Database Forensic Analysis Through Internal Structure Carving

By

James Wagner, Alexander Rasin and Jonathan Grier

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2015 USA

Philadelphia, PA (Aug 9th - 13th)

Digital Investigation 14 (2015) S106–S115

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2015 US

Database forensic analysis through internal structure carving

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

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

Keywords:
Database forensics
File carving
Memory analysis
Stochastic analysis
Database storage modeling

A B S T R A C T

Forensic tools assist analysts with recovery of both the data and system events, even from corrupted storage. These tools typically rely on "file carving" techniques to restore files after metadata loss by analyzing the remaining raw file content. A significant amount of sensitive data is stored and processed in relational databases thus creating the need for database forensic tools that will extend file carving solutions to the database realm. Raw database storage is partitioned into individual "pages" that cannot be read or presented to the analyst without the help of the database itself. Furthermore, by directly accessing raw database storage, we can reveal things that are normally hidden from database users.

There exists a number of database-specific tools developed for emergency database recovery, though not usually for forensic analysis of a database. In this paper, we present a universal tool that seamlessly supports many different databases, rebuilding table and other data content from any remaining storage fragments on disk or in memory. We define an approach for automatically (with minimal user intervention) reverse engineering storage in new databases, for detecting volatile data changes and discovering user action artifacts. Finally, we empirically verify our tool's ability to recover both deleted and partially corrupted data directly from the internal storage of different databases.

Introduction

Because most personal and company data is stored in digital form, forensic analysts are often tasked with restoring digital data contents or even reconstructing user actions based on system snapshots. The digital data recovery process is composed of both hardware and software phases. Hardware techniques extract data from physically damaged disks, while software techniques make sense of the recovered data fragments. Our work presented here focuses on software-based restoration techniques in the context of relational database management systems (DBMSes). A well-recognized forensic technique is the process of "file carving" that bypasses metadata and inspects file contents directly. If a sufficient proportion of the file can be recovered and recognized, then the content of the file (e.g., images or document text) can then be restored.

It is our contention that a significant amount of data, particularly what is referred to as *Big Data*, is not stored in flat files, but rather resides in a variety of databases within the organization or personal devices. Standard file carving techniques are insufficient to meaningfully recover the contents of a database; indeed, without the metadata of the DBMS (*catalog*), the contents of database tables could not be presented to the forensic analyst in a coherent form. The work presented here thus bridges this gap by introducing a **novel database carving approach** that allows us to reconstitute database contents and reason about actions performed by the database users.

* 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.2015.05.013

Our contributions

We present a comprehensive collection of techniques for forensic analysis of both static and volatile content in a database:

- We define **generalized storage layout parameters** for parsing the raw storage (including the volatile kind) of many different relational databases.
- We compare and contrast **different storage design decisions** made by a variety of DBMSes and discuss the resulting implications for forensic analysis.
- We present a tool that can **reverse-engineer new DBMS storage parameters** by iteratively loading synthetic data, executing test SQL commands and comparing resulting storage changes.
- We also present a tool that, given a **disk image** or a **RAM snapshot** can do the following:
  - Identify intact **DBMS pages**, even for multiple DBMSes on the same disk, for all known storage configuration parameters.
  - Recover the **logical schema** (SQL tables and constraints) and all **database table rows** for known parameters (a parameter set will support several different versions of the DBMS, depending on storage changes version-to-version).
  - Extract a variety of **volatile data artifacts** (e.g., deleted rows or pre-update values).
  - Detect evidence of **user actions** such as row insertion order or recently accessed tables.

Paper outline

Fig. 1 shows the high-level architecture overview. In Section "Database storage structure" we review the principles of page-based data storage in relational databases and define the parameters for parsing and recovering these pages. In the same section we also summarize important database-specific storage structures (i.e., non-tables) and discuss the fundamentals of volatile storage and updates. In Section "Deconstructing database storage", we analyze the interesting storage layout parameter trade-offs and explain how these parameters and some user actions can be discovered within a DBMS. Section "Experiments" reports experimental analysis results for a variety of different databases and environment scenarios. Finally, Section "Related work" summarizes related work and Section "Conclusion and future work" contains the conclusions and mentions a number of promising future work directions.

Database storage structure

The storage layer in relational databases partitions all physical structures into uniform pages with a typical size of 4 or 8 KBytes because using a fixed page size significantly simplifies storage and cache management. Page size can be changed by the database administrator, but such a change requires rebuilding data structures: page size cannot be changed for individual tables, at a minimum it is global per tablespace. Two different layers of metadata are involved in database storage: the general information that describes where and how the tables are stored and the per-page metadata for the contents of each individual page. The forensic challenge lies in reconstructing all surviving database content directly from disk (or memory) image using only the metadata included with each page.

