Auditing database systems through forensic analysis

James Wagner
DePaul University, JWAGNE32@depaul.edu

Follow this and additional works at: https://via.library.depaul.edu/cdm_etd

Part of the Computer and Systems Architecture Commons, and the Data Storage Systems Commons

Recommended Citation
Wagner, James, "Auditing database systems through forensic analysis" (2020). College of Computing and Digital Media Dissertations. 23.
https://via.library.depaul.edu/cdm_etd/23

This Dissertation is brought to you for free and open access by the College of Computing and Digital Media at Via Sapientiae. It has been accepted for inclusion in College of Computing and Digital Media Dissertations by an authorized administrator of Via Sapientiae. For more information, please contact digitalservices@depaul.edu.
AUDITING DATABASE SYSTEMS THROUGH FORENSIC ANALYSIS

BY

JAMES WAGNER

A DISSERTATION SUBMITTED TO THE SCHOOL OF COMPUTING, COLLEGE OF COMPUTING AND DIGITAL MEDIA OF DEPAUL UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPAUL UNIVERSITY
CHICAGO, ILLINOIS
2020
DePaul University
College of Computing and Digital Media

Dissertation Verification

This doctoral dissertation has been read and approved by the dissertation committee below according to the requirements of the Computer and Information Systems PhD program and DePaul University.

Name: James Wagner

Title of dissertation:

AUDITING DATABASE SYSTEMS THROUGH FORENSIC ANALYSIS

Date of Dissertation Defense:

07/17/2020

Alexander Rasin
Dissertation Advisor*

Boris Glavic
1st Reader

Tanu Malik
2nd Reader

Jacob Furst
3rd Reader

Karen Heart
4th Reader (if applicable)

5th Reader (if applicable)

* A copy of this form has been signed, but may only be viewed after submission and approval of FERPA request letter.
The majority of sensitive and personal data is stored in a number of different Database Management Systems (DBMS). For example, Oracle is frequently used to store corporate data, MySQL serves as the back-end storage for many webstores, and SQLite stores personal data such as SMS messages or browser bookmarks. Consequently, the pervasive use of DBMSes has led to an increase in the rate at which they are exploited in cybercrimes. After a cybercrime occurs, investigators need forensic tools and methods to recreate a timeline of events and determine the extent of the security breach. When a breach involves a compromised system, these tools must make few assumptions about the system (e.g., corrupt storage, poorly configured logging, data tampering). Since DBMSes manage storage independent of the operating system, they require their own set of forensic tools.

This dissertation presents 1) our database-agnostic forensic methods to examine DBMS contents from any evidence source (e.g., disk images or RAM snapshots) without using a live system and 2) applications of our forensic analysis methods to secure data. The foundation of this analysis is page carving, our novel database forensic method that we implemented as the tool DBCarver. We demonstrate that DBCarver is capable of reconstructing DBMS contents,
including metadata and deleted data, from various types of digital evidence. Since DBMS storage is managed independently of the operating system, DBCarver can be used for new methods to securely delete data (i.e., data sanitization). In the event of suspected log tampering or direct modification to DBMS storage, DBCarver can be used to verify log integrity and discover storage inconsistencies.
ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Alex Rasin. If it was not for Alex, I would have probably not done my Ph.D. Alex saw my potential during my Masters' independent study, and he encouraged me to pursue my Ph.D. Alex spent countless hours working with me to develop as researcher (e.g., practicing presentations, writing, teaching me how to motivate projects). Alex has offered advice and mentored me on other skills that will allow me to be successful in my future career in academia, such as offering advice while teaching my first courses at DePaul. Without his support, I would not be where I am now.

I am very grateful to Karen Heart for always making herself available to contribute to our research projects. Karen was always willing to provide feedback on my work and took the time to work with me to resolve problems she saw. Even more specifically, Karen was generous with her time by serving as our mentor during the NSF I-Corps program.

I am grateful to Tanu Malik who not only made herself available to contribute to our research projects, but she also encouraged me to pursue new opportunities. Throughout my Ph.D., Tanu has directed toward several conferences and programs that have helped not only my research mature but given me valuable experiences to mature as a researcher.
I would also like to thank all of the other people that helped me reach graduation. Faculty that helped me to navigate my Ph.D., including Jacob Furst, Boris Glavic, and Jonathan Gemmell. My friends and lab mates that provided moral support, including Priya, Hai, and Caleb. I would also like to thank Jonathan Grier for providing support and contributing to our research projects.
## Contents

### 1 Introduction

- 1.1 Overview ................................. 3
  - 1.1.1 Part 1: Database Forensic Carving ................................. 3
  - 1.1.2 Part 2: Standardized Storage & Abstraction ......................... 3
  - 1.1.3 Part 3: Applications ........................................... 3
  - 1.1.4 Future Work and Conclusion .................................... 4

### 2 Background

- 2.1 Pages ....................................... 5
- 2.2 Auxiliary Objects ............................ 6
- 2.3 Other Database Operation Topics ......................... 8

### 3 Database Forensics

- 3.1 Introduction .................................. 11
  - 3.1.1 Our Contributions ............................... 12
- 3.2 Related Work .................................. 13
  - 3.2.1 File Carving ..................................... 13
  - 3.2.2 Database Forensics .............................. 13
- 3.3 Deconstructing Database Storage .................... 14
  - 3.3.1 Database Storage Parameter Trade-offs ............... 15
  - 3.3.2 Parameter Discovery .............................. 15
  - 3.3.3 Reconstructing Volatile Artifacts .................... 16
- 3.4 DBCarver Overview ............................ 18
- 3.5 The Life Cycle of a Row ....................... 18
  - 3.5.1 Page Structure ................................ 19
  - 3.5.2 Updated Rows .................................. 20
  - 3.5.3 Transactional Effects ............................. 21
- 3.6 The Life Cycle of a Page ...................... 21
4.2.3.5 Column Directory ........................................... 52
4.2.3.6 Raw Data ....................................................... 54
4.2.4 Data Encoding ................................................... 55
  4.2.4.1 Strings ...................................................... 56
  4.2.4.2 Integers ...................................................... 57
  4.2.4.3 Other Datatypes ............................................ 58
4.3 Carving Pages ..................................................... 58
  4.3.1 Page Header ................................................... 58
    4.3.1.1 General Page Identifier ................................. 59
    4.3.1.2 Structure Identifier .................................... 59
    4.3.1.3 Unique Page Identifier ................................. 59
    4.3.1.4 Record Count ............................................ 60
  4.3.2 Row Directory ................................................ 60
  4.3.3 Row Data ..................................................... 61
    4.3.3.1 Row Delimiter ........................................... 61
    4.3.3.2 Row Identifier ........................................... 61
    4.3.3.3 Column Count ............................................ 62
    4.3.3.4 Column Directory vs. Column Sizes ................. 62
    4.3.3.5 Column Directory ........................................ 63
    4.3.3.6 Column Sizes ............................................ 63
    4.3.3.7 Raw Data .................................................. 65
    4.3.3.8 Additional Obsolete Record Pass (Unallocated Storage Scan) ........................................... 65
  4.3.4 Data Decoding ................................................ 66
    4.3.4.1 ASCII Strings ........................................... 66
    4.3.4.2 Integers .................................................... 66
    4.3.4.3 Other Datatypes ........................................... 67

5 Standard Storage Format ......................................... 68
  5.1 Introduction .................................................... 68
  5.2 Related Work ................................................... 70
    5.2.1 Storage Format ............................................. 70
    5.2.2 View and Search Model .................................... 71
  5.3 Design Requirements .......................................... 73
    5.3.1 DB3F Requirements .......................................... 73
    5.3.2 DF-Toolkit Requirements ................................... 74
7.5 Detecting Inconsistencies for Read-Only Queries ............................................. 108
  7.5.1 Full Table Scan ................................................................. 108
  7.5.2 Index Access ................................................................. 109
  7.5.3 Data Lifetime in Memory .................................................... 110
7.6 Experiments .................................................................................. 110
  7.6.1 Experiment 1: DBDetective Performance Evaluation ...................... 111
  7.6.2 Record Modification Detection ................................................. 111
    7.6.2.1 Experiment 2: Deleted Record Detection ................................. 111
    7.6.2.2 Experiment 3: Updated Record Detection ................................. 113
    7.6.2.3 Experiment 4: Modified Record Detection ................................. 114
  7.6.3 Read-Only Query Detection .................................................... 116
    7.6.3.1 Experiment 5: Full Table Scan Detection ................................. 116
    7.6.3.2 Experiment 6: Index Access Detection ................................. 117
7.7 Conclusions and Future Work ........................................................... 118
8 File Tampering Detection ....................................................................... 119
  8.1 Introduction .............................................................................. 119
  8.2 Related Work ........................................................................... 121
    8.2.1 Database Auditing and Security ............................................ 122
    8.2.2 Database Forensics ............................................................. 122
    8.2.3 File Tampering and Anti-Forensics ....................................... 123
  8.3 Threat Model ............................................................................. 123
  8.4 File Tampering ........................................................................... 124
    8.4.1 Value Modification ............................................................. 126
    8.4.2 Record Addition ................................................................. 126
    8.4.3 Record Wiping ................................................................. 128
  8.5 Approach Overview ....................................................................... 128
  8.6 Forensic Analysis ......................................................................... 129
    8.6.1 Checksum Comparison ......................................................... 129
    8.6.2 Index Carving and Pointer Deconstruction ................................ 130
  8.7 Verifying Index Integrity .................................................................. 132
    8.7.1 B-Tree Integrity Verification .................................................. 133
    8.7.2 The Neighboring Value Problem .......................................... 133
    8.7.3 SQL Index Rebuild .............................................................. 134
    8.7.4 Manual Index Rebuild .......................................................... 134
  8.8 Index Sorting .............................................................................. 135
List of Figures

1.1 Dissertation overview. Publications include [86, 87, 85, 91, 93, 89, 88, 92].

2.1 Page Example: A) high-level, B) header, C) row directory, and D) row data.

2.2 An example of how an index references a record.

2.3 An example of how an index references an IOT record.

3.1 Overview of parameter detection and data analysis.

3.2 Deleted row examples for different DBMSes: 1) MySQL or Oracle 2) PostgreSQL 3) SQLite.

3.3 Example of how a deleted page is marked.

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

3.5 Transformation of buffer cache contents as queries are executed.

3.6 An example for row insertion behavior in SQL Server.

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

4.1 Record parsing overview.

5.1 The role of DB3F and DF-Toolkit in database forensics.

5.2 An example of DB3F header.

5.3 An example DB3F page JSON object.

5.4 DF-Toolkit evidence tree implemented with a user interface.

5.5 The relational schema used to store the evidence tree data in a SQL database.

5.6 DF-Toolkit filtering implemented with a user interface.

6.1 ODSA storage access.

6.2 ODSA completes raw database storage abstraction in an end-to-end process for storage access.

6.3 A sample set of ODSA calls.

6.4 The relational schema used to store ODSA data.
6.5 Using ODSA to find deleted records. ............................................. 95
6.6 Using ODSA to find all index entries for one record .......................... 96

7.1 Illustrates that the active records for Peter and Bob can be explained by 
audit log events, whereas the deleted record Malice can not be explained 
by any audit log events. ................................................................. 98
7.2 Architecture of the DBDetective. .................................................. 99
7.3 Detection of unattributed deleted records. ...................................... 105
7.4 Detection of unattributed inserted and updated records. ..................... 106
7.5 Matching index values to table records. ........................................ 108
7.6 Read-only query matching between disk and buffer cache. .................. 109

8.1 Example attack through DBMS files. .......................................... 120
8.2 Database file tampering examples. ............................................. 127
8.3 Architecture of the DBStorageAuditor. ........................................ 127
8.4 An example of mapping index values to a record. ............................... 131
8.5 Mapping index values to an IOT record example. ............................... 132
8.6 Preventing the neighboring value problem. ..................................... 133
8.7 An approximately sorted index example. ....................................... 136
8.8 Records added to the LINEORDER file. ...................................... 143

9.1 Storage layout of PLI and native database indexes (secondary and index- 
organized). ...................................................................................... 149
9.2 Architecture of PLI. ....................................................................... 152
10.1 Architecture of automated corpus generation framework. .................. 162
Chapter 1

Introduction

Cyber-crime (e.g., data exfiltration or computer fraud) is a significant concern in today’s society. A well-known fact from security research and practice is that unbreakable security measures are virtually impossible to create. For example, 1) incomplete access control restrictions allows users to execute commands beyond their intended roles, and 2) users may illegally obtain privileges by exploiting security holes in a Database Management System (DBMS) or OS code or through other means (e.g., social engineering). Thus, in addition to deploying preventive measures (e.g., access control), it is necessary to 1) detect security breaches in a timely fashion, and 2) collect evidence about attacks to devise countermeasures and assess the extent of the damage (e.g., what data was leaked or perturbed). This evidence provides preparation for legal action or can be leveraged to improve security and prevent future attacks.

