQL Server. (*Supplier1, Bob*) was initially marked as deleted. On the left side we demonstrate inserting a new record (*Supplier3, Ed*). Since *(Supplier3, Ed)* requires fewer bytes than (*Supplier1, Bob*), the inserted row can overwrite the deleted row. Note that since the inserted row is smaller than the original deleted row, fragment of the old row, i.e., *b*, can be recovered from page storage. On the right of Fig. 5 we inserted the record (*Supplier3, Gregory*). Since (*Supplier3, Gregory*) requires more storage than (*Supplier1, Bob*), there is not enough space to overwrite the deleted row. This forces the inserted row to be appended to

Table 6
Number of deleted rows recovered by DICE.

Action	Oracle		SQL server	
	$T1_{rand}$	$T2_{cont}$	$T1_{rand}$	$T2_{cont}$
Delete 1 K Rows	1000	1000	1000	1000
Insert 1 K Rows	1000	8	416	354
Insert 1 K Rows	1000	8	394	12

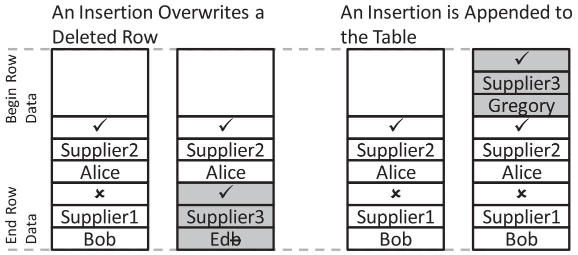

Fig. 5. An example for row insertion behavior in SQL Server.

table page, leaving (*Supplier1, Bob*) intact with a deletion mark.

$T1_{rand}$ is far more representative of day to day database use because it is hard to delete contiguously stored rows, even on purpose. Row storage shifts over time and particular rows are unlikely to be co-located. Only a DBA would know which rows are stored on the same page.

Experiment 2: Recovering Pre-Update Rows. In this experiment we demonstrate that for a typical workload of UPDATEs we can recover many old rows, although fewer (5%–10%) compared to DELETEed row recovery in Experiment 1. Fewer old rows can be recovered because while updating values to a larger value results in DELETE + INSERT, updating row to a smaller value overwrites the old value in-place. We perform this experiment using DB2 (DB2 behaves similarly to SQL Server in that context) and PostgreSQL. For each database, we started with two tables of 20 K randomly sized rows and updated 1000 of the rows to a new random row, followed by another 1000 random updates for a total of 2000. 1000 updates in $T1_{rand}$ were distributed across the table storage at random, while 1000 updates in $T2_{cont}$ updated a contiguously stored sequence of rows. Both DB2 and PostgreSQL compact page contents to keep larger updated value on the same page. However, if there is not enough free space available, the new row is stored in a different page and the old value is marked as deleted in the original page. New updates will overwrite old deleted-by-update rows over time.

As Table 7 shows, for $T1_{rand}$ in DB2, we recovered 121 pre-update records after 1000 updates and 125 records after a total of 2000 updates were executed. Approximately 6%–12% of old records remained recoverable due to the way DB2 manages page storage. For $T2_{cont}$ in DB2, we were only able to recover 6 old records after 1000 updates and 10 old records after all 2000 updates were performed. For $T1_{rand}$ in PostgreSQL, we recovered 137 values after the first 1000 updates and 92 records after the second 1000 updates. For $T2_{cont}$ in PostgreSQL, we recovered a single row, which was the last update in the sequence of 1000 updates. We have observed (as expected) that continuous patch of deleted-by-update rows is overwritten by new data quickly. The numbers in Table 7 *only* include fully recoverable rows, ignoring some *partial* old values that can be recovered as well (e.g., b example in Fig. 5).

Experiment 3: Recovering Indexed Values. This experiment demonstrates that DICE can recover thousands of old deleted or pre-update values (both from active and discarded index pages) from an index structure in a database.

Table 7
Number of updated rows recovered.