From a high level perspective, all relational database pages share the same general structure and break down into three components of interest: the header, the row directory and the row data itself. Depending on the specifics of each database, the page header stores general page information (e.g., *table or an index?* or *which table or index is it?*). This part of the overhead is found at the beginning of the page structure. The row directory component is responsible for keeping track of the row locations as new rows are inserted or old rows are deleted. This row directory may be positioned either between the page header and the row data or at the very end of the page following the row data. The third component is the row data structure that contains the actual page content along with some additional overhead. Fig. 2 shows an overview of how these structures typically interact within a page; the "other structures" area can contain other optional elements only relevant under specific circumstances (e.g., particular kinds of updates). We next describe the comprehensive set of

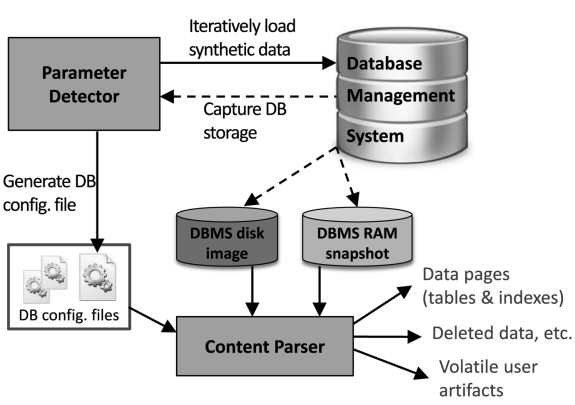

Fig. 1. Overview of parameter detection and data analysis.

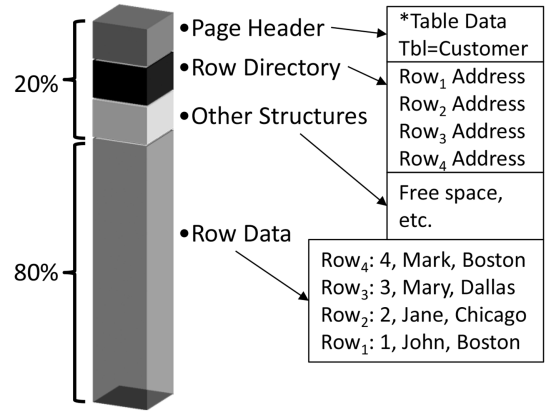

Fig. 2. A structural overview of a database page.

parameters used to parse table page storage for the eight different DBMSes.

Page storage layout parameters

The content of each component on a database page can be described by a general set of parameters. Creating a database-specific tool would have been significantly easier, but our goal was to develop a generalized forensic approach that can support many databases and can be easily expanded to support even more. Once the descriptive parameters have been generated for a particular database (Section "Parameter discovery" discusses how we automate this process), our code can identify pages and parse data from a page belonging to that database. The set of parameters presented here has been generalized to support a variety of DBMSes (listed in Section "Experiments"). In the rest of this section, we explain the purpose of each parameter for each page component. Section "Database storage parameter trade-offs" expands on this discussion by analyzing the significance of the parameters that affect interesting storage and reconstruction trade-offs.

Page header parameters. The page header contains values that define page identity and its membership in a database structure. Fig. 3 outlines the parameter and the corresponding page layout (the parameters are not stored contiguously, but the addresses are fixed). The *general page identifier address* locates the position of where the *general page identifier* can be found – *general page identifier* is used for detecting page presence in disk image and helps determine the type of the page contents (index vs table or other). The *structure identifier address* points us to the ID of the particular structure to which the page actually belongs (e.g., *customer* table). Finally, *unique page identifier address* and *unique page identifier size* allow detecting the unique ID of each particular database page.

Row directory parameters. The row directory maintains an overview of where each row is stored within the page – Fig. 4 provides a visual view of how that information is stored. The row directory contains a list of addresses where each address points to a particular row and may also keep track of deletions or modifications applied by database

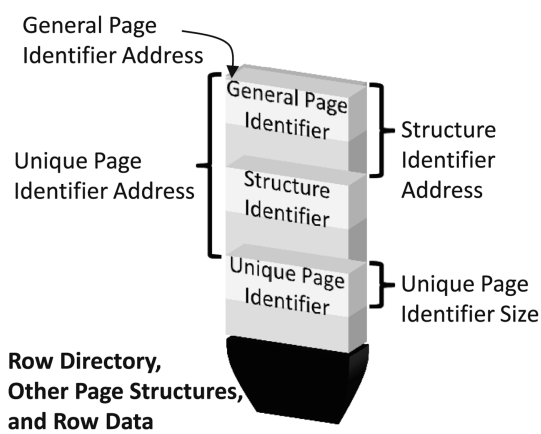

Fig. 3. Database page header structure.

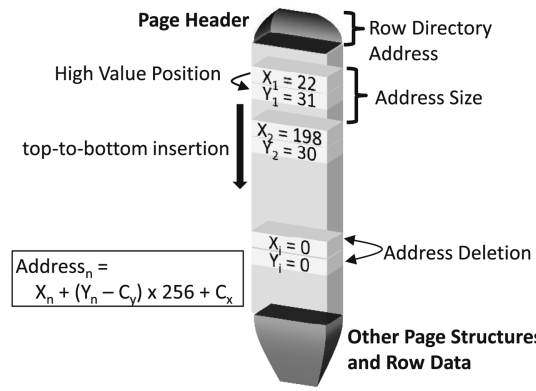

Fig. 4. Row directory structure.

users to each row. The *row directory address* determines the location or the very first address contained within row directory; the *address size* tells our tool how many bytes there are between each subsequent directory address. In order to deconstruct each address within row directory we need the *high value position* and the *address conversion constants* (C_x and C_y), which are then substituted into the Address_n formula in Fig. 4). The *high value position* determines which byte is the Y_n parameter in that computation. The *directory order sequence* simultaneously determines two things about row directory storage: the row directory may be stored after the page header and grow incrementally or at the end of the page with each new address preceding the previously added address. Fig. 4 assumes the former option, but Fig. 5 has examples of the latter. One variable not explicitly shown in Fig. 4 is the *slot size*, which determines whether the row directory is sparse or dense. That is, the *slot size* value of k means that row directory addresses only point to each k_{th} row in the page – setting k to value higher than 1 reduces the row directory overhead. Finally, the *address deletion* parameter tells us whether the address of a row in the row directory will be set to NULL when that row is deleted.