DBMSes are targeted by criminals because they serve as repositories of data. Therefore, investigators must have the capacity to examine and forensically interpret contents of a DBMS. Currently, an audit log with SQL query history is a critical (and perhaps only) source of evidence for investigators \[51\] when a malicious operation is suspected. However, in field conditions, a DBMS may not provide the necessary logging granularity (unavailable or disabled). Moreover, the storage itself might be corrupt or contain multiple DBMSes.

Digital forensics provides approaches for an independent analysis with minimal assumptions about the environment. A particularly important and well-recognized technique is file carving \[25, 73\], which extracts files (but not DBMS files) from a disk image, including deleted or corrupted files. Traditional file carving techniques interpret files (e.g., JPEG, PDF) individually and rely on file headers. DBMS files, on the other hand, do not maintain a file header and are never independent (e.g., table contents are stored separate from table name and logical structure information). Even if DBMS files could be carved, they cannot be meaningfully imported into a different DBMS and must be parsed to retrieve their
content. To accomplish that task, DBMSes need their own set of digital forensics rules and tools.

Even in an environment with ideal log settings, DBMSes can not necessarily guarantee log accuracy or their immunity from tampering. For example, log tampering is a concern when a data breach originated from a privileged user such as an administrator (DBA or an attacker who obtained DBA privileges). Tamper-proof logging mechanisms were previously proposed [64, 80], but these only prevent logs from illegitimate modifications and do not account for attacks that skirt logging (e.g., logging was disabled). Knowing that even privileged users have almost no control of the lowest level DBMS storage, an analysis of forensic artifacts provides a unique approach to identify data tampering in an untrusted environment.

The primary goal of this work is to 1) develop forensic methods for DBMSes, and 2) use these methods to detect and describe security breaches in untrusted environments. A secondary goal of this work is to use our forensic methods to understand and optimize database storage beyond what DBMSes typically expose and support.

Figure 1.1: Dissertation overview. Publications include [86, 87, 85, 91, 93, 89, 88, 92, 90].
1.1 Overview

Figure 1.1 displays a diagram to visualize for the reader how the chapters of this dissertation are interrelated and serves as a guideline to read the material. Chapter 2 introduces database systems terminology and concepts used throughout this dissertation. Database storage at the page level, auxiliary object behavior, and additional database operations are covered.

1.1.1 Part 1: Database Forensic Carving

The first part of this dissertation focuses on collecting forensic artifacts from carving database storage. Chapter 3 presents our novel database forensic method, page carving, and introduces our page carving implementation, DBCarver. Page carving was developed as the database compliment to file carving to reconstruct database contents independent of the DBMS and OS. We previously presented page carving and DBCarver in [86, 87, 91]. Chapter 4 builds upon Chapter 3 by presenting the algorithms used by DBCarver in detail. The difference between these two chapters is that the material presented in Chapter 3 is based on the research published in [86, 87, 91], whereas Chapter 4 discusses algorithms in more detail than would typically be included in a research paper and are more appropriate for a patent.

1.1.2 Part 2: Standardized Storage & Abstraction

The second part of this dissertation extends the work in Part 1 by presenting a standard storage format for database forensics and an API to access this data. Chapter 5 describes our standard storage format, the Database Forensic File Format (DB3F), to store page carving results. DB3F abstracts DBMS storage engine specifics that can be used by all database carving tools, not just DBCarver, simplifying application development in Part 3 of this dissertation. Chapter 5 is based on the research published in [88]. Chapter 6 extends the work in Chapter 5 by presenting an API called Open Database Storage Access (ODSA), which the applications described in Part 3 can use to access DB3F files. Chapter 6 is based on the work published in [92, 90].

1.1.3 Part 3: Applications

The third part of this dissertation presents applications that require the database forensics data extracted by approaches in Part 1. Chapter 7 addresses an attack vector where logging was disabled by a database administrator (or an attacker with similar privileges). Chapter 7
is based on the worked published in [85]. Chapter 8 addresses direct DBMS file tampering without SQL commands by a system administrator. Chapter 8 is based on the work published in [89]. Finally, Chapter 9 presents a method for approximate data clustering. Rather than strictly ordering data (to improve data access speed), sections of approximately ordered data can be scanned at with competitive query runtimes. Chapter 9 is based on the worked published in [93]. While this is not an exhaustive list of applications that require database forensic data, other applications remain as future work.

1.1.4 Future Work and Conclusion

The final part of this dissertation consists of an overview of future work in Chapter 10 and a conclusion in Chapter 11. Future work describes our plans for a more thorough testing of approaches presented in this dissertation and of database forensics in general; it also discusses building on our novel methods in database forensics to expand this research to all of digital forensics.
Chapter 2

Background

The material in this dissertation requires access and modifications to database storage at the page level. Internal database storage at the page level is, by design, hidden from users – and thus our approach requires an understanding of database storage. In this chapter, we provide a generalized description of database storage at the page level for all (relational) DBMSes and define the terminology used throughout this dissertation. The terminology and concepts in this chapter apply (but are not limited) to IBM DB2, Microsoft SQL Server, Oracle, PostgreSQL, MySQL, Apache Derby, MariaDB, SQLite, and Firebird.

2.1 Pages

The storage layer in relational DBMSes partitions all physical structures (e.g., tables, indexes, and system catalogs) into fixed size pages (typically 4, 8, or 16 KB). A fixed page size across an entire database instance significantly simplifies storage and cache management.

When data is inserted or modified, the DBMS controls data placement within pages and internally maintains additional metadata. Despite the wide variety of DBMSes from different vendors on the market, many commonalities exist between DBMSes in how data is stored and maintained at the page level. Every row-store (storing records values on the same page) DBMS uses pages with three main structures: header, row directory, and row data. Figure 2.1A displays a high-level breakdown of a page with all three of these structures.

The page header contains metadata describing the user records stored in the page. Figure 2.1B demonstrates how some of this metadata could be positioned in a page header. The checksum is used by the DBMS to detect page corruption – whenever a page is modified, the checksum is updated. The object identifier represents the object to which the page belongs. The name of an object (e.g., table name) is not stored in a page, but the object
identifier can be mapped to the system catalog data to retrieve plaintext name. Depending on a DBMS, the page identifier is unique to each page for either each object, within a file, or across all database files. The record count refers to the number of active records within a page. If a record is deleted in a page the record count is decremented by one; if a record is added to a page, it is incremented by one.

The row directory stores pointers to each record – when a record is added to a page, a corresponding pointer is added to the row directory. Figure 2.1.C shows an example of how the row directory could be positioned. The row data segment stores user data along with metadata that describes record layout. Figure 2.1.D shows an example of how the row data may be structured (within minor DBMS-specific variations). In this example, each record stores a row delimiter, row identifier, column count, value sizes, and the user data values. The row delimiter marks the start of a record and is typically where row directory entries point to. The row identifier is a subset of an internal database pseudo-column. The column count represents the number of columns in that record. The sizes of values are typically stored for strings, but not other data types (e.g., integers).

### 2.2 Auxiliary Objects

**System Catalog** The system catalog refers to the data and metadata maintained by the DBMS. The system catalog is stored in tables and pages just like user data (with a few variations in the row data). Sometimes the system catalog tables use domain datatypes that are not available to the user (e.g., the Object Name datatype in PostgreSQL). Examples
of data and metadata stored in the system catalog are object types (e.g., table or index), object plaintext name (e.g., customer or employee), and object identifier, which is a unique identifier for each object stored in the user data page headers.

Users interact with DBMS tables; however, multiple copies of user data are stored in many other internal objects. Copies are stored in auxiliary objects (e.g., indexes, materialized views) that are used for improving query performance or for constraint enforcement. Note that indexes are created both by explicit user commands or automatically by the DBMS itself (e.g., primary key or unique constraint).

**Indexes** An index stores value-pointer pairs (typically a B-Tree structure) to locate rows within a table, providing performance benefits. A DBMS sometimes creates indexes automatically – for example, constraints (e.g., primary key or UNIQUE) cause force an index build. Index value-pointer pairs are stored in pages as the one in Figure 2.1 – an index is structurally similar to a table that stores (value, pointer) records. It is important for this dissertation to note that NULL values are not stored in indexes.

Figure 2.2 displays an example index page, and how a value references a record. A pointer to a table record is stored with each city value. Here, the pointer stores the page identifier, 8, and the respective row identifier, 25.

**Index Organized Tables** While MySQL is the only of the known DBMS that creates index organized tables (IOTs) by default, IOTs are often used in other DBMSes under different names (e.g., IOT table in Oracle or index included columns in Microsoft SQL Server).
An IOT is structured as a traditional B-Tree index on the primary key, and all remaining columns are included columns (or not used for ordering).

Figure 2.3: An example of how an index references an IOT record.

Figure 2.3 illustrates how a secondary index on the City column points to the record stored in an IOT. Just as in Figure 2.2, the City index stores value-pointer pairs. However, in this case the secondary index points to an intermediate page of the IOT B-Tree. The intermediate page is then used to retrieve the table record from the IOT leaf page.

**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 precomputes 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.

### 2.3 Other Database Operation Topics

**Constraints**  A fundamental rule that exists in all relational databases is that a relation (table) is a set of tuples (rows). Each row must be unique, enforced through a primary key.
Keys are always unique and by definition can never be NULL (“unknown” or “undefined”); the DBMS automatically blocks any operation that attempts to violate that rule.

Uniqueness of the primary key is further used to enforce integrity through foreign keys. A foreign key is a cross-table reference: for example, a loan payment record holds the loan ID (primary key) to which it belongs. Referential integrity requires such foreign key references to always be valid – either reference an existing record (e.g., an existing loan ID) or contain NULL as a placeholder.

Relational database rules furthermore impose a constraint on every table column. Each column must have a well-defined data type such as INTEGER or VARCHAR(15). As with other constraints, the DBMS actively enforces these rules by checking every operation that attempts to modify stored values. Any step found to be in violation of these rules (e.g., inserting a sixteen character string into VARCHAR(15)) is blocked.

**Query Execution** A DBMS engine has two strategies to fetch data from tables: 1) an index access performs a targeted data retrieval (i.e., use an index to fetch relevant rows), or 2) a full table scan searches the entire table reading both relevant and irrelevant rows. An index access is only used when an index is available and deemed to be cheaper than a full table scan.

An SQL query that accesses multiple tables combines them through a join operation. The default join type is an INNER JOIN, which combines only the matching rows. Using our loan example, a report about loans and loan payments includes only loans with existing payments – a loan without any associated payments is excluded in the join result. Several other operations such as NATURAL JOIN or subqueries are also executed as an INNER JOIN. An SQL query may explicitly request that unmatched values be included in the result – this operation is known as an OUTER JOIN. An OUTER JOIN in our loan example returns both loans with and without associated payments, substituting NULLs for missing loan payment data.

**Database Caching** In order to explain why the buffer cache contents can be used to determine how the data was possibly accessed, we will briefly discuss database memory management concepts. DBMSes manage their own cache separate from the operating system. Unallocated space in the buffer cache is commonly referred to as a free buffer. A free buffer is available to store a page accessed from persistent storage. When a query is issued, the buffer cache is first searched for the relevant pages. If a page needed to satisfy a query is found in the buffer cache, the page can be read directly from memory. This is referred to as a cache hit. A cache miss refers to when the page is not in memory and
must be read from persistent storage. When a cache miss occurs a free buffer is required. The least-recently used (LRU) replacement policy is one of the most commonly used ways to create free buffers. Fundamentally, the LRU replacement policy evicts pages with the oldest cache hit time. However, some DBMSes may implement a slight variation where eviction may not occur strictly on pages with the oldest cache hit time.
Chapter 3

Database Forensics

3.1 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 storage 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 DBMSes. A well-recognized forensic technique is the process of file carving that reconstructs file contents directly without the use of any file system metadata. If a sufficient portion 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, page carving, that allows us to reconstruct database contents and reason about actions performed by the database users.

Our motivating philosophy is that a comprehensive analytic method should reconstruct everything from all databases. Beyond simple recovery, forensic analysts will benefit from seeing the “hidden” content, including artifacts in unallocated storage.