Upd. Rows	DB2		PostgreSQL	
	$T1_{rand}$	$T2_{cont}$	$T1_{rand}$	$T2_{cont}$
1000	121	6	137	1
2000	125	10	92	1

We used SQL Server and an index on *region* column for SSBM Scale1 (=30 K rows) *customer* table – in general, indexes behave similarly across all DBMSes.

Region column has 5 distinct values, including ‘AMERICA’, ‘ASIA’, and ‘EUROPE’, with roughly 6 K records for each ($6K*5 = 30K$). Table 8 summarizes recovered value counts – each time a count changes, the cell in Table 8 is highlighted with gray. We note that additional duplicate values were recovered based on the behavior described in Section *Auxiliary structure: indexes*, but we do not include those to avoid double-counting results. We first deleted 1000 rows from *customer* table with *region* value of ‘AMERICA’. This resulted in deallocation of two index pages containing ‘AMERICA’ that we recovered. Next, we updated 1000 rows in *customer* table with the value ‘AMERICA’ to the value ‘CAMELOT’ (not a real value for this benchmark). This action created new ‘CAMELOT’ values and displaced more of the ‘AMERICA’ index pages.

We next deleted all of the rows with value ‘ASIA’, forcing the index to deallocate 20 pages. All of the ‘ASIA’ remained recoverable. We then updated all ‘EUROPE’ rows to ‘ATLANTIS’ in the table. The index only grew by 5 pages, but the number of deallocated pages increased by 18 pages. The number of recoverable ‘AMERICA’ and ‘ASIA’ values decreased after some deallocated pages were overwritten. Finally, we updated all of the remaining 16 K original values in *customer* to a new value not in this benchmark. And yet a significant fraction of ‘AMERICA’, ‘ASIA’, and ‘EUROPE’ values were recovered – either from active or from deallocated pages of the index.

Experiment 4: Aborted Transaction Effect. This experiment proves that data inserted by aborted transactions is fully recoverable by our tool, both from memory and disk, just like regular deleted rows. We were also able to independently recover these never-inserted values from indexes that were attached to the table. We begun the experiment by loading the *supplier* table from SSBM benchmark into Oracle. We then inserted 1000 rows and issued an ABORT command resulting in ROLLBACK. The data from canceled inserts was cached, then marked as deleted and subsequently recovered from pages in memory. Once cache contents were flushed, pages containing rows from the aborted transaction were recovered from disk storage as well. One might intuitively expect that in-memory cache of modified pages would be simply discarded on ABORT – but all 1000 rows were appended at the end of the table on disk. We have also found that the values from canceled inserts were added to the *supplier*'s index.

Experiment 5: Materialized View Refresh. In this experiment we show that: 1) we can recover all of the deleted rows from an MV (*in addition* to recovering these deleted rows from table storage, 2) after MV is refreshed we can

Table 8
Life cycle of deleted and updated values in a SQL Server index.

Action	Index (Pg)	Deallocated (Pg)	America	Asia	Europe	Camelot	Atlantis
Initial	115	0	5992	6051	5937	0	0
Delete 1 K America Rows	113	2	5992	6051	5937	0	0
Update 1 K America to Camelot	116	5	5992	6051	5937	1000	0
Delete All Asia Rows	96	25	5992	6051	5937	1000	0
Update All Europe to Atlantis	101	43	4692	5993	5937	1000	6167
Update Rem. 16 K Rows	135	71	4687	1080	1419	1000	6167

still recover 5% of the deleted rows from the MV, 3) the refresh operation *also* generates extra copies of other, non-deleted rows. We initialized this experiment with two MVs containing 20 K random sized rows and then deleted 1000 rows from the underlying tables. As in previous experiments, for $M1_{rand}$, 1000 deletes are applied to random storage locations in the table and for $M2_{cont}$ table deletes are applies in a contiguous fashion. Table 9 summarizes the number of deleted rows and extra copies of active rows (1100 + is not a typo – and duplicated rows *do not intersect* with 1000 deleted rows) recovered from both MVs.