Row data parameters. The third and final component in a database page is the actual row data, which takes up the majority of page space. Table 1 lists the parameters used by our recovery tool, partitioned by parameter category and including a brief explanation. The *row identifier* is particularly significant because it represents a database-generated ID that does not belong to the data – if present, this identifier is generated in different ways by different DBMSes. The *column count* parameters helps us parse the column values from each row – note that *column count* does not always match the actual number of columns in a table because it may include the *row identifier*. As discussed in Section "Database storage parameter trade-offs", it is common for string values but not other types of values to include a size, so most of the *column size* settings refer to string columns (although we did observe this for NUMBER/DATE types in Oracle). The *column directory* in Table 1 is the per-row equivalent of the *row directory*. While the *row directory* stores addresses that locate rows in a page, the *column directory* keeps track of individual value addresses in each particular row. The *raw data* parameters describe

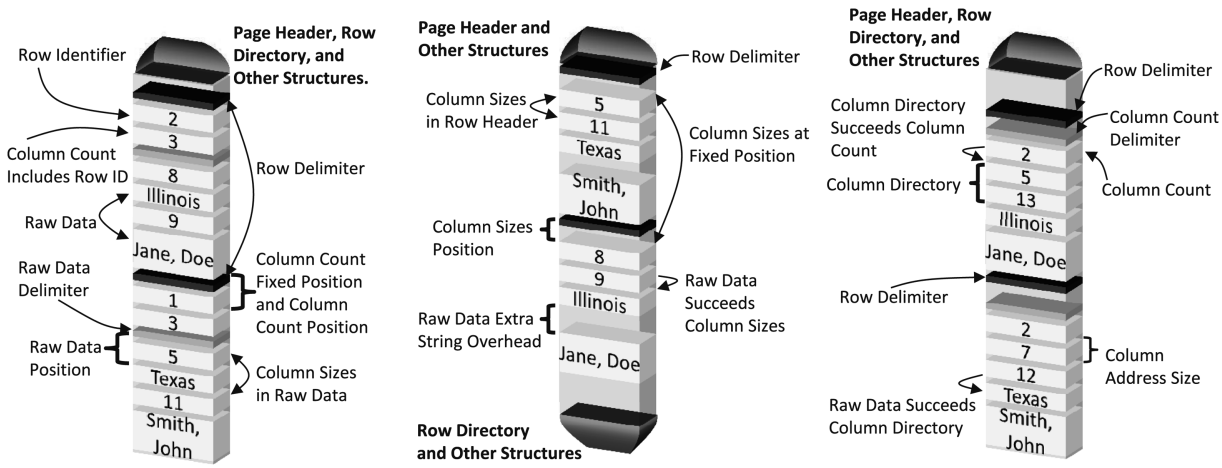

Fig. 5. Different layout choices for row data and row directory within a page.

the location and the delimiters and any other extra overhead present among the stored values, while the *data decoding* parameters describe the size information for individual values in the row.

Fig. 5 shows a few structurally different examples of row data layout on the page. Row data content may be placed in the middle (between page header and row directory) or at the end of the page. In addition to the actual raw values, this component may contain delimiters that separate different fields in a row, individual attribute sizes or the total count of columns.

Non-table database structures

While much of the data content resides in tables, other database structures can prove very useful for rebuilding data and reasoning about volatile changes. Here we briefly describe some of these structures although we do not include the corresponding parsing parameters or go into excessive detail owing to space limitations. Since data tables contain the "original" data, these auxiliary structures are redundant in that they may be dropped without losing any raw data; the most common reason for adding auxiliary structures is to shorten user query runtime.

Indexes. An index is an auxiliary structure that helps locate rows within a database table. For example, consider a *customer* table from SSBM benchmark by O.Neil et al. (2009) — a table is typically sorted by customer ID or not at all. A query searching for a particular city (city = 'Boston') would have to scan the entire table, but constructing an index on the *city* column can expedite the search by building an additional structure that stores value–pointer pairs (e.g., [Boston, Row#2], [Boston, Row#17], [Chicago, Row#12]). Index structures are stored in database pages with a layout similar to that of table pages described here (in many ways, an index is a table with two special columns).

Besides the data values, several unique pieces of information can be extracted from indexes. First, indexes are often automatically created for primary keys (that are part of join connections between tables) and thus help in reconstructing the logical table structure. Second, indexes are stored as a tree structure Comer (1979) in order to reduce maintenance overhead, which means values cannot be overwritten in-place (in tables, values can sometimes be overwritten in-place). As a result, changes made to an indexed column (*city* in our example) will, on average, remain visible (to our tool) in the index storage for much longer than in the original table.

Materialized views (MVs). Intuitively, MVs are dynamically constructed tables — not to be confused with regular views that are simply a "memorized" query and are not physically stored. For example, if the SQL query SELECT * FROM Customer WHERE City = 'Boston' (i.e., give me all customers from Boston) is executed often, the database administrator may choose to construct a BostonCustomers MV that pre-computes the answer in order to speed up that query. MVs are stored in the same way as tables but provide additional information because they are not necessarily updated at the same time as the source table.

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 to account B, transactions are used to ensure that the *transient* account state cannot be observed by the database users. 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., been "finalized").