In this chapter and the next chapter, we deconstruct database storage and present generalized techniques for reconstructing any database content. We use our page carving method to restore deleted and unallocated data across a variety of different DBMSes. This chapter
provides a high-level overview of page carving, and Chapter 4 presents a more detailed description of the page carving parameters and operation.

### 3.1.1 Our Contributions

The next two chapters present a comprehensive collection of techniques for forensic analysis of persistent, volatile, and unallocated database content. The research contributions in these chapters include:

- A defined set of **generalized storage layout parameters** used to parse the raw storage of commonly used relational DBMSes.

- A comparison of **different storage design decisions** made by these DBMSes and discussion of the resulting implications for forensic analysis.

- A method to **reverse-engineer new DBMS storage parameters** by iteratively loading synthetic data, executing test SQL commands, and observing resulting storage changes.

- We also present a tool, **DBCarver**, that, given a **disk image** or a **RAM snapshot** does the following:
  
  - Identifies intact **DBMS pages**, even for multiple DBMSes on the same disk, for all known storage configuration parameters.

  - Recovers 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).

  - Extracts a variety of **volatile data artifacts** (e.g., deleted rows or pre-update values).

  - Detects evidence of **user actions** such as row insertion order or recently accessed tables.

- 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 page carving and DBCarver against many DBMSes.

3.2 Related Work

3.2.1 File Carving
Forensic data analysis is generally concerned with recovering partially damaged remnants of a file, typically from a hard drive. Seminal work by Garfinkel [25] discusses efficient file carving strategies that rely on file content rather than metadata, in order to restore the content of a hard drive. [6] 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. [31] 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 [29], 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. [34] 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 [57] 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.

3.2.2 Database Forensics
Drinkwater had studied carving data out of SQLite storage [17]. SQLite had been the focus of forensic analysis particularly because it is used in Firefox [66] and in a number of mobile device applications [70]. [11] investigated recovery of deleted records from
the Windows Search database. OfficeRecovery provides a number of commercially sold emergency recovery tools for corrupted DBMSes [54, 56, 55] 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 [65], 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 [68] as well as MS SQL Server for multiple versions [69].

Oliver [58] characterized the differences between File System Forensics and Database Forensics, but did not implement a database reconstruction tool. Adedayo [4] 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.

### 3.3 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 3.8 lists DBMS versions and parameter settings).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>Firebird</th>
<th>DB2</th>
<th>SQL Server</th>
<th>MySQL</th>
<th>Apache Derby</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure Identifier</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Unique Page Identifier</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row Directory Sequence</td>
<td></td>
<td>Top-to-bottom insertion</td>
<td>Bottom-to-top insertion</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row Identifier</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column Count</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column Sizes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column Directory</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numbers Stored with Strings</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td></td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: A summary of significant page layout trade-offs made by DBMSes.
3.3.1 Database Storage Parameter Trade-offs

As illustrated in Table 3.1, 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 the sequence in which 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 3.3.3 we will also show that the presence of 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 3.3.2). 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.

3.3.2 Parameter Discovery

With the exception of modest user intervention, the collection of storage parameters described in Chapter 4 is automated in our tool. We use a combination of our own syntheti-
cally generated data and the SSBM benchmark data to iteratively populate a database and use the resulting storage snapshots to auto-detect 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).

### 3.3.3 Reconstructing Volatile Artifacts

When database contents are updated, that action creates a number of opportunities. First, we 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) 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. 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.
3.4 **DBCarver Overview**

![Diagram](image)

Figure 3.1: Overview of parameter detection and data analysis.

Figure 3.1 shows the high-level architecture overview of **DBCarver**. Algorithm 1 describes the overall page carving process. A file and a database parameter file are passed as an input. For every *general page identifier* found in the image file, **DBCarver** 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 is recorded for each *general page identifier*. Finally, the **DBCarver** parser returns the list of all pages.

The remainder of this chapter discusses the reconstruction of deleted data followed by a thorough experimental evaluation of page carving in Section 3.8. Section 3.5 deals with deleted record reconstruction, Section 3.6 addresses unallocated pages, and Section 3.7 considers additional copies of data that are left behind following a deleted. Chapter 4 provides more detailed description of the **DBCarver** algorithms.

### 3.5 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
Algorithm 1 DBCarver Parsing Algorithm

Require: (Any image file, database parameter file)

1: for each GeneralPageIdentifier in image file do
2:    set PageHeader and RowDirectory parameters
3:    for each ValidAddress in RowDirectory do
4:       append ValidAddress to Addresses
5:    for each Address in Addresses do
6:       set RowParameters
7:       Row ← ParseRowData()
8:       append Row to RowList
9:    append (PageParameters, RowList) to PageList
10: return PageList

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.

3.5.1 Page Structure

Deleted rows can both be recovered and explicitly identified as “deleted” by DBCarver. In contrast, a discarded page (see Section 3.6) 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 3.2. MySQL and Oracle mark the row delimiter at the position stored in the row directory address – a deleted row can be identified using Table 3.2 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.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Parameter</th>
<th>Active</th>
<th>Deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>Row Delimiter</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>Oracle</td>
<td>Row Delimiter</td>
<td>44</td>
<td>60</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Data Delimiter</td>
<td>2, 9, 24</td>
<td>2, x, 24</td>
</tr>
<tr>
<td>SQLite</td>
<td>Row Identifier</td>
<td>4 unique bytes</td>
<td>0,226,0,57</td>
</tr>
</tbody>
</table>

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

Table 3.2: Row data parsing parameters used to identify deleted rows.

Figure 3.2 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. Figure 3.2 omits DB2 and SQL Server as they only alter the row directory on deletion.

### 3.5.2 Updated Rows

When a row is updated, it can be updated in-place (see Section 3.7) 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
3.5.3 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.

3.6 The Life Cycle of a Page

3.6.1 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 3.5).

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. 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 [14]. 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 DBCarver (discarded and active pages are usually identical), but to identify whether a page is discarded we need to look at system tables.

### 3.6.2 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 3.5.1) 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 3.3. 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>
<thead>
<tr>
<th>Datatype</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID</td>
<td>4 bytes</td>
<td>Identifier to represent objects.</td>
</tr>
<tr>
<td>Name</td>
<td>64 bytes</td>
<td>Name of objects.</td>
</tr>
<tr>
<td>Aclitem</td>
<td>Variable</td>
<td>An array of access privileges.</td>
</tr>
<tr>
<td>XID</td>
<td>4 bytes</td>
<td>Transaction identifier.</td>
</tr>
</tbody>
</table>

Table 3.3: MV refresh options for each database.

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, MySQL still uses an older MyISAM 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). DBCarver 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 DBCarver).

Figure 3.3 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 3.5. 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.

![Diagram of Table Catalog Page and Customer/Supplier Tables]

Figure 3.3: Example of how a deleted page is marked.

### 3.7 The Life Cycle of a Value

Database tables are stored and processed as a collection of individual rows that comprise them. In Section 3.6 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.

#### 3.7.1 Auxiliary Structure: Indexes

Chapter 2 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 [12], 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 (Experiment 3.8.7).
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 (see Chapter 2) 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. Figure 3.4 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.

Figure 3.4: Example of how a deleted value can still live in an index.

Row from an aborted insert is treated as if it were 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.

### 3.7.2 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 3.4 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).

<table>
<thead>
<tr>
<th></th>
<th>DBMS</th>
<th>Function</th>
<th>Commit</th>
<th>On Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
<td>✓</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>Oracle</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td></td>
<td>✓</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>SQLite</td>
<td></td>
<td>✓</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>SQL Server*</td>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

*Indexed views are immediately refreshed. The user cannot change this setting.

Table 3.4: MV refresh options available in each database.

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 3.5 and 3.6 because this MV is really a table.

### 3.8 Experiments

Page carving was found to be applicable for at least ten different row-store DBMSes under both Windows and Linux operating systems. The experiments in this section present results using six representative databases (Oracle, SQL Server, PostgreSQL, DB2, MySQL and SQLite). 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). Experiment were performed on
<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Purpose:</strong> Verify that page carving supports many row-store DBMSes. <strong>Results:</strong> Table 3.6 lists eight different DBMSes (along with multiple versions) for which page carving was tested.</td>
</tr>
<tr>
<td>2</td>
<td><strong>Purpose:</strong> Demonstrate that page carving works for non-DBMS files, specifically a buffer cache snapshot. <strong>Results:</strong> Figure 3.5 presents the carved results from a series of RAM snapshots.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Purpose:</strong> Use page carving to determine the lifetime of deleted data. <strong>Results:</strong> Table 3.7 presents the data reconstructed using page carving following random writes to a disk image.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Purpose:</strong> Use page carving to determine the lifetime of deleted data. <strong>Results:</strong> Table 3.8 summarizes the lifetime of a record and its data copies.</td>
</tr>
<tr>
<td>5</td>
<td><strong>Purpose:</strong> Determine operations that damage a record to a point where it can no longer be carved. <strong>Results:</strong> Page carving reconstructed 40-100% of the deleted rows following simulated database activity.</td>
</tr>
<tr>
<td>6</td>
<td><strong>Purpose:</strong> Show that an old record version from an UPDATE can be reconstructed. <strong>Results:</strong> For up to 14% of updated rows, the full pre-update version of the row can be recovered.</td>
</tr>
<tr>
<td>7</td>
<td><strong>Purpose:</strong> Show the data that can be carved from indexes. <strong>Results:</strong> Values were reconstructed from active and deallocated index pages.</td>
</tr>
<tr>
<td>8</td>
<td><strong>Purpose:</strong> Show that aborted transactions leave behind storage artifacts. <strong>Results:</strong> Canceled transactions leave just as many recoverable values in storage as regular transactions.</td>
</tr>
<tr>
<td>9</td>
<td><strong>Purpose:</strong> DBCarver can recover 0.5% deleted rows and duplicate active rows after MV rebuild.</td>
</tr>
<tr>
<td>10</td>
<td>Table rebuild leaves behind 1) 85% deleted rows or 2) a large number of duplicate active rows.</td>
</tr>
</tbody>
</table>

Table 3.5: Summary of experimental results in this section.

servers with Intel X3470 2.93 GHz processor and 8GB of RAM; Windows servers run Windows Server 2008 R2 Enterprise SP1 and Linux experiments used CentOS 6.5. Either the database files or the raw hard drive images were directly read since page carving does not rely on file system structure.

### 3.8.1 Experiment 1: DBMS Support Testing

The purpose of this experiment was to verify that page carving supports many row-store relational DBMSes, various versions of each DBMS, and DBMSes that run on both Linux and Windows operating systems. Table 3.6 summarizes the DBMS versions, operating systems, and parameter settings that were used. Acquiring older versions of some databases proved to be challenging, and we also had difficulty installing some older software, such as PostgreSQL 6.3.2 (circa 1999) on our servers.
For the DBMSes listed in Table 3.6, we confirmed that our parameter discovery mechanism (described in Section 3.3.2) 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 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.

<table>
<thead>
<tr>
<th>DBMS Version</th>
<th>Testing OS</th>
<th>Buffer Cache Size(MB)</th>
<th>Page Size(KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Derby 10.10</td>
<td>Linux</td>
<td>400</td>
<td>4</td>
</tr>
<tr>
<td>Apache Derby 10.5</td>
<td>Linux</td>
<td>400</td>
<td>4</td>
</tr>
<tr>
<td>DB2 Express-C 10.5</td>
<td>Linux</td>
<td>400</td>
<td>4</td>
</tr>
<tr>
<td>Firebird2.5.1</td>
<td>Linux</td>
<td>400</td>
<td>8</td>
</tr>
<tr>
<td>Firebird2.1.7</td>
<td>Windows</td>
<td>400</td>
<td>8</td>
</tr>
<tr>
<td>MySQLServer5.1.73</td>
<td>Linux</td>
<td>800</td>
<td>16</td>
</tr>
<tr>
<td>MySQLServer5.6.1</td>
<td>Windows</td>
<td>800</td>
<td>16</td>
</tr>
<tr>
<td>Oracle11gR2</td>
<td>Windows</td>
<td>800</td>
<td>8</td>
</tr>
<tr>
<td>Oracle12cR1</td>
<td>Windows</td>
<td>1200</td>
<td>8</td>
</tr>
<tr>
<td>PostgreSQL7.3</td>
<td>Linux</td>
<td>400</td>
<td>8</td>
</tr>
<tr>
<td>PostgreSQL8.4</td>
<td>Linux</td>
<td>400</td>
<td>8</td>
</tr>
<tr>
<td>PostgreSQL9.3</td>
<td>Windows</td>
<td>800</td>
<td>8</td>
</tr>
<tr>
<td>SQLite3.8.6</td>
<td>Linux</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SQLite3.8.7</td>
<td>Windows</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SQLServer 2008 Enterprise</td>
<td>Windows (Linux)</td>
<td>800</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.6: The comprehensive list of all databases used in this chapter.