Before the refresh, we can recover every single one of the 1000 deleted rows from the MV. This is independent of rows recovered from table storage, such as in Experiment 1. After refresh, we found 51 “deleted” rows and 1107 duplicates of the active rows in $M1_{rand}$. The duplicated rows came in two flavors: 1) rows marked as deleted in active pages (but *not* from the 1000 of user-deleted rows) and 2) rows from de-allocated MV pages but not marked as deleted and *also* not from any of the 1000 user-deleted rows. Some rows were available from both sources, but our results only count one recovered copy per row. Less than 10% of the duplicates were discovered in both sources and these were eliminated from our counts. For $M2_{cont}$, we found 60 deleted values and recovered 1111 distinct active rows from de-allocated storage. Similar recovery rates for $M1_{rand}$ and $M2_{cont}$ were as expected, because rows are being deleted *from the original table*, not from the MV that is reconstructed by DICE.

Experiment 6: Table Rebuild. This last experiment demonstrates in PostgreSQL that: 1) a table refresh following typical workloads will leave only 1% + of recoverable deleted rows but more unrelated duplicate row copies, 2) a table refresh that follows a continuous sequence of deletes from that table will generate 85% of recoverable deleted rows and few unrelated duplicate row copies. One way or another table refresh leaves behind recoverable duplicate rows, similar to MV refresh. PostgreSQL is the only database where users have easy access to a manual defragmenting command – in other DBMSes, one typically has to recreate the structure to compact storage. When building a brand new structure, old pages are even more likely to be left behind, so PostgreSQL is chosen as the database likely to leave the *fewest* discarded pages.

We created two tables with 20 K random sized rows and then deleted 1000 rows. 1000 rows deleted in $T1_{rand}$ were distributed across the page storage and 1000 rows deleted in $T2_{cont}$ were stored contiguously. Table 10 shows the number of recovered deleted rows and duplicated active rows. After a refresh of $T1_{rand}$, we recovered 16 deleted rows and 1134 discarded copies of active rows. Similarly to the previous experiment, 16 deleted recovered rows were marked deleted, and 1134 duplicate values were de-allocated without any markings – 16 deleted values were from 1000 deleted rows, but 1134 duplicates are from the other 19,000 rows. For $T2_{cont}$, we instead found 854 deleted rows and 182 duplicates of active rows on disk.

Fig. 6 illustrates why we recover so many duplicates for $T1_{rand}$ but instead recover many more deleted values for $T2_{cont}$. PostgreSQL defragments rows within the page when rebuilding tables, which results in different storage allocation depending on whether deletes were randomly scattered or contiguous before the rebuild. In Fig. 6, for sparsely deleted rows before rebuild, Row2 has been marked as deleted in the row metadata. As the first page in Fig. 6 indicates, PostgreSQL does not alter the row directory for deletion. After the table is rebuilt, Row3 is moved to overwrite the deleted Row2 row, the old record for Row3 is then marked “deleted” on the page, the row directory address for Row3 is updated to reflect the new location, and finally the row directory address for Row2 is set to NULL (this NULL has nothing to do with deletion). The result is a contiguous free space on the page between the row directory and the row data – which also happens to duplicate Row3 in storage, without user's knowledge. For the densely (contiguously) deleted rows in Fig. 6 (3rd page in figure), all rows on the page are marked as deleted. When the table is rebuilt, the row directory addresses for all deleted rows are set to NULL. Because there are no live rows in this page, live rows are not duplicated as in the

Table 9
The number of deleted rows and duplicate active rows recovered from disk storage after MV refresh in Oracle.

Row type	Before refresh		After refresh	
	$M1_{rand}$	$M2_{cont}$	$M1_{rand}$	$M2_{cont}$
Deleted	1000	1000	51	60
Duplicated	0	0	1107	1111

Table 10
The number of deleted rows and duplicate active rows recovered after a table rebuild in PostgreSQL.