Volatile storage

DBMSes maintain a cached set of data pages in RAM (*buffer cache*) to speed up query access and updates. When initially accessed, data pages are read into memory as-is; however, when database users make changes (e.g., inserts or deletes), pages would be modified in-place, creating so-called *dirty* pages. Eventually, the data page is evicted from the cache and written back to disk, replacing the original page. Just as when storing data on disk, there is a significant similarity in how different DBMSes handle page

Table 1
A list of parameters used to describe the row data of a database page.

Row structure	Parameter	Description
Row delimiter		The delimiter between rows.
Row identifier	Exists	Is an internal row identifier present? (not part of original data)
	Static size	Do all row identifiers have a fixed size?
Column count	Exists	Is a column count is present?
	Includes row identifier	Does the column count include the row identifier in the count?
	Delimiter	A delimiter used to locate column count that follows.
	Fixed position	Is the column count at a fixed position?
	Position	Location of the column count.
	Pointer exists	Does a pointer locate the column count?
	Pointer position	Address at which the column count is located.
	NULL markers	Is the column count marked by NULLs?
Column size	Exist	Do column sizes for strings exist in the page?
	In raw data	Are column sizes with the raw data?
	Float	Are column sizes at a floating location?
	Fixed position	Are column sizes at a fixed location?
	Position	Location of the column sizes within the page.
Column directory	Exists	Does a column directory exist? (pointers to row attributes)
	Address size	The size of a column address.
	Fixed position	Is the column directory at a fixed position?
	Succeeds column count	Are the column addresses found after the column count?
	Position	The location of the column directory.
Raw data	Extra string overhead	Size of extra overhead between strings.
	Fixed position	Is the raw data placed at a fixed location within a page?
	Succeeds column directory	Does the raw data follow the column directory on page?
	Delimiter	A delimiter used to locate the raw data.
	Succeeds header sizes	Does the raw data follow the column sizes in the row header?
	Position	The location of the raw data.
	NULL markers	NULL bytes used to locate the raw data.
Data decoding	String conversion constants	A set of constants used to decode string size.
	Numbers stored with strings	Are raw numbers stored with raw strings?
	Numbers static size	Is the same number of bytes are used to a store a number?
	Numbers size	The number of bytes used to store a number.

modification. Inserts, deletes and updates create opportunities for recovering old, new or even *tentative* data (transactions can be aborted). We discuss the implications of changing data in different DBMSes in Section "Reconstructing volatile artifacts".

There are several distinct stages at which volatile data changes may still be discovered. First, there are tentative changes that are only reflected in the memory cache – if the change is canceled, the dirty page will likely be simply discarded. Since indexes cannot be changed in-place, should the updated column be indexed, index changes will persist for a longer time, even if the update was canceled and data page discarded. Second, if the update is confirmed (transaction COMMIT), the dirty page will eventually be written to disk. In that case the updates will be visible for an even longer period of time until the page is rebuilt. Although physical data structures may be defragmented manually, this process is expensive and rarely invoked. Individual pages may also be occasionally rebuilt by the DBMS, depending on many factors (e.g., settings of when and how newly inserted rows can overwrite "free" space left by deleted rows).

Deconstructing database storage

In this section, we delve into how parameter usage varies between different DBMSes and discuss the implications of the storage design choices. Our tool currently supports eight distinct DBMSes: Oracle, PostgreSQL, MySQL, SQLite, Apache Derby, DB2, SQLServer and FireBird (Section "Experiments" lists DBMS versions and parameter settings).

Database storage parameter trade-offs

As illustrated in Table 2, the majority (six out of eight) of the DBMSes use the *structure identifier* which makes it easier to detect the presence of pages in the data image snapshot and simplifies reassembling DB structures from individual pages. For the remaining two databases, our tool has to rely on the *column count* to reconstruct the schema of each structure (both of these databases do use *column count*). Therefore in those two databases, two tables with identical schemas (same number of columns and all column types are the same) **may** be erroneously merged into one table when rebuilt. A *unique page identifier* is available in all but one of the databases, letting us match the identity of the same page (e.g., between on-disk and in-memory). In some cases, the *unique page identifier* is a composition of different IDs (e.g., file ID plus the page ID) providing some additional information. The choice of *row directory sequence* is split (five versus three) between the different DBMSes. The ordering of the row directory is helpful when recovering data because it determines in which sequence rows were initially inserted/added to the page. The presence or absence of the *row identifier* is evenly split between the different databases – in Section "Reconstructing volatile artifacts" we will also show that the presence of

Table 2
A summary of significant trade-offs made by DBMSes in page layout.

Parameter	Oracle	PostgreSQL	SQLite	Firebird	DB2	SQLServer	MySQL	ApacheDerby
Structure Identifier	Yes	No	Yes					No
Unique Page Identifier	Yes							No
Row Directory Sequence	Top-to-bottom insertion					Bottom-to-top insertion		
Row Identifier	No	Yes		No			Yes	
Column Count	Yes			No		Yes	No	Yes
Column Sizes	Yes				No		Yes	
Column Directory	No				Yes		No	
Numbers Stored with Strings	Yes				No		Yes	

the row identifier is particularly significant when recovering data in presence of updates and deletes.