3.8.2 Experiment 2: Rebuilding row data

The purpose of this experiment was to evaluate page carving’s ability to reconstruct data from a non-DBMS file. 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. Figure 3.5 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 Figure 3.5 displays pages cached after CUSTOMER and PART tables were queried for a total of about 7000 pages (using 50 different queries) with the corresponding bar chart below; the following two images show what happens after the LINEORDER table was repeatedly queried. The third snapshot displays 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 \((300 \times 120)\) 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 recently accessed data. 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 chapter, but note that when a query accesses 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.
### 3.8.3 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.

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

Table 3.7: Data reconstructed from a damaged disk image.

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 3.7 summarizes the 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, running a query in PostgreSQL after overwriting page metadata caused the following error:

> 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 the following error:

> ERROR: invalid memory alloc request size 2037542769.
3.8.4 Experiment 4: The lifetime of deleted data

In this experiment, we test a DBMS to see when a deleted value is overwritten, rather than just marked as 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 $T_0$, the phone value is present on disk in three different pages (in the table, the index and the MV). Table 3.8 shows the timeline of all three structures on-disk (HDD) and in-memory (RAM) – a $\square$ symbol means that the phone number can also still be returned by a SQL query. Both $\checkmark$ and $\times$ symbols mean that the value is inaccessible with SQL but can be reconstructed with page carving. The $\checkmark$ symbol means the value was marked as active, and the $\times$ symbol means the value was marked as deleted.

<table>
<thead>
<tr>
<th>Event</th>
<th>Table</th>
<th>Index</th>
<th>MV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HDD</td>
<td>RAM</td>
<td>HDD</td>
</tr>
<tr>
<td>$T_0$</td>
<td>$\square$</td>
<td></td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>$\checkmark$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$\checkmark$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$\checkmark$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$T_4$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$T_5$</td>
<td>$\times$</td>
<td></td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$T_6$</td>
<td>$\times$</td>
<td></td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$T_7$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.8: A timeline for the true deletion of a deleted phone value.

$T_1$ A phone row is deleted (including a COMMIT) by a user – this causes an index page with the phone (index values are not be marked deleted) and a table page with the phone marked as deleted to be cached in RAM.

$T_2$ User queries the MV causing the phone page to be cached in RAM.

$T_3$ 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).

$T_4$ 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.

$T_5$ A long series of queries is executed during which customer table is not accessed, evicting the table page with phone entry from RAM.
T_6_ The index is rebuilt and flushed from RAM.

T_7_ The table is rebuilt and flushed from RAM.

Thus the deleted value is overwritten 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.

### 3.8.5 Experiment 5: 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.

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 (Oracle default threshold is 39%). For both Oracle and SQL Server, we started with two different tables, each with 20K random sized rows. Both databases used a page size of 8KB, 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 T_{1\_rand}, 1000 deleted rows were randomly distributed across the page storage, while in table T_{2\_cont} 1000 deleted rows were contiguous (i.e., delete all rows from just a few pages).

<table>
<thead>
<tr>
<th>Action</th>
<th>Oracle T_{1_rand}</th>
<th>Oracle T_{2_cont}</th>
<th>SQL Server T_{1_rand}</th>
<th>SQL Server T_{2_cont}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete 1K Rows</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Insert 1K Rows</td>
<td>1000</td>
<td>8</td>
<td>416</td>
<td>354</td>
</tr>
<tr>
<td>Insert 1K Rows</td>
<td>1000</td>
<td>8</td>
<td>394</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3.9: Number of deleted rows recovered by DBCarver.
As Table 3.9 demonstrates, before new inserts come in, all of the deleted rows can be recovered by DBCarver. None of the deleted rows for T1rand 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 T2cont 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 T2cont 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 T1rand and T2cont first 1000 new inserts overwrote 60% to 65% of de-allocated rows (due to compaction applied by SQL Server). For the second 1000 inserts, T2cont replaced most of the deleted rows because they are contiguous and easy to overwrite. For T1rand, only 20 additional rows were displaced by the second batch of 1000 inserts because remaining T1rand are the smallest surviving rows that are difficult to overwrite.

Figure 3.6: An example for row insertion behavior in SQL Server.

Figure 3.6 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 Figure 3.6 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 page, leaving (Supplier1, Bob) intact with a deletion mark.

T1rand 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.

3.8.6 Experiment 6: Recovering Pre-Update Rows

In this experiment we demonstrate that for a typical workload of \textit{UPDATE}s we can recover many old rows, although fewer (5\%-10\%) compared to \textit{DELETE}ed row recovery in Experiment 1. Fewer old rows can be recovered because while updating values to a larger value results in \textit{DELETE}+\textit{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 20K 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 \textit{T1\textsubscript{rand}} were distributed across the table storage at random, while 1000 updates in \textit{T2\textsubscript{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 3.10 shows, for \textit{T1\textsubscript{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\% to 12\% of old records remained recoverable due to the way DB2 manages page storage. For \textit{T2\textsubscript{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 \textit{T1\textsubscript{rand}} in PostgreSQL, we recovered 137 values after the first 1000 updates and 92 records after the second 1000 updates. For \textit{T2\textsubscript{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 3.10 \textit{only} include fully recoverable rows, ignoring some partial old values that can be recovered as well (e.g., \textit{b} example in Figure 3.6).

<table>
<thead>
<tr>
<th></th>
<th>DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upd. Rows</td>
<td>\textit{T1\textsubscript{rand}}</td>
<td>\textit{T2\textsubscript{cont}}</td>
</tr>
<tr>
<td>1000</td>
<td>121</td>
<td>6</td>
</tr>
<tr>
<td>2000</td>
<td>125</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.10: Number of updated rows recovered.
### Table 3.11: Life cycle of deleted and updated values in a SQL Server index.

<table>
<thead>
<tr>
<th>Action</th>
<th>Index (Pg)</th>
<th>Unallocated (Pg)</th>
<th>America</th>
<th>Asia</th>
<th>Europe</th>
<th>Camelot</th>
<th>Atlantis</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>115</td>
<td>0</td>
<td>5992</td>
<td>6051</td>
<td>5937</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_1$</td>
<td>113</td>
<td>2</td>
<td>5992</td>
<td>6051</td>
<td>5937</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_2$</td>
<td>116</td>
<td>5</td>
<td>5992</td>
<td>6051</td>
<td>5937</td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>$T_3$</td>
<td>96</td>
<td>25</td>
<td>5992</td>
<td>6051</td>
<td>5937</td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>$T_4$</td>
<td>101</td>
<td>43</td>
<td>4692</td>
<td>5993</td>
<td>5937</td>
<td>1000</td>
<td>6167</td>
</tr>
<tr>
<td>$T_5$</td>
<td>135</td>
<td>71</td>
<td>4687</td>
<td>1080</td>
<td>1419</td>
<td>1000</td>
<td>6167</td>
</tr>
</tbody>
</table>

#### 3.8.7 Experiment 7: Recovering Indexed Values

This experiment demonstrates that DBCarver can recover thousands of old deleted or pre-update values (both from active and discarded index pages) from an index structure in a database. We used SQL Server and an index on `region` column for SSBM Scale1 (=30K rows) `customer` table—in general, indexes behave similarly across all DBMSes.

`Region` column has 5 distinct values, including ‘AMERICA’, ‘ASIA’, and ‘EUROPE’, with roughly 6K records for each ($6K \times 5 = 30K$). Table 3.11 summarizes recovered value counts—each time a count changes, the cell in Table 3.11 is highlighted with gray. We note that additional duplicate values were recovered based on the behavior described in Section 3.7.1, 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 16K 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 7 steps:

- $T_0$ Initial
- $T_1$ Delete 1K America Rows
$T_2$ Update 1K America $\rightarrow$ Camelot

$T_3$ Delete all Asia Rows

$T_4$ Update all Europe $\rightarrow$ Atlantis

$T_5$ Update remaining 16K Rows

### 3.8.8 Experiment 8: 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.

### 3.8.9 Experiment 9: 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 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 20K random sized rows and then deleted 1000 rows from the underlying tables. As in previous experiments, for $M_{1\text{rand}}$, 1000 deletes are applied to random storage locations in the table and for $M_{2\text{cont}}$ table deletes are applies in a contiguous fashion. Table 3.12 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 $M_{1\text{rand}}$. 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
Table 3.12: The number of deleted rows and duplicate active rows recovered from disk storage after MV refresh in Oracle.

<table>
<thead>
<tr>
<th>Row Type</th>
<th>Before Refresh</th>
<th>After Refresh</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$M_{\text{1 _rand}}$</td>
<td>$M_{\text{2 _cont}}$</td>
</tr>
<tr>
<td>Deleted</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Duplicated</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 $M_{\text{2 \_cont}}$, we found 60 deleted values and recovered 1111 distinct active rows from de-allocated storage. Similar recovery rates for $M_{\text{1 \_rand}}$ and $M_{\text{2 \_cont}}$ were as expected, because rows are being deleted from the original table, not from the MV that is reconstructed by DBCarver.

3.8.10 Experiment 10: 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 20K random sized rows and then deleted 1000 rows. 1000 rows deleted in $T_{\text{1 \_rand}}$ were distributed across the page storage and 1000 rows deleted in $T_{\text{2 \_cont}}$ were stored contiguously. Table 3.13 shows the number of recovered deleted rows and duplicated active rows. After a refresh of $T_{\text{1 \_rand}}$, 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 $T_{\text{2 \_cont}}$, we instead found 854 deleted rows and 182 duplicates of active rows on disk.
Table 3.13: The number of deleted rows and duplicate active rows recovered after a table rebuild in PostgreSQL.

<table>
<thead>
<tr>
<th>Row Type</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1_{rand}</td>
<td>T2_{cont}</td>
</tr>
<tr>
<td>Deleted</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Duplicated</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.7: An example of how PostgreSQL reorganizes pages with sparse or dense deletes during table rebuild.

Figure 3.7 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 Figure 3.7, for sparsely deleted rows before rebuild, Row2 has been marked as deleted in the row metadata. As the first page in Figure 3.7 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 Figure 3.7 (3^{rd} 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 sparse case, but marked-asdeleted rows are preserved on the page (row directory NULLs are unrelated to deletion).
3.9 Conclusion and Future Work

We presented a forensic tool, DBCarver, 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.

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 \texttt{DELETE} and \texttt{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).

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.
Chapter 4

DBCarver

This chapter gives a detailed description of the tool, DBCarver, and algorithms to perform page carving, which was introduced in the previous chapter.

4.1 Initial Setup

When support for a new DBMS or functionality for a new DBMS is necessary, the DBMS version in question should be installed on a trusted system. Typically, the default settings should be used. However, if something is known about the sample, such as page size, these parameters should be set to match the sample. The exact DBMS version may not be necessary. For here on, when we refer to the DBMS, we mean the DBMS on the trusted system.

4.2 Parameter Collection

This chapter will discuss how to deconstruct the internal database page format for any RDBMS. A page stores both the user data and metadata, which describes the user data and page. By deconstructing the page format, the parts within the page can be described with a set of parameters that generalize across all DBMSes. We have divided deconstruction into four main categories based on the high-level page format used by all RDBMSes: page header (Section 4.2.1), row directory (Section 4.2.2), and row data (Section 4.2.3), and datatype decoding (Section 4.2.4).

4.2.1 Page Header

The page header stores general information about the page and its data. User data is not stored in the page header. This section discusses how to deconstruct page header meta-
data that is commonly used by RDBMSes, which includes general page identifier (Section 4.2.1.1), structure identifier (Section 4.2.1.2), unique page identifier (Section 4.2.1.3), and record count (Section 4.2.1.4). Other metadata may exist in a page header, such as free space pointer, checksum, or logical timestamp, but we believe this metadata has little forensic significance or is not commonly used by RDBMSes. Experience has demonstrated that all page header metadata is located within the first 2% of a page. Therefore, when a page is evaluated in this section, we only consider the first 2% of bytes for the page.

Page header metadata is generally deconstructed by:

1. Locate the metadata typically using similarities or differences between pages.

2. Record the address of the metadata as a parameter.

### 4.2.1.1 General Page Identifier

The general page identifier is a sequence of non-NULL bytes shared by all pages in a particular DBMS version. The general page identifier is used by our parser to initially search for pages within a file. We make two assumptions about the general page identifier: 1) it must be between two and three bytes in length and 2) at least one byte must be non-NULL, meaning a decimal value not equal to zero. In addition to extracting the general page identifier as a parameter, the address of the general page identifier is also determined for use as a parameter. These parameters must not be non-NULL for parsing to be performed.