Row type	Before		After	
	$T1_{rand}$	$T2_{cont}$	$T1_{rand}$	$T2_{cont}$
Deleted	1000	1000	16	854
Duplicated	0	0	1134	182

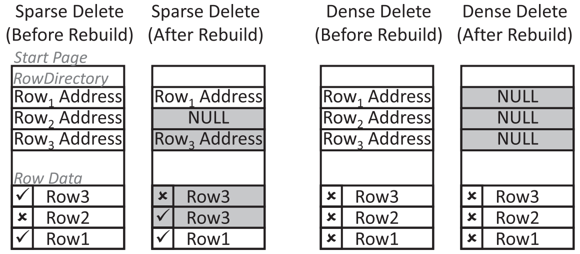
Sparse Delete (Before Rebuild)
Sparse Delete (After Rebuild)
Dense Delete (Before Rebuild)
Dense Delete (After Rebuild)
Start Page
RowDirectory
Row$_1$ Address
Row$_2$ Address
Row$_3$ Address
NULL
Row Data
Row3
Row2
Row1

Fig. 6. An example of how PostgreSQL reorganizes pages with sparse or dense deletes during table rebuild.

sparse case, but marked-as-deleted rows are preserved on the page (row directory NULLs are unrelated to deletion).

Related work

Wagner et al. (2015) introduced a database-agnostic mechanism to deconstruct database storage. That work described database page structure with layout parameters and performed a comparative study of multiple DBMSes. This work extends deconstruction to the “unseen” database storage, complementing recovery of regular table content from deleted and corrupted database files. Drinkwater had studied carving data out of SQLite storage (Drinkwater). SQLite had been the focus of forensic analysis particularly because it is used in Firefox (Pereira, 2009) and in a number of mobile device applications (Pieterse and Olivier, 2014). Chivers and Hargreaves (2011) investigated recovery of deleted records from the Windows Search database.

OfficeRecovery provides a number of commercially sold emergency recovery tools for corrupted DBMSes (OfficeRecovery and Recoverya; OfficeRecovery and Recoveryb; OfficeRecovery and Recoveryc) that support several versions of each DBMS. OfficeRecovery products recover most of database objects (except for constraints) – for Oracle that also includes backup file recovery which is not something we currently support because our primary focus is on a universal multi-DBMS tool. Percona Projects supplies a tool that recovers corrupted or deleted tables in MySQL (Percona and Percona data rec), but does not recover the schema (and in fact requires that the user to provide the descriptive data structure for the schema). Stellar Phoenix sells DB2 recovery software for IBM DB2 (UDB) v8 (Phoenix and recovery) as well as MS SQL Server for multiple versions (Phoenix and database).

Forensic data analysis in general is concerned with recovering partially damaged remnants of a file. Seminal work by Garfinkel (Garfinkel, 2007) discusses efficient file carving strategies that rely on file content rather than metadata, in order to restore files. Brown (Brown, 2013) presents a mechanism for recovering a compressed file that includes a corrupted region. Similarly, research that concentrates on the analysis of volatile memory looks for particular patterns in RAM. Grover (Grover, 2013) describes a framework for identifying and capturing data from an Android device in order to protect that device from malware or investigate its owner. Volatile data analysis also benefits from *stochastic forensics* defined by Grier (2011), which derives probabilistic conclusions about user actions based on side effects of these actions – in our case we study side-effects of user and database actions. Guido et al. (Guido et al., 2013) describes collecting data from a running Android device to identify patterns of malicious software, identifying malicious applications without an apriori known signature by observing system events in real-time. Work in Okolica and Peterson (2010) presents a generalized process of performing a version-agnostic Windows memory dump analysis. Similarly, we generalize the process of database carving (disk or RAM) across all DBMSes and operating systems.

Olivier (2009) characterized the differences between File System Forensics and Database Forensics, but did not implement a database reconstruction tool. Adedayo and Olivier (2012) described techniques for restoring database to an earlier version using the database schema and log file records. This requires a still-functional database, availability of the log files and a valid schema. Our work reconstructs data at the page level in database files without relying on any of these assumptions. We capture the full state of the database, including deleted data that has survived, rather than restoring the accessible (visible) parts of the database to an earlier version in time.