Most databases use *column count* (six versus two), which simplifies the process of parsing the page. Without the explicit column count, additional effort is required for reconstructing table contents – in essence our tool would need to discover the schema (see Section "Parameter discovery"). Once the table schema has been determined, we use *structure identifier* to identify its other pages – in all of the databases we have seen so far, at least one of the *structure identifier* or *column count* was always present. Similarly to *column count*, *column sizes* are commonly present in a database page (in six out of eight databases). The use of column sizes is directly connected with presence of a *column directory* structure within the raw data. Intuitively, explicitly storing column sizes simplifies parsing the individual values; without sizes, databases use a directory that specifies how to find columns within the row. This parameter choice also coincides with the *raw numbers stored with strings* decision, as having a column directory means that the columns do not have to be stored sequentially and can be interleaved. However, even if strings and numbers are stored separately the relative ordering (among strings and among numbers) is still preserved.

Parameter discovery

With the exception of modest user intervention, the collection of storage parameters described in Section "Database storage structure" is automated in our tool. We use a combination of our own synthetically generated data and the SSBM benchmark data to iteratively populate a database and use the resulting storage snapshots to autodetect the parameter values.

Automated Parameter discovery. User intervention primarily involves creating a configuration file for our tool to define the following database characteristics: page size setting, directory where the database file(s) are stored, database name, and the login credentials that have sufficient privileges to create tables/load data. If this is a new DBMS, a wrapper class for that database needs to be created, which will expose a function that can take a user name, user password, database name and SQL file as arguments, and run the SQL commands against the database. During parameter discovery, we perform inserts individually (without a bulk loader) because such tools do not preserve the insert order of the rows.

The SQL schema file (e.g., CREATE TABLE commands) may require changes depending on the particular database because, unfortunately, different data types are defined inconsistently. For example, owing to legacy issues, Oracle uses the VARCHAR2 type instead of VARCHAR type. Also, in most databases implement DATE type differently (it may include the time or a separate TIMESTAMP may be present). Some global settings may also need to be adjusted: MySQL needs to have the storage engine set to InnoDB because the old storage engine (which is no longer used in recent versions) does not use pages.

Recovering database schema. If the table schema is not available and no *column count* is present in the pages, discovering the original schema requires additional work. Our tool approaches that problem by approximating the schema and parsing the data under that assumption. If the schema is incorrect, the parser eventually encounters an error while deconstructing the data and a new schema is attempted instead. Only three out of the eight databases may require this approach and, since they all include a *structure identifier*, once the schema of the page has been discovered, all other pages from the same structure are easy to identify.

By looking at the recovered data, we can also discover other components of the schema. We automatically identify columns that contain unique values throughout the entire table, which tells us that the column is likely to have a UNIQUE or a PRIMARY KEY constraint. By comparing these columns we can identify primary keys (because foreign keys refer to primary keys).

Reconstructing volatile artifacts

When database contents are updated, that action creates a number of opportunities. First, we can recover the newly introduced data from inserts and updates. Second, we can recover recently performed user actions (i.e., reconstructing the fact that data was inserted, deleted or updated). Third, we can discover information about the changes that were canceled and undone (i.e., aborted transactions). The latter category is the most interesting, because this information would normally be unavailable to users even if the database were operating normally.

INSERT. Insert operations supply relatively little information (beyond data itself) because a brand new row is created. We can use the storage order to reconstruct the order of insertion. For performance reasons, new rows would typically be appended to existing (partially free)

Table 3
A summary of supported parsing functionality.

Functionality	Supported DB
General page detection	[1, 2, 3, 4, 5, 6, 7, 8]
Table page parsing	[1, 2, 3, 4, 5, 6, 7, 8]
Index page parsing	[1, 2, 3, 4]
Materialized view parsing	[1, 2]
String decoding	[1, 2, 3, 4, 5, 6, 7, 8]
Integer decoding	[1, 2, 3, 4, 5, 6, 7, 8]
Date decoding	[1, 2, 3, 4, 5, 6, 7, 8]

[1]Oracle, [2]PostgreSQL, [3]SQLServer, [4]DB2, [5]MySQL, [6]SQLite, [7]Firebird, [8]ApacheDerby.

database pages as they are inserted into tables. We can also sometime determine if the entire page has been bulk loaded based on the insert pattern; if the rows were inserted individually, we can determine that insert order.

DELETE. The deletion of rows provides more information. Just as file systems marks a file "deleted", databases would mark rows "deleted" as well. Space limitations prohibit us from including the parameters that describe how data is deleted in different DBMSes; however, we do summarize what different databases actually do on delete. When a row is deleted in Oracle and ApacheDerby, the page header and row delimiter are marked. When a row is deleted in PostgreSQL, the page header and *raw data delimiter* are marked. When a row is deleted in MySQL, page header and row metadata is marked. When a row is deleted in SQLite, the page header is marked and the row identifier is deleted. When a row is deleted in DB2, SQLServer and Firebird, the page header is marked, and the row directory address is deleted.

UPDATE. Although from database user perspective an update is a combination of a delete followed by an insert, the underlying storage changes are handled very differently. As with deletes, we summarize how updates are handled by the different DBMSes. When a row value is updated with a new value of a size equal to or less than the previous entry for Oracle, SQLite, DB2, and SQLServer, the page header is marked and the old row data is overwritten in-place. When a row is updated to a size equal to or less than the previous row for PostgreSQL, the page header and raw data delimiter are marked and the old raw data is written over. When a row is updated to a size equal to or less than the previous row for MySQL and ApacheDerby, the page header and the row metadata are marked and the old raw data is written over. When a row is updated to a size equal to or less than the previous row for Firebird, the page header is marked and the rows are reinserted. The only behavior consistent among all databases is when a column is updated to a size larger than the previous row value, in which case the old row deleted and the new row is inserted.