The following steps summarize how we determine the general page identifier and general page identifier address parameters:

1. Byte commonalities are found for all data pages containing synthetic data. Byte commonalities are found as follows:

   (a) A dictionary is created for each page. The address of a byte within the page and the decimal value of a byte are stored as the key-value pairs.

   (b) All dictionaries are then compared and the results are written to a new dictionary, called the results dictionary. If the values across all dictionaries match for a particular key, the byte value and its address are recorded in the results dictionary; otherwise, the address and the value -1 is added to the results dictionary.

2. Using the results dictionary, all sequences of two to three contiguous bytes are considered.
3. The longest sequence with the minimum number of NULL characters is returned as the general page identifier and the address of this sequence from beginning of the page is returned as the general page identifier address. In case of a tie, the first sequence is chosen.

4.2.1.2 Structure Identifier

The structure identifier is a 16-bit or 32-bit integer that is shared by all pages that comprise an individual structure (e.g., table or index); this identifier is unique across structures. The structure identifier is ascertained by the structure identifier address and structure identifier size parameters. These parameters can be NULL. Note that most DBMSes store a structure identifier in the page header. However, if a structure identifier is not stored in the page header, one may be stored with each record in the row data of the page. In this instance, the structure identifier would be considered a piece of row data metadata.

In determining where the structure identifier is located, only the low byte of the identifier is considered. Although more rigorous comparisons could be utilized, such a process would require creating at least 256 structures and comparing them. Experience has shown that such a robust process for this identification is unnecessary. The following steps summarize how we determine the structure identifier address and structure identifier size:

1. Byte commonalities are found using the process described in Section 4.2.1.1; however, only pages from the same table are used to determine candidate identifiers.

2. The first byte of common sequences are compared across structures. If they differ and either that byte or the next one is non-null, then the byte position is returned.

3. When sequences of bytes across all pages are found to meet the conditions, the address of these sequences is returned as structure identifier address, and the number of high bytes plus the low byte is returned as structure identifier size.

4.2.1.3 Unique Page Identifier

The unique page identifier is a 32-bit integer that is unique for each page across the entire database or within a file. The unique page identifier is located using the unique page identifier address parameter, which is the address of the unique page identifier. This parameter can be NULL in the case there is no unique page identifier.

The unique page identifier address is determined as follows. Byte differences are found across all data pages that comprise a single synthetic table (since the unique page
identifier need only be unique per structure). To reduce runtime, we currently consider differences for the first three bytes and assume the fourth byte. A fourth byte can be confirmed, but $256^3$ (or 16M) pages are needed. Currently then, each three byte sequence is tested for uniqueness among all of the data pages. Additionally, the identifier must increase as page address within the file increases. When a sequence of bytes across all pages is found to meet these conditions, the address of these sequences is returned as unique page identifier address.

4.2.1.4 Record Count

The record count is a 16-bit number that represents the number of active records stored within the page. The record count is located using the record count address. The record count address is determined using the following process. Each page containing synthetic table data is searched for records. In particular, the synthetic data contains unique string data that conforms to a specific pattern, e.g., ‘Curly0001’, ‘Curly0002’, .... The count of these strings indicates the number of records on that page. The page header is then searched for a 16-bit number representing that count. This process is repeated on several table data pages in order to confirm that the location of the record count is consistent across pages, thereby confirming that the actual position of the record count has been located.

4.2.2 Row Directory

4.2.2.1 Assumptions

The row directory stores pointers, in the form of byte offsets, to records within the page. The row directory may be dense (i.e., a pointer to each record) or sparse (i.e., a pointer for a group of records). We assume, based on our earlier experience inspecting DBMS files, that a pointer is a 16-bit integer and that a sparse row directory may have a maximum of 10 records. Along with pointers, the row directory may also store the size of each record. Only deconstruction of pointers is currently implemented since record size is not commonly observed. Again, based on our experience, we assume that no more than 8 bytes may exist between pointers.

4.2.2.2 Pointer Order Sequence

Pointer order sequence refers to the placement of pointers as they are added to the row directory. Notably, the order of the records appended to the row data is the inverse of the pointer order. For example, if records are added to the row data from bottom-to-top of
the page, then row directory pointers are appended from top-to-bottom on the page. The pointer addition sequence is represented with the Boolean order sequence parameter, with top-to-bottom being True and bottom-to-top being False. This parameter must be non-NULL.

The following steps summarize how we determine the order sequence:

1. A random sample of several pages containing synthetic data are arbitrarily selected.

2. The synthetic data contains strings that indicate the order in which they were added to the database through Insert statements. For example, ‘Curly0001’ was inserted first, ‘Curly0002’ was inserted second, etc.

3. If the address of the record inserted first is greater than the record that was inserted second, then we conclude that records are inserted from bottom-to-top; otherwise, records are inserted from top-to-bottom. This conclusion is confirmed with the remaining records in the page and the other sample pages.

4. If records are inserted bottom-to-top, then row directory pointers are added from top-to-bottom. In this case, order sequence is assigned True. If records are inserted top-to-bottom, then row directory pointers are added from bottom-to-top. In this case, order sequence is assigned False.

4.2.2.3 Row Directory Location

The row directory location refers to the address of the pointer for the first inserted record within the page. Since a row directory may be dense or sparse, the number of records referenced by each pointer is also necessary; this quantity is referred to as the slot size. The row directory location, pointer distance (number of bytes between pointers), and slot size are represented with the row directory start, pointer distance, and slot size parameters. These parameters must be non-NULL.

The following steps summarize how we determine the row directory start, pointer distance, and slot size parameters:

1. A random sample page containing synthetic data is selected.

2. Each page stores records that each have a unique string containing numeric data that indicates the order in which the record was inserted. For example, ‘Curly0001’ was inserted first, ‘Curly0002’ was inserted second, etc. We collect the distances between each unique string.
3. The page is scanned for a sequence of bytes whose values differ by the distances between unique strings. Each such sequence would constitute a candidate for the sequence of pointers; therefore, the distance between them must be no more than eight bytes, as mentioned above.

4. Candidate sequences are selected by varying independently both the pointer distance and the slot size. Initially, a slot size of one (or a dense row directory) is first considered.

5. If the sequence is not found, then the slot size is incremented by one until a slot size of 10 is reached. When the slot size is increased the distances between unique strings are combined to reflect this change.

6. If a sequence is found, then we conclude that we have located the low byte of each two-byte row directory pointer. The start of this sequence is returned as row directory start, the distance between each low byte is returned as pointer distance, and the slot size is returned as slot size. The high byte for the pointers will be considered in Section 4.2.2.4.

4.2.2.4 Pointer Deconstruction

After the parameters in Section 4.2.2.3 are collected, the position of the high byte must be determined. The high byte either immediately precedes or follows the low value byte. This is represented with the high value position parameter. This parameter is determined with the following steps:

1. An arbitrary page containing synthetic data is selected.

2. Each page stores records that each have a unique string containing numeric data that indicates the order in which the record was inserted. For example, ‘Curly0001’ was inserted first, ‘Curly0002’ was inserted second, etc. We collect the distances between each unique string.

3. Starting from the row directory start, we consider each 16-bit integer using the preceding byte as the high value byte.

4. If the difference between each 16-bit integer equals each distance between records, then high value position is returned as -1.

5. If there is a discrepancy between the 16-bit integers and the record distances, then high value position is returned as 1.
Additionally, the 16-bit integer used for a row directory pointer may not represent the explicit record address. To decode the explicit address, two decoding constants are required: \( C_x \) and \( C_y \). These constants are used in the following equation:

\[
RecordAddress = (Y - C_y) \times 256 + X + C_x
\] (4.1)

where \( Y \) is the decimal value of the high value byte and \( X \) is the decimal value of the low value byte. \( C_x \) and \( C_y \) are represented with the parameters \( C_x \) and \( C_y \). These constants are determined with the following steps:

1. An arbitrary page containing synthetic data is selected.
2. Let \( P_1 \) be the integer value of a pointer from the row directory of that page.
3. Let \( A_1 \) be the actual byte address of the record that corresponds to \( P_1 \). The actual address includes additional metadata in the row data; it is not the address of the known string.
4. The beginning of the additional metadata is determined using the order sequence parameter, which was determined earlier. If order sequence is True we know that the first record inserted into the page directly follows the second record in the page. Therefore, the metadata for the first record begins directly after the second record ends, giving us \( A_1 \). If order sequence is False we know that the first record was inserted at the beginning of the row data. Therefore, the metadata for the first record is the first thing stored in the row data, giving us \( A_1 \).
5. \( C_y \) is then returned as \((P_1 - A_1)/256\), where integer division is used.
6. \( C_x \) is then returned as \((A_1 - (Y - C_y) \times 256) - X\).

### 4.2.3 Row Data

As mentioned earlier, each row contains metadata and user data; the metadata describes the user data and is discussed below.

#### 4.2.3.1 Row Delimiter

The row delimiter is a single byte that marks the beginning of a row. Row directory pointers typically reference the address of the row delimiter. Moreover, the row delimiter may be used to mark a row as deleted. The row delimiter is represented by the row delimiter parameter, which we determine by the following steps:
1. A large quantity of pages containing synthetic data is selected, specifically 257 or more pages from each of two different tables, each having different schema. In practice, we typically use thousands of pages of synthetic data.

2. If the row data is stored bottom-to-top, then we define $Row_1$ as the bytes from the end of the last column of $Row_2$ to the end of the last column of $Row_1$.

3. Conversely, if the row data is stored top-to-bottom, then we define $Row_1$ as the bytes from the first byte of the row data to the end of the last column of $Row_1$.

4. If the first byte for every row is common and non-NULL across all rows for all of the synthetic data, then that byte value is returned as row delimiter. Otherwise, row delimiter is NULL.

4.2.3.2 Row Identifier

The row identifier is an internal pseudo-column that the DBMS uses to uniquely identify rows; it is typically used within an index to point to a row. The row identifier typically requires 1 - 4 bytes of storage. The presence of a row identifier in the row data is represented with the Boolean row identifier exists parameter. The following steps summarize how we determine the row identifier exists parameter:

1. We define the row header as the storage between the beginning of the row (i.e., the row directory pointer) and the first column of user data.

2. For a single-column table containing all string values of the same size, we consider the row headers.

3. If the the row headers are exactly the same (byte-for-byte) for all records, row identifier exists is set to False. Otherwise, row identifier exists is set to True.

4. To confirm the results, this process is repeated for a table containing two columns of string where the all of the values are the same size.

From our experience, the row identifier is always at a static position, which is described with the row identifier offset parameter. The following steps summarize how we determine this parameter:

1. We define the row header as the storage between the beginning of the row (i.e., the row directory pointer) and the first column of user data.
2. For a single-column table containing all values of the same size, we consider the row headers.

3. The offset of the bytes that change between all row headers are recorded.

4. We assume that the byte with the highest offset is the low byte. **Row identifier offset** is returned as this position.

The row identifier may use a static or variable number of bytes, which is represented with the **row identifier static size** parameter. The following steps summarize how we determine the **row identifier static size** parameter if and only if **row identifier exists** is True:

1. We define the **row header** as the storage between the beginning of the row (i.e., the row directory pointer) and the first column of user data.

2. For a single-column table containing all values of the same size, we consider the row headers.

3. If the the row headers are exactly the same size (i.e., the number of bytes) for all records, **row identifier static size** is set to True. Otherwise, **row identifier static size** is set to False.

4. To confirm the results, this process is repeated for a two-column table where the all of the values are the same size.

If **row identifier static size** is True, we represent that size of the row identifier with the **row identifier size** parameter. The following steps summarize how we determine this parameter:

1. We define the **row header** as the storage between the beginning of the row (i.e., the row directory pointer) and the first column of user data.

2. For a single-column table containing all values of the same size, we consider the row headers.

3. **Row identifier size** is set to the number of the bytes that change between all row headers.

4. To confirm the results, this process is repeated for a two-column table where the all of the values are the same size.

If the row identifier is variable in size, its value for each row by the parsing code, which is described later.
4.2.3.3 Column Count

The column count represents the number of columns in the row. The presence of the column count is represented with the Boolean column count exists parameter. The row identifier, if included in the row, may be treated as a column, although it is technically a pseudo-column. In such cases, the column count would be the total number of user data columns plus one. The inclusion of the row identifier pseudo-column in the column count is represented with the column count includes row identifier parameter. The following steps summarize how we determine the column count exists and column count includes row identifier parameters:

1. We define the row header as the storage between the beginning of the row (i.e., the row directory pointer) and the first column of user data.