Conclusions and future work

In this work, we presented a tool that can recover data that exists outside of forensic analyst's field of view. Current tools cannot recover deleted rows and yet this is where we *begin* our analysis. We start with recovery of inaccessible rows and then move on to recovering database contents that analysts do not know about. Few people know just how much de-allocated storage is constantly being created and shifted around in storage (on disk and in memory).

We also demonstrated why simple recovery of phantom data is insufficient – to analyze the results, forensic analyst must understand how database storage works. There are changes that appear similar at a glance – e.g., both `DELETE` and `UPDATE` create a “deleted” row in storage. Normal DBMS operation creates strange storage artifacts – deleted or de-allocated page may be created through simple internal maintenance (with no human action).

We intend to open-source our tool for the community, after adding a number of other uses cases (beyond simple recovery). We want to evaluate database RAM storage changes in real-time, including non-page data content. We plan to expand our work beyond row-store DBMSes (e.g., columns-stores and key-value engines). Finally, it is our goal to incorporate additional intelligence to help forensic analysts present and explain digital evidence extracted from database storage.

References

Adedayo OM, Olivier MS. On the completeness of reconstructed data for database forensics. In: Digital forensics and cyber crime. Springer; 2012. p. 220–38.

Brown RD. Improved recovery and reconstruction of deflated files. Digit Investig 2013;10:S21–9.

Chivers H, Hargreaves C. Forensic data recovery from the windows search database. Digit Investig 2011;7(3):114–26.

Comer D. Ubiquitous b-tree. ACM Comput Surv 1979;11(2):121—37. http://dx.doi.org/10.1145/356770.356776. http://doi.acm.org/10.1145/356770.356776.

DB2, Automatic table maintenance in db2, part 2: Automatic table and index reorganization in db2 for linux, unix, and windows, http://www.ibm.com/developerworks/data/library/techarticle/dm-0707tang/.

Garfinkel SL. Carving contiguous and fragmented files with fast object validation. Digit Investig 2007;4:2—12.

Grier J. Detecting data theft using stochastic forensics. Digit Investig 2011;8:S71—7.

Grover J. Android forensics: automated data collection and reporting from a mobile device. Digit Investig 2013;10:S12—20.

Guido M, Ondricek J, Grover J, Wilburn D, Nguyen T, Hunt A. Automated identification of installed malicious android applications. Digit Investig 2013;10:S96—104.

S. Phoenix, Db2 recovery software, http://www.stellarinfo.com/database-recovery/db2-recovery.php.

S. Phoenix, Sql database repair, http://www.stellarinfo.com/sql-recovery.htm.

InnoDB Engine, http://www.innodb.com/.

MyISAM Storage Engine, http://dev.mysql.com/doc/refman/5.7/en/myisam-storage-engine.html.

Neil PO, Neil EO, Chen X, Revilak S. The star schema benchmark and augmented fact table indexing. In: Performance evaluation and benchmarking. Springer; 2009. p. 237—52.

OfficeRecovery, Recovery for postgresql, http://www.officerecovery.com/postgresql/.

OfficeRecovery, Recovery for postgresql, http://www.officerecovery.com/oracle/.

OfficeRecovery, Recovery for mysql, http://www.officerecovery.com/mysql/.

Okolica J, Peterson GL. Windows operating systems agnostic memory analysis. Digit Investig 2010;7:S48—56.

Olivier MS. On metadata context in database forensics. Digit Investig 2009;5(3):115—23.

Percona, Percona data recovery tool for innodb, https://launchpad.net/percona-data-recovery-tool-for-innodb.

Pereira MT. Forensic analysis of the firefox 3 internet history and recovery of deleted {SQLite} records. Digit Investig 2009;5(3.4):93—103. http://dx.doi.org/10.1016/j.diin.2009.01.003.

Pieterse H, Olivier M. Smartphones as distributed witnesses for digital forensics. In: Advances in digital forensics X. Springer; 2014. p. 237—51.

R. Drinkwater, Forensics from the sausage factory, http://forensicsfromthesausagefactory.blogspot.com/2011/04/carving-sqlite-databases-from.html.

Wagner J, Rasin A, Grier J. Database forensic analysis through internal structure carving. Digit Investig 2015;14:S106—15.