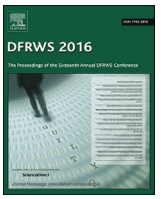
Digital Investigation 18 (2016) S97–S107

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

A B S T R A C T

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 $Index_{EmplD}$. 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).

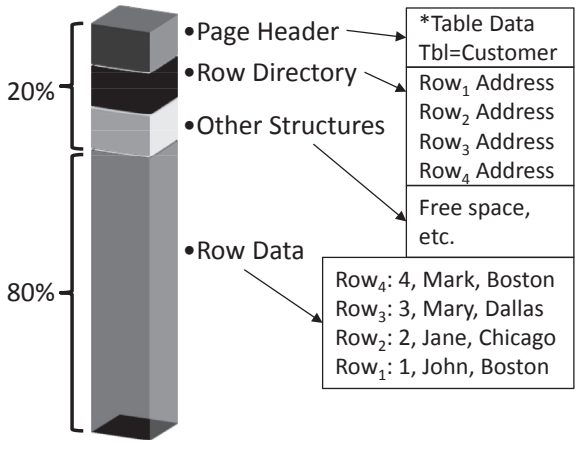


Fig. 1. An overview of database page structure.