2. We consider synthetic tables with 1, 2, and 3 columns.

3. For the 1 column table, the row header is scanned for a decimal value 1 at the same position across all row headers. If such a commonality is found, the 2 column table is scanned for the same commonality of decimal value 2. If the commonality is found again, the 3 column table is scanned for the commonality of decimal value 3. If this final commonality is found, column count exists is returned as True and column count includes row identifier is returned as False.

4. If a column count is not identified, the previous step is repeated except the decimal value 2 is used for the 1 column table, 3 is used for the 2 column table, and 4 is used for the 3 column table. If these commonalities are found, column count exists is returned as True and column count includes row identifier is returned as True.

5. If a column count is still not identified, column count exists and column count includes row identifier are returned as False.

The location of the column count may be indicated in a few different ways. The necessary parameters include column count fixed offset, column count delimiter, column count delimiter offset, and column count pointer offset. The following steps summarize how we determine the column count fixed offset parameter:

1. We consider a synthetic table with 1 column.

2. If the decimal value 1 (or 2 if column count includes row identifier is True) is at the same offset within the row header, column count fixed offset is set to this offset. Otherwise, column count fixed offset is set to NULL.
3. These results are confirmed with the 2 and 3 column synthetic tables.

   If `column count fixed offset` is NULL, tests for the column count delimiter are performed. The following steps summarize how we determine the `column count delimiter` and `column count delimiter offset` parameters:

   1. Based on our experience, we assume the column count delimiter is a sequence of two or three non-NULL bytes shared by all row headers.
   2. Byte commonalities are found for all row headers in a synthetic table with 1 column.
   3. If a sequence of 3 contiguous bytes are found in all row headers and the distance between this sequence and the decimal value 1 (or 2 if `column count includes row identifier` is True) is the same for each row, `column count delimiter` is set to the sequence of bytes and `column count delimiter offset` is set to the distance between the sequence of bytes and the decimal value 1.
   4. If a sequence of 3 bytes is not found, a sequence of 2 bytes is considered.
   5. If a 2 byte sequence is not found, `column count delimiter` and `column count delimiter offset` are set to NULL.
   6. These results are confirmed with the 2 and 3 column synthetic tables.

   If `column count delimiter` is NULL, tests for a column count pointer are performed. The column count pointer is a pointer to the column count stored in the row header. The following steps summarize how we determine the `column count pointer offset` parameter:

   1. For all rows in 1, 2, and 3 column synthetic tables, the distance between the beginning of the row and the decimal value 1, 2, and 3 (or 2, 3, and 4 if `column count includes row identifier` is True) are collected.
   2. Each row header is scanned for a value that equals the respective distance found in the previous step.
   3. If a matching value is found at the same offset within each row, `column count pointer offset` is set to this offset. Otherwise, `column count pointer offset` is set to NULL.
4.2.3.4 Column Sizes

Column sizes refer to the size of strings. This section focuses on locating the column sizes to learn the general layout of the row, rather than decoding these values, which is discussed in Section 4.2.4. Column sizes are either adjacent to each value or stored in the row header.

We first consider the case when column sizes could be stored adjacent to each value. This information is represented with the Column Sizes are Stored with Raw Data parameter. The following steps summarize how we determine this parameter:

1. For a single-column table containing all values of the same size, we consider the byte that precedes our known synthetic value (one per row).

2. If this byte is the same value for all rows, and the decimal value is greater than or equal to the actual length of the value, we continue; otherwise, Column Sizes are Stored with Raw Data is returned as False.

3. For a two-column table containing all values of the same size, although a larger size than in the single-column table, we consider the byte that precedes our known synthetic values (two per row).

4. If these bytes are the same value for all rows, and the decimal values are greater than or equal to the actual length of the values and greater than values found for the single-column table, Column Sizes are Stored with Raw Data is returned as True; otherwise, Column Sizes are Stored with Raw Data is returned as False.

If Column Sizes are Stored with Raw Data is False, we next consider the case when column sizes are stored in the header. This information is represented with the Column Sizes in Header parameter. When this parameter is True, the Column Sizes Offset parameter is also needed to locate the column sizes. The following steps summarize how we determine these parameters:

1. We define the row header as the storage between the beginning of the row (i.e., the row directory pointer) and the first column of user data.

2. For a single-column table containing all values of the same size, the row header is scanned for a value that is equal to or greater than the length of our known synthetic value.

3. If a common byte is found for all rows and at the same location, Column Sizes in Header is set to True and Column Sizes Offset is set to the location of the common
byte; otherwise, **Column Sizes in Header** is returned as *False* and **Column Sizes Offset** is returned as *NULL*.

4. For a two-column table containing values of the same size (but different size than the single-column table), the row header is scanned for a value that is equal to or greater than the length of our known synthetic values.

5. If two common bytes are found for all rows and at the same location as the single-column table, **Column Sizes in Header** is returned as *True* and **Column Sizes Offset** is returned as the location of the common bytes; otherwise, **Column Sizes in Header** is returned as *False* and **Column Sizes Offset** is returned as *NULL*.

If **Column Sizes in Header** is *False*, we finally consider the case when column sizes are stored at a floating location within the header. In such a case, column sizes are stored before the position referenced by the row directory pointer. This information is represented with the **Column Sizes at Floating Location** parameter. The following steps summarize how we determine this parameter:

1. For a single-column table containing all values of the same size, the byte before the row directory pointer address is scanned for a value that is equal to or greater than the length of our known synthetic value.

2. If a common byte is found for all rows, **Column Sizes at Floating Location** is set to *True*; otherwise, **Column Sizes at Floating Location** is returned as *False*.

3. For a two-column table containing values of the same size (but different size than the single-column table), the row header is scanned for a value that is equal to or greater than the length of our known synthetic values.

4. If two common bytes are found for all rows, **Column Sizes at Floating Location** is returned as *True*; otherwise, **Column Sizes at Floating Location** is returned as *False*.

### 4.2.3.5 Column Directory

The column directory stores pointers to each value within the row. The column directory is either at a fixed offset from the beginning of the row or a fixed offset from the column count. If column sizes were previously found, we assume that a column directory does not exist.
We first consider the case when the column directory is at a fixed position from the beginning of the row, which is represented with the **column directory at fixed offset** parameter. While determining the location of the column directory, we also determine the distance between pointers, which is represented with the **column directory pointer size** parameter. The following steps summarize how we determine these parameters:

1. We first assume that the distance between the low bytes of two consecutive column directory pointers will not be more than 4 bytes apart.

2. We define the *row header* as the storage between the beginning of the row (i.e., the row directory pointer) and the first column of user data.

3. For a two-column table containing all values of the same size, we scan the row headers for the low bytes of the two column pointers.

4. These column pointers will have a difference of the length of the first column value, i.e., the low byte of the first pointer plus the length of the first column equals the low byte of the second pointer.

5. While scanning the row header for these bytes, we consider that the distance between the low bytes could be 1, 2, 3, or 4 bytes.

6. If a correct sequence of bytes is found at the same position for all rows, **column directory at fixed offset** is set to the location of the first low byte, and **column directory pointer size** is set to the distance between the low bytes when the sequence was found.

It is possible that the position of the column directory may be shifted by other metadata. From our experience, we have found that a shift in the column directory matches a shift with the column count. Therefore, we believe it is more accurate to locate the column directory using the column count (if the column count is available). The **column directory column count offset** parameter describes the distance of the column directory from the column count. The following steps summarize how we determine this parameter:

1. If a column count exists, **column directory column count offset** is set to **column directory at fixed offset** minus the address of the column count; otherwise, **column directory column count offset** is returned as NULL.

2. This result is confirmed for all rows.
4.2.3.6 Raw Data

The raw data refers to the user data. The raw data can be located at a fixed offset within the row, following a delimiter, following the column directory, or following column sizes that may be stored in the row header.

We first consider the case when the raw data could be located at a fixed offset, which is described with the Raw Data Fixed Offset parameter. The following steps summarize how we determine this parameter:

1. We define the row header as the storage between the beginning of the row (i.e., the row directory pointer) and the first column of user data.

2. For a single-column table and a two-column table, we consider the row headers.

3. If the row headers are exactly the same size (i.e., the number of bytes) for all records in both tables, Raw Data Fixed Offset is set to the size of the row headers. Otherwise, Raw Data Fixed Offset is set to NULL.

If Raw Data Fixed Offset is NULL, we consider the case when a delimiter may indicate where the raw data is located, which is described by the Raw Data Delimiter and Raw Data Delimiter Offset parameters. The following steps summarize how we determine these parameters:

1. Based on our experience, we assume the raw data delimiter is a sequence of two or three non-NUL bytes shared by all row headers.

2. Byte commonalities are found for all row headers in a single-column table and a two-column table.

3. If the same sequence of 3 contiguous bytes is found in all row headers, Raw Data Delimiter is set to that sequence of bytes. Raw Data Delimiter Offset is set to the distance between the sequence of bytes and the first byte of the first known synthetic value.

4. If a sequence of 3 bytes is not found, a sequence of 2 bytes is considered.

5. If a 2 byte sequence is not found, Raw Data Delimiter and Raw Data Delimiter Offset are set to NULL.
If **Raw Data Delimiter** is **NULL** and if a column directory is known to exist, we consider the case when the raw data follows the column directory, which is described with the **Raw Data Succeeds Column Directory Offset** parameter. The following steps summarize how we determine this parameter:

1. The column directory is located in the row headers for all rows in a single-column table and a two-column table.

2. The distance between the row directory and the first known synthetic value are collected for all rows in both tables.

3. If the distance is the same for all rows in the single-column table, the distance is same for all rows in the two-column table, and the distance in the single-column table plus **column directory pointer size** equals the distance in the two-column table, **Raw Data Succeeds Column Directory Offset** is returned as the distance in the single-column table minus **column directory pointer size**. Otherwise, **Raw Data Succeeds Column Directory Offset** is returned as **NULL**.

If **Raw Data Succeeds Column Directory Offset** is **NULL** and the column sizes are stored in row data headers, we consider the case when the raw data follows the column sizes, which is described with the **Raw Data Succeeds Column Sizes Offset** parameter. The following steps summarize how we determine this parameter:

1. The column sizes are located in the row headers for all rows in a single-column table and a two-column table.

2. The distance between the column sizes and the first known synthetic value are collected for all rows in both tables.

3. If the distance is the same for all rows in the single-column table, the distance is same for all rows in the two-column table, and the distance in the single-column table plus 1 equals the distance in the two-column table, **Raw Data Succeeds Column Sizes Offset** is returned as the distance in the single-column table minus 1. Otherwise, **Raw Data Succeeds Column Sizes Offset** is returned as **NULL**.

### 4.2.4 Data Encoding

Data encoding parameters describe how a given DBMS stores values in the row data.
4.2.4.1 Strings

Column sizes for strings may be stored in either the row header or adjacent to the string data; however, these values may not represent the actual string sizes. In practice, some DBMSes compute and store column sizes that differ from the actual column sizes. To decode the actual sizes from the columns sizes as stored, two decoding constants are required: \( C_x \) and \( C_y \). These constants are used in the following equation:

\[
ColumnSize_{\text{stored}} = C_x \times ColumnSize_{\text{actual}} + C_y
\] (4.2)

Once determined, these constants are used by the parser to determine actual column sizes. These constants are determined with the following steps:

1. We consider synthetic two tables, \( T_1 \) and \( T_2 \), with variable length strings.

2. For the string columns from any given pair of rows from each table,

\[
C_x = \frac{T_2.ColumnSize_{\text{stored}} - T_1.ColumnSize_{\text{stored}}}{T_2.ColumnSize_{\text{actual}} - T_1.ColumnSize_{\text{actual}}}
\] (4.3)

3. For any given column from either table,

\[
C_y = ColumnSize_{\text{stored}} - (C_x \times ColumnSize_{\text{actual}})
\] (4.4)

Strings may sometimes be padded with extra NULL character to achieve 4-byte alignment, represented with the Boolean 4-byte alignment parameter.

1. We consider a table with three columns: a variable-length string, an integer, and another fixed-length string.

2. If the distance between the variable length string and the integer is constant, then 4-byte alignment is returned as False.