Experiments

Our current implementation of the forensic analytic tool supports eight different RDBMS systems under both Windows and Linux OS. The breakdown of supported functionality is listed in Table 3 (more features are under development). The parsing rate currently falls in the range between 5.5 MB per second to 26.5 MB per second, depending on the specifics of each database storage layout. Our experiments were carried out using an Intel X3470 2.93 GHz processor with 8 GB of RAM; Windows servers run Windows Server 2008 R2 Enterprise SP1 and Linux experiments use CentOS 6.5. The cloud based instance in Experiment 3 used Intel Xeon 2.5 GHz processor with 2 GB of RAM. Windows operating system memory snapshots were generated using a command-line tool User Mode Process Dumper (version 8.1). This tool outputs a process memory dump for a given a process identification number. Linux operating system memory snapshots were generated by reading the process' memory under/proc/$pid/mem. For on-disk pages we either read the database storage files or deconstructed the data directly from a hard drive image since we do not need the file structure. Fig. 6 shows a few sample lines of output produced by our tool (for a table and an MV) in Windows.

Experiment 1: *Testing a variety of DBMS versions*. We begin by verifying that our carver tool supports different version of the eight databases. In our initial experiments we used the version that was easiest to acquire, but here we install and test a variety of other DBMSes, also verifying that our tool can handle both Linux and Windows. Table 4 summarizes different versions, operating systems and parameter settings that we used. Acquiring older versions of some databases has proven to be challenging, we also had difficulty installing some older software, such as PostgreSQL 6.3.2 (circa 1999) on our servers.

For databases listed in Table 4 we verified that our parameter discovery mechanism (described in Section "Parameter discovery") was able to auto-detect necessary parameters and successfully reconstruct data from pages. Not surprisingly, we found that for most alternate versions, the storage layout had not changed from version to version. However, we did find a number of changes in PostgreSQL

```
F:\userdump\memory\outputs>more ORACLE11R2t1_deletePages.txt +88759
PageType   Address      StrctID     Row Cnt   Col per Row   Datatypes   Rows
-------------------------------------------------------------------------------
Table      1065899570   154, 172      75          8         NSSSSSSS    961, Customer#0000000961, W0SZ2oflx9aWTggt
                                                                        962, Customer#0000000962, GnBVTZhkN gXW, E
                                                                        963, Customer#0000000963, X,mN4HNNQ4KatlaV
                                                                        964, Customer#0000000964, 2DpdvfAw4TuX18DV
                                                                        965, Customer#0000000965, 7IKpO1RNZIgyaq5R
                                                                        966, Customer#0000000966, 6ROS8aS4  INDIA

F:\userdump\memory\outputs>more ORACLE11R2t3_refreshMVPages.txt +142030
PageType   Address      StrctID     Row Cnt   Col per Row   Datatypes   Rows
-------------------------------------------------------------------------------
MV         1042472162   156, 172     142          3         SSS         Customer#000076256, CHINA    2, 28-327-58
                                                                        Customer#000076257, RUSSIA   3, 32-305-72
                                                                        Customer#000076258, ETHIOPIA 7, 15-585-17
                                                                        Customer#000076259, MOROCCO  0, 25-264-93
                                                                        Customer#000076260, UNITED ST9, 34-761-34
                                                                        Customer#000076261, SAUDI ARA2, 30-444-28
```

Fig. 6. Sample recovered data, both from a table page and an MV page.

Table 4
The comprehensive list of all databases used in this paper.

DBMS version	Testing OS	Buffer size(MB)	Page size(KB)
Apache Derby 10.10	Linux	400	4
Apache Derby 10.5	Linux	400	4
DB2 Express-C 10.5	Linux	400	4
Firebird 2.5.1	Linux	400	8
Firebird 2.1.7	Windows	400	8
MySQL Server 5.1.73	Linux	800	16
MySQL Server 5.6.1	Windows	800	16
Oracle 11g R2	Windows	800	8
Oracle 12c R1	Windows	1200	8
PostgreSQL 7.3	Linux	400	8
PostgreSQL 8.4	Linux	400	8
PostgreSQL 9.3	Windows	800	8
SQLite 3.8.6	Linux	2	1
SQLite 3.8.7	Windows	2	1
SQLServer 2008 Enterprise	Windows	800	8

7.3: the values for the *general page identifier* and its address, the *structure identifier position*, *row directory address*, the conversion constants for both row directory and string size computation and the delimiter used to separate row data have all changed to a different value between PostgreSQL 7.3 and PostgreSQL 8.4. Thus a variety of DBMS versions can be handled by the same set of known parameters but if the underlying storage changes, we need to detect the new parameters.

Experiment 2: *Rebuilding row data*. In this experiment we evaluate our tool's ability to reconstruct data straight from an image with DBMS pages. The process of rebuilding page contents is the same for disk or memory (the only difference being that an in-memory copy of the page may temporarily differ from its on-disk version due to updates). Furthermore, the contents of the database cache buffer provide some insight into the rows that were recently accessed by user queries, so we chose to visualize the database cache buffer as different queries are being executed. Fig. 7 shows the contents of the Oracle (50K pages) cache buffer, with each dot representing a single page and a bar chart summarizing the page counts. Initially buffer cache is prepopulated with synthetic data from several tables (aggregated into one bar in the bar chart), which is shown in the first snapshot and the corresponding bar chart below.