3. If the distance between the variable length string and the integer is not constant, the variable bytes between the string and the integer are NULL characters, and the number of NULL characters corresponds to 4-byte alignment; then 4-byte alignment is returned as True.
4.2.4.2 Integers

We first determine where integers are stored within the rows. Integers can either be stored with the other columns (i.e., strings) or in the row data header. We represent this information with the Integer Location parameter. The following steps describe how we determine this parameter:

1. We consider a table with three columns: a fixed-length string, an integer, and another fixed-length string. In our synthetic data, we populate the integer with a value in a linear series, starting from 1.

2. If the space between the two fixed-length strings contains the same value for all rows or no space exists, we assume that integers are stored in the row data header, and we set Integer Location equal to ‘header’. Otherwise, we assume that integers are stored with the string column data, and we set Integer Location equal to ‘strings’.

When Integer Location equals ‘strings’, some DBMSes write integer values at 4-byte aligned positions. Accordingly, they pad strings that immediately precede them with NULLs. We represent this information with the 4 Byte Integer Alignment parameter. The following steps describe how we determine this parameter:

1. We consider a table with three columns: a fixed-length string of 19 characters, an integer, and another fixed-length string. In our synthetic data, we populate the integer with a value in a linear series, starting from 1.

2. If the space between the two fixed-length strings is greater than 4 bytes and the previous string is padded with NULL bytes, 4 Byte Integer Alignment is returned as True. Otherwise, 4 Byte Integer Alignment is returned as False.

Integers are commonly stored as 32-bit numbers; the 32-bit Integer parameter represents this information. The following steps describe how we determine this parameter:

1. For this parameter to be True, four bytes must be used to represent the value and the following equation will yield the actual value:

\[ Value_{actual} = 256^0 \times Byte_1 + 256^1 \times Byte_2 + 256^2 \times Byte_3 + 256^3 \times Byte_4 \]  (4.5)

2. We consider a table with three columns: a fixed-length string, an integer, and another fixed-length string. In our synthetic data, we populate the integer with a value in a linear series, starting from 1.
3. We collect all of the bytes used to represent the integer. This is either the space between the two string columns or the space between the two row directory pointers. This is based on the previously determined Integer Location parameter.

4. If the space is 4 bytes for all rows and the above equation is true for all rows, 32-bit Integer is returned as True. Otherwise, 32-bit Integer is returned as False.

4.2.4.3 Other Datatypes

Currently, we do not detect parameters for data types other than strings and integers. We intend to continue working on parameter detection for other data types in the future.

4.3 Carving Pages

This chapter discusses page carving, a method to reconstruct the internal database page format that was generalized for all RDBMSes by the parameters previously defined in Chapter 4.2. We divided page carving into four main categories based on the high-level page format used by all RDBMSes: page header (Section 4.3.1), row directory (Section 4.3.2), row data (Section 4.3.3), and datatype decoding (Section 4.3.4).

Once the parameter collector generates parameter files, they are passed to the carver to reconstruct metadata and raw user data from digital evidence. For each provided parameter file, everything in this chapter is done for each evidence file passed to the carver.

4.3.1 Page Header

This section discusses how to carve the most common page header metadata, which includes general page identifier (Section 4.3.1.1), structure identifier (Section 4.3.1.2), unique page identifier (Section 4.3.1.3), and record count (Section 4.3.1.4). Other metadata may exist in a page header, such as free space pointer, checksum, or logical timestamp. Page header metadata is generally carved using the respective steps:

1. Locate a metadata field using an offset parameter.
2. Collect all metadata field bytes using a size parameter.
3. Decode the metadata field using either a specified equation or the default 32-bit little endian encoding.
4.3.1.1 General Page Identifier

To find a page header, the evidence file is scanned for the general page identifier parameter. When this sequence of bytes is found, the carver uses the address of the general page identifier to locate the start of a page and begins page carving. It is possible that a sequence of bytes matching the general page identifier parameter is found, but a DBMS page is not present. To eliminate such false-positives, we make assumptions throughout this chapter. If an assumptions is broken, page carving is terminated and NULL is returned for that page.

4.3.1.2 Structure Identifier

The following steps summarize how to carve the structure identifier. When structure identifier address and structure identifier size are not NULL:

1. Move to the page offset of the structure identifier address parameter.

2. The number of bytes (typically 2 or 4) are collected based on structure identifier size.

3. The value is calculated assuming a little-endian integer unless otherwise specified in the parameter file.

Assumption: The structure identifier integer value must be greater than 1. Otherwise, page reconstruction is terminated. This assumption is safe because a DBMS builds a collection of system catalog tables, which are assigned a structure identifier, upon instance creation. Since the structure identifier is a sequential integer, any structure created by the user is assigned a structure identifier greater than 1.

4.3.1.3 Unique Page Identifier

The unique page identifier is carved using the unique page identifier address parameter. This parameter can be NULL when there is no unique page identifier. The following steps summarize how to carve the unique page identifier. When unique page identifier address is not NULL:

1. Move to the page offset of the unique page identifier address.

2. The value is calculated assuming a 32-bit little-endian integer unless otherwise specified in the parameter file.
4.3.1.4 Record Count

The record count is carved using the record count address. The following steps summarize how to carve the record count. When record count address is not NULL:

1. Move to the page offset of the record count address.
2. The value is calculated assuming a 16-bit little-endian integer unless otherwise specified in the parameter file.

4.3.2 Row Directory

The row directory is carved with the following steps:

1. The carver moves to offset of the row directory start parameter.
2. For each $i^{th}$ iteration, the carver moves to pointer distance * $i$. This is the location of the pointer low byte. Note that pointer distance is a positive integer when order sequence is True and it is a negative integer when pointer distance is False.
3. Since endianness is not assumed, the high byte is retrieved using the high value position, which is either 1 or -1.
4. Using the $C_x$ and $C_y$ parameters, the pointer value is determined with the following equation where $Y$ is high byte and $X$ is the low byte.

$$
Pointer = (Y - C_y) * 256 + X + C_x
$$

(4.6)

5. These steps are repeated until an invalid pointer is found based on assumptions. When order sequence is True, we assume a pointer must be greater than pointer distance * $i$ and less than pointer distance * $i$. When order sequence is False, we assume a pointer must be less than pointer distance * $i$.

Note that other rules and assumptions can be customized for specific DBMS storage engines. For example, a pointer may be zero upon delete for some DBMSes. In such a case, valid pointers may follow this zero. Therefore, zero is not a reasonable break condition.
4.3.3 Row Data

Every row in the row data contains both metadata and user data. This section discusses how to carve the metadata and use it to reconstruct the user data.

For every row directory pointer, a record is located (except for the case of a sparse row directories). Figure 4.1 provides an overview of how record carving is performed. For each record, all of the metadata from the record header is reconstructed using the respective parameters from the parameter file. When the raw data is parsed, the data types are determined for every value (Section 4.3.4 provides a more detailed description for data type decoding). Once all row directory pointers are considered, any unallocated storage within the row data is scanned for bytes that resemble the structure of a record (i.e., deleted records).

![Figure 4.1: Record parsing overview.](chart)

4.3.3.1 Row Delimiter

If the row delimiter parameter is not NULL, then a row directory pointer must point to a byte(s) that equals the row delimiter parameter. Otherwise, the carving for that record is aborted and that row directory pointer is considered invalid.

4.3.3.2 Row Identifier

A row identifier is carved using the row identifier exists, row identifier offset, row identifier static size, and row identifier size parameters. The following steps summarize how to carve a row identifier when the row identifier exists parameter is True:

1. Move to the row offset of the row identifier offset parameter.

2. If row identifier static size is True, the row identifier size parameter is used to collect the appropriate number of bytes, and the unsigned integer is returned.

3. If row identifier static size is False, every byte is assumed to be signed. All bytes greater than or equal to decimal value 128 are collected, and the integer is returned.
4.3.3.3 Column Count

A column count is carved using the column count exists, column count includes row identifier, column count fixed offset, column count delimiter, column count delimiter offset, and column count pointer offset parameters. The following steps summarize how to carve a column count when the column count exists parameter is True:

1. If the column count delimiter parameter is not NULL, find the column count delimiter value, move column count fixed offset, and read the byte.

2. Else if the column count fixed offset parameter is not NULL, move to the row offset of column count fixed offset and read the byte.

3. Else if the column count pointer offset parameter is not NULL, move to offset column count pointer offset, decode the pointer as a 16-bit unsigned integer, move to the pointer location, and read the byte.

4. Finally, if column count includes row identifier is True, the byte read minus one is returned as the column count. Otherwise, the byte read is returned as the column count.

4.3.3.4 Column Directory vs. Column Sizes

Up until this point, the raw user data parsing is not dependent on the metadata parsed (i.e., the row delimiter, row identifier, and column count); although the column count metadata can be leveraged for more accurate raw user data parsing. The parser next decides between a either column directory or column size (format) to reconstruct since the raw user data is dependent on this information. If either the Column Directory at Fixed Offset or Column Directory Column Count Offset parameters are not NULL, then a column directory is parsed and column sizes are not considered. If both the Column Directory at Fixed Offset or Column Directory Column Count Offset parameters are NULL, then the parser moves to determine the format in which the column sizes are stored and a column directory is not considered.

Since the raw user data is dependent on the column directory or column size metadata, the raw data is often located and evaluated in parallel.
4.3.3.5 Column Directory

A column directory is reconstructed if either Column Directory at Fixed Offset or Column Directory Column Count Offset parameters are not NULL. To reconstruct the column directory, the column directory is located and then the pointers are parsed.

The following steps describe how the column directory is located:

1. If Column Directory at Fixed Offset is not NULL, then the parser moves to the offset of the Column Directory at Fixed Offset parameter.

2. Else if a column count was parsed (from Section 4.3.3.3 and Column Directory Column Count Offset is not NULL, the parser moves the number of bytes stored as Column Directory Column Count Offset from the column count location.

The following steps describe how the column directory pointers are parsed. Since integers are stored in the column directory with pointers, the parser determines if the each value is an integer or a pointer:

1. For each field in the column directory, a pointer is first assumed.

2. If the assumed pointer stores an unreasonable address (i.e., an address within the column directory, an address outside of the row, and the sequence of pointers does not ascend), an integer is assumed. The integer is reconstructed and returned.

3. Else if a reasonable pointer is determined, the user data value is evaluated for a string – string conditions are discussed in Section sec:datadecoding2. Since the size of the string is not provided, the carver considers all bytes up until the next pointer or up until the row end as part of the string.

4. If the user data value is determined to be a string, it is returned. Otherwise, unknown data type is assumed.

5. The breaking condition for iterating over fields in the column directory is either the column count (when present) or when the carver reaches the address of the first column directory pointer (or the end of the row in the case of all integers).

4.3.3.6 Column Sizes

Column sizes refers to the size of strings. Column sizes are carved using the following parameters: Column Sizes are Stored with Raw Data, Column Sizes in Header, Column Sizes Offset, and Column Sizes at Floating Location. The followings steps summarize to carve column sizes when Column Sizes are Stored with Raw Data is True:
1. As each user data field in the row is considered as string is first assumed.

2. The first byte in field is evaluated as an appropriate string size (i.e., the size must not overflow into the next row and an integer must be returned when the string decoding constants ($C_x$ and $C_y$) are applied). Otherwise, another data type is assumed to be present.

3. Finally, the string conditions (discussed in Section sec:datadecoding2) must be meet for the given field. Otherwise, another data type is assumed to be present.

4. The breaking condition for iterating over fields in the user data fields is either the column count (when present) or when the carver reaches the next row.

The followings steps summarize to carve column sizes when Column Sizes in Header is True:

1. The carver assumes that the next byte(s) read is a string size.

2. The byte(s) evaluated is an appropriate string size (i.e., the size must not overflow into the next row and an integer must be returned when the string decoding constants ($C_x$ and $C_y$) are applied). Otherwise, the end of the column sizes stored in the header has been reached.

3. Finally, the string conditions (discussed in Section sec:datadecoding2) must be meet for the given string size. Otherwise, the end of the column sizes stored in the header has been reached.

4. Another breaking condition used by the carver to determine the end of the column sizes stored in the header are the column count (when present) or when the sum of the column sizes is equal to the end of the row.

The followings steps summarize to carve column sizes when Column Sizes at Floating Location is True:

1. The carver assumes that the next byte(s) read is a string size.

2. The byte(s) evaluated is an appropriate string size (i.e., the size must not overflow into the next row and an integer must be returned when the string decoding constants ($C_x$ and $C_y$) are applied). Otherwise, the end of the column sizes stored in the header has been reached.
3. Finally, the string conditions (discussed in Section sec:datadecoding2) must be met for the given string size. Otherwise, the end of the column sizes stored in the header has been reached.

4. Another breaking condition used by the carver to determine the end of the column sizes stored in the header are the column count (when present) or when the sum of the column sizes is equal to the end of the row.

4.3.3.7 Raw Data

The raw data refers to the user data fields. The carver locates the raw data and then determines datatypes and reconstructs fields using data decoding rules discussed in Section 4.3.4. The following steps describe how the carver locates the raw data:

1. If Raw Data Fixed Offset is not NULL, then the carver moves to offset of the Raw Data Fixed Offset parameter value.

2. Otherwise, the carver searches for the bytes stored in the Raw Data Delimiter parameter and moves to the offset of the Raw Data Delimiter Offset parameter value.

4.3.3.8 Additional Obsolete Record Pass (Unallocated Storage Scan)

We define an obsolete record as a record that resides in unallocated storage. Obsolete records are typically created by SQL DELETE or UPDATE, but can also be created by other commands, such as object reorganization or defragmentation commands. For example, the PostgreSQL VACUUM command defragments a table. The VACUUM command moves active records to a new page location, an obsolete copy remains in the old page location. The VACUUM command then erases the row directory pointer for the obsolete copy making it unallocated storage in the page.

Up until this point, obsolete records in unallocated storage are carved if the corresponding metadata for a record is still intact (i.e., the row directory pointer and row header metadata). However, it is still possible that metadata for an obsolete record in unallocated storage was destroyed (e.g., a row directory pointer was NULL-ed upon record deletion). As a final step to carve a page, the carver searches for any remaining obsolete data in unallocated storage. The following describes the process to reconstruct this data when the row directory pointer no longer exists, but the row data header metadata is still intact. If a row data header is not completely intact, data can still be carved, but we do not make any claims on this process.
1. The carver collects all remaining byte segments of the page that were not carved.

2. We assume that a record must contain at least 4 bytes. Therefore, if a byte segment is less than 4 bytes, the carver does not consider it for carving.

3. To determine if a record is present, the carver iterates over the bytes in a given segment.

4. For each iteration, the carver assumes a record is present and attempts to reconstruct all metadata fields described in this section.

5. If any of the metadata fields cannot be reconstructed or a metadata field does not meet any declared assumptions, then the carver continues to iterate through the segment.

6. If all metadata and user data conditions are fulfilled, the record is returned and iteration continues.

4.3.4 Data Decoding

4.3.4.1 ASCII Strings

The parser evaluates each byte to confirm if a raw data value is an ASCII string. We assume that ASCII strings only contain printable characters (i.e., decimal value 32 to 127). If any byte is found to be outside of this range, the parser aborts string parsing, and moves on to another data type.

If 4-byte alignment is True, then the appropriate number of NULL bytes following the string are removed from the record.

4.3.4.2 Integers

The default integer reconstruction for the parser is an unsigned 32-bit little endian integer. Known exceptions are listed below.

Anonymous DBMS#1. A signed 32-bit little endian integer is reconstructed.

Anonymous DBMS#2. A zero-compression key is used to reconstruct the number. In this case, there are three main components to integer reconstruction:

1. The first byte represents the number of bytes needed to store the integer.

2. The second byte is a zero-compression key. For example, the value 192 means a 1-digit integer, 193 means a 2-digit integer, 194 means a 4-digit integer, and 195 means a 6-digit integer. Each time the zero-compression key increases by (past 193), the
number of digits for the integer increases by two. The integer zero uses the value 128 for the zero-compression key.

3. Any remaining bytes represent two digits of the integer, but 1 must be subtracted from that value. For example, the values 12 and 12 represent the integer 1111. If the resulting integer does not have a number of digits that corresponds to the zero-compression key, the integer is padded to the right with the appropriate number of zeros.

4.3.4.3 Other Datatypes

Besides strings and integers, other datatypes often require DBMS-specific functions. A decoding function can be created by collecting the byte storage representation for a set of known synthetic values and determining the encoding. Once a decoding function is created for a specified DBMS datatype, an IF condition is added to the carver to detect the datatype; the DBMS is known since this information is included in the parameter file. When the IF condition is met by the carver, the bytes are passed to the respective decoding function.
Chapter 5

Standard Storage Format

5.1 Introduction

Database forensic carving tools have been proposed [33 17 86 87 91 65], but incorporating their output into an investigation remains difficult to impossible. The storage disparity between DBMSes and operating systems may well in fact be the main culprit for the stunted growth and limited applications of database forensics. We identified two major pieces currently missing from the field of database forensics that have prevented its involvement in forensic investigations: 1) a standardized storage format, and 2) a toolkit to view and search database forensic artifacts.

Standard Storage Format  A standard database forensic storage format would abstract the specifics of DBMS storage engines for users unfamiliar with DBMS internals and guide the development of database carving tools. All DBMSes use their own storage engine. A standard storage format would allow users to view and search database forensic artifacts, generate reports, and develop advanced analytic tools without knowledge of storage engine specifics for any given DBMS. A uniform output for database carving tools would also allow these tools to be compared and tested against each other.

View and Search Toolkit  A toolkit to view and search reconstructed DBMS artifacts would allow investigators to easily interpret the artifacts. While database data is stored and queried through records in tables, the records alone do not accurately represent the forensic state of a database since this data is accompanied by a variety of metadata (e.g., byte offset of the record). Investigators need a way to view how the metadata and table records are interconnected.

In this chapter, we describe a comprehensive framework to represent and search database forensic artifacts. A preliminary version of this framework was implemented for this chap-
Figure 5.1: The role of DB3F and DF-Toolkit in database forensics.

The introduction of a standardized intermediate format and a comprehensive toolkit for database forensics benefits the community in two important ways. First, it streamlines the addition of new tools on either side of the flow chart in Figure 5.1. With the introduction of a new database carving tool (e.g., Tool D), users would benefit from all available advanced applications that support DB3F. Similarly, any newly developed advanced application can trivially process output from any carving tool that supports DB3F output. This intermediary approach is conceptually similar to Low Level Virtual Machine (LLVM) [44], a collection of reusable compiler technologies that defines a set of common language-independent
primitives. The second benefit is the explicit documentation and built-in reproducibility of the analyses process and outcomes, bringing a scientific approach to digital forensics. Garfinkel [24] emphasized the lack of scientific rigor and reproducibility within the field; although in [24] he focused on developing standard corpora, a standard storage format as well as a querying and viewing mechanism is also necessary to achieve these goals. Rather than building custom analytic tools (e.g., [85]), DF-Toolkit’s approach will offer a well-documented querying mechanism based on defined standard fields in DB3F. Any query report can be easily reproduced by another party or re-tested via a different database carver.

This chapter serves as the foundation for a vision of a complete system with full support for database forensics and integration with other forensic tools. Section 5.6 discusses planned improvements for future developments to our framework, including advanced analytic applications.

5.2 Related Work

This section presents work related to both DB3F and DF-Toolkit. To help formulate our storage format, we took into consideration metadata usage by many forensic tools, the capabilities of database carving tools, and forensic languages used outside of database forensics. To help design our view and search toolkit, we consider the evidence tree structure used by many forensic tools and current data filtering approaches.

5.2.1 Storage Format

Metadata Standards File system metadata is widely used in digital forensics to navigate file system information and reconstruct event timelines. Popular tools, such as The Sleuth Kit [9], FTK [1], and EnCase [32] use body files to represent this metadata. Thus, our database forensic storage format was designed to include not only the records that could be accessed through a live system, but also the DBMS metadata, which users may not always have access to through the DBMS API.

Database Carving Tools Several database carving tools exist, but they lack a unified output to store their results. These tools examine and reconstruct database forensic artifacts at the page level. Pages (typically 4K or 8K) are the minimum read/write unit for all row-store relational DBMSes. Page configuration is typically described in documentation by DBMS vendors (e.g., Oracle [61], Microsoft SQL Server [52], IBM DB2 [36]).
PostgreSQL [30], MySQL [62], and SQLite [81]). Drinkwater was one of the earliest to
describe a database carving approach for SQLite DBMSes [17]. Guidance Software’s SQLite
Parser implements much of what Drinkwater discussed; they reconstruct both allocated
and unallocated SQLite data [33]. SQLite Parser returns the results in the form of a new
SQLite instance (i.e., a single database file). Wagner et al. proposed a generalized method
to learn and reconstruct DBMS storage through page carving [86, 91]. They proved this
method worked for most row-store relational DBMSes, including Apache Derby, Firebird,
IBM DB2, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, and SQLite. Their tool,
DBCarver, returned much of the metadata along with the allocated and unallocated user
records in a series of CSV files. Percona’s Recovery Tool for InnoDB recovers MySQL
DBMS files [65], but we do not consider it a tool for forensic investigations. Once MySQL
files are recovered, they are imported into a live MySQL instance. Therefore, none of
the unallocated data or metadata is presented to the user. One of the main goals in this chapter
is to define a unified storage format for the allocated data and unallocated data returned by
the work of Drinkwater and Guidance Software, and the allocated data, unallocated data,
and metadata returned by the work of Wagner et al. To evaluate DB3F and DF-Toolkit
for this chapter, we used our previously developed page carving tool, DBCarver [91]. As
DBCarver does not support DB3F output, we converted its output (CSV files) into DB3F.

Structured Forensic Languages File names and file system properties are represented
in formats such as JSON or XML with digital forensic tools. Some examples include Man-
diant’s Indicators of Compromise in Malware Forensics [47], The MITRE Corporation’s
Making Security Measurable Project [50], and DFXML by Garfinkel et al. [26, 22]. For
this project we used JSON to represent database forensic artifacts. JSON can readily be
migrated to XML if needed using most programming languages.

5.2.2 View and Search Model

Evidence Tree Most forensic tools (e.g., FTK, The Sleuth Kit/Autopsy, and Encase) that
offer an interface to traverse and view artifacts use a tree structure to present these forensic
artifacts. Database forensic artifacts are inherently different from typical forensic objects;
therefore, objects such as files cannot serve as tree nodes. For example, a database table
can span across multiple files (as in PostgreSQL) or a database file can contain multiple
database tables and indexes (as in Oracle). In this chapter, we present a new evidence tree
that was inspired by existing tools, but designed to represent database forensic artifacts.
Filtering SQL is a powerful tool that can enhance searching forensic artifacts. Instead of iterating over a series of files, forensic filtering tools can integrate SQL (i.e., relational) database capabilities. FTK and The Sleuth Kit store case information in SQL databases, and we believe our framework should take the same approach. The main challenge with this, which we address in this chapter, is that to properly use SQL, the data must be first stored in a properly defined relational schema. Some of the forensic SQLite tools (e.g., Guidance Software’s SQLite Parser) return results as a SQLite DBMS file, which can be natively filtered using SQL. However, it does not include forensically relevant metadata defined in [86], which we believe should be incorporated. Therefore, simply recreating the DBMS is insufficient as it provides only data and not metadata. The following examples illustrate this problem with two simple questions a database filtering framework should be capable of answering.

Example 1: “Return all deleted records and their offsets” A recreated DBMS does not store metadata that describes deletion status of a record or its offset within a disk image. To answer this query, at least two additional columns (deletion flag and position offset) must be added to every table reconstructed in the DBMS. It is immediately apparent that such a model is not extensible, as additional metadata columns will be needed to support answers for other types of forensic queries. Furthermore, by adding meta-columns, distinguishing the meta-columns from the original (“real”) data columns could become a challenge for users.

Example 2: “Find all records containing the string ‘MaliciousText’” This query poses even more challenges than the previous example. The user must search all columns across all tables. Such operation is not well-supported by SQL, as SQL language has no capability to apply a filter condition “for all columns”. To illustrate this problem, assume we know there is just one table, Employee. The following query would have to be written for every table:

```sql
SELECT *
FROM Employee
WHERE FirstName LIKE '%MaliciousText%'
OR LastName LIKE '%MaliciousText%'
OR Department LIKE '%MaliciousText%'
OR JobTitle LIKE '%MaliciousText%';
```

We discuss our solution for this problem in Section 5.5.1.
5.3 Design Requirements

The requirements identified for this work were based on the overall goals and challenges in digital forensics discussed by Garfinkel [27] and the requirements defined by other digital forensic frameworks, including Autopsy [8], DFXML [26, 22], and FROST [18]. This section describes some of the key requirements we considered for the design DB3F and DF-Toolkit.

5.3.1 DB3F Requirements

**Storage Engine Agnostic**  One of the major goals of DB3F is to abstract DBMS storage engine specifics. This abstraction must generalize to all row-store DBMSes and not lose any forensic artifacts. One example of an artifact that may be interpreted differently depending on the specific DBMS is the storage of the DBMS-internal object identifier metadata. An object identifier is a unique identifier for