The second image in Fig. 7 shows cached pages after *customer* and *part* tables were queried for a total of about 7000 disk pages (using 50 different queries) with the corresponding bar chart below; the following two images show what happens after the *lineorder* table has been repeatedly accessed by queries. The third snapshot shows caching effects after executing 100 (120-page) *lineorder* queries (summarized in the third bar chart) and the fourth image shows the results of executing 200 more similar queries which effectively overwrite the entire cache buffer, replacing all of the previously cached data. While *lineorder* queries add up to approximately $(300 * 120)$ 36K pages, recall that indexes are commonly used to facilitate table access. Thus, there is a number of index pages, not shown on the bar chart, that are present in the last snapshot visualization.

The contents of the current buffer cache snapshot reflect the data that was accessed recently. However, note that all of the queries in this experiment were chosen to ensure that their pages are fully cached. A detailed discussion about database caching policies is beyond the scope of this paper, but note that when a query is accessing a large number of pages (e.g., more than one third of the total buffer cache size), only a particular portion of the read data is be cached. This is done to avoid evicting too many other table's pages from buffer cache and is used to reason about what table data was recently accessed.

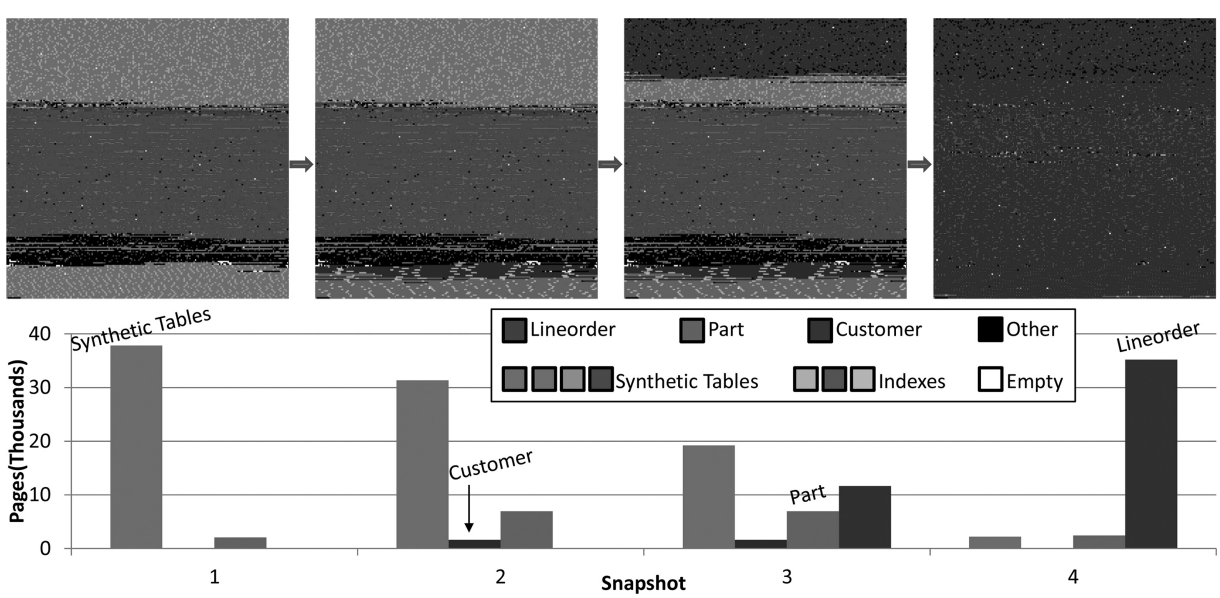

Fig. 7. Transformation of buffer cache contents as queries are executed.

Experiment 3: *Reconstructing corrupted data*. We next evaluate our forensic tool when the raw data has been damaged as well. Using one of the popular cloud service providers, we rented an instance and created a new database using PostgreSQL – here we use a cloud service to illustrate that data can be scavenged from neighboring or decommissioned instances if they are not properly sanitized (actually trawling the instances for private data would be against the ToS). After loading PostgreSQL with the SSBM benchmark (Scale4, 24M rows in the *lineorder* table), we have shutdown the database and **deleted** (using rm) the files that contained database storage.

Deleted disk space is marked "available" and will eventually be overwritten by new files. We simulate this overwrite process by performing random 1 KB writes throughout the disk image at random. We use small writes in order to test our tool's ability to rebuild pages when pages are partially damaged (if the entire page is overwritten, then it is simply gone). Once a certain percentage of 1 KB chunks was written to disk at random, we measured the amount of data that our tool could reconstitute. Table 5 summarizes the results in terms of the recovered table pages. The second column has the initial number of blocks, before any page damage had taken place, and then we show the distribution for 10% and 25% worth of damage. While the exact losses vary depending on each particular table's luck, the average number of restored pages closely matches the amount of inflicted damage.

Finally, note that running a query in PostgreSQL after overwriting page metadata caused The connection to the server was lost. Attempting reset: Failed.; changing the size of the table storage file (e.g., adding or removing a few bytes) caused ERROR: invalid memory alloc request size 2037542769.

Experiment 4: *The echoes of a database delete*. In this experiment, we test a DBMS to see when a deleted value is **really deleted**. Using Oracle, we created an index on the *phone* column in the *customer* table as well as a materialized view that contains a few of the customer columns, including *phone*. At time T_0, the phone value is present on disk in three different pages (in the table, the index and the MV). Table 6 shows the timeline of all three structures on-disk (HDD) and in-memory (RAM) – a ☎ symbol means that the phone number can also **still be returned by a SQL query** and both cmark ✓ and ✗ symbols mean that the value is **inaccessible by SQL** but **can be recovered by our tool**. The ✓ symbol means we can restore the phone number itself and ✗ symbol means that we can both extract the phone number and determine that it was already **marked as deleted**.

Table 5
Disk data damage experiment.

Damage	Dmg = 0%	Dmg = 10%	Dmg = 25%
Dwdate	35(100%)	31(88.6%)	20(57.1%)
Supplier	565(100%)	455(80.5%)	326(57.7%)
Customer	1915(100%)	1559(81.4%)	1075(56.1%)
Part	8659(100%)	6969(80.5%)	4864(56.2%)
Lineorder	115K(100%)	104K(89.9%)	87K(75.2%)
Total	416K(100%)	374K(89.9%)	312K(74.9%)

Table 6
A timeline for the **true** deletion of a deleted phone value.

Event	Table		Index		MV	
	HDD	RAM	HDD	RAM	HDD	RAM
T_0	☎		✓		☎	
T_1	✓	✗	✓	✓	☎	
T_2	✓	✗	✓	✓	☎	✓
T_3	✓	✗	✓	✓		
T_4	✗	✗	✓			
T_5	✗		✓			
T_6	✗					
T_7						

- At T_1 a phone row is deleted (including a COMMIT) by a user – this causes an index page with the phone (index values **cannot** be marked deleted) and a table page with the phone marked as deleted to be cached in RAM.
- At T_2 user queries the MV causing the phone page to be cached in RAM.
- At T_3 the MV is refreshed, the RAM page is removed and new MV no longer contains the phone (fragments of the old MV page **may** still be available in RAM).
- By T_4 a series of queries (enough to overwrite the buffer) are executed, evicting the index page from RAM. Because customer table is accessed by a user, the table page containing the deleted phone remains in RAM.
- By T_5 a long series of queries is executed during which customer table is not accessed, evicting the table page with phone entry from RAM.
- At T_6 the index is rebuilt and flushed from RAM.
- At T_7 the table is rebuilt and flushed from RAM.

Thus the deleted value is **truly gone** by time T_7 which, depending on database activity, may be a very long time away from time T_0. In some databases (including Oracle) MV behavior can be configured to automatically refresh; the value may also be overwritten by new inserts, but only after a certain number of rows on the page has been deleted.

Related work

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 (b,c,a) 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, 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 (a) as well as MS SQL Server for multiple versions Phoenix (b).

Forensic data analysis is generally concerned with recovering partially damaged remnants of a file, typically from a hard drive. Seminal work by Garfinkel (2007) discusses efficient file carving strategies that rely on file content rather than metadata, in order to restore the content of a hard drive. 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 (RAM flash memory) tends to look for particular patterns of interest. Grover (2013) describes a framework for identifying and capturing data from an Android device in order to protect that device from malware or investigate and/or audit its owner. Approaching volatile data analysis also benefits from *stochastic forensics* defined in Grier (2011), which derives probabilistic conclusions about user actions based on side effects of these actions. Our approach relies a similar idea, with page layout and database caching acting as side effects. Guido et al. (2013) describes collecting data from a running Android device to identify patterns of malicious software. The goal is to identify malicious applications without an apriori known signature by observing system events in real-time. Work by Okolica and Peterson (2010) presents a generalized process of performing a version-agnostic Windows memory dump analysis. Similarly, it is our goals is to generalize the process of database carving (disk or RAM) across all DBMSes and operating systems.

Conclusion and future work

We presented a forensic tool that can auto-detect internal DBMS storage mechanics for new databases and reconstruct the data structure and contents of known DBMSes. Due to the particular storage techniques employed by relational databases, our tool is able to restore any remaining fraction of a DBMS as well as already-deleted and otherwise inaccessible data. This generalized forensic tool can thus eventually supplant the DBMS-specific recovery tools currently available to forensic analysts. We intend to release our code to the wider community and think that it can also serve as an **independent** open-source auditing tool for all (including closed-source) DBMSes.

This work only begins to explore the possibilities opened up by looking into the raw database storage directly. In addition to the self-evident benefit of reconstructing database contents, we can learn a great deal of other subtler facts. DBMS data caching behavior can be directly observed to monitor user database activity based on internal caching heuristic rules; databases also cache a number of other elements of interest (e.g., SQL queries, raw user output) that can be captured. Finally, looking at the page storage offers precise fragmentation knowledge, which opens up opportunities for improving database design by performing tailored defragmentation.

References

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;7:114–26.

Comer D. Ubiquitous b-tree. ACM Comput Surv 1979;11:121–37. http://dx.doi.org/10.1145/356770.356776. URL, http://doi.acm.org/10.1145/356770.356776.

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

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.

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

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

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

okolica2010windows Okolica J, Peterson GL. Windows operating systems agnostic memory analysis. Digit Investig 2010;7:S48–56.

O.Neil P, O.Neil E, Chen X, Revilak S. The star schema benchmark and augmented fact table indexing. In: Performance evaluation and benchmarking. Springer; 2009. p. 237–52.

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;5:93–103. URL, http://www.sciencedirect.com/science/article/pii/S1742287609000048., http://dx.doi.org/10.1016/j.diin.2009.01.003.

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

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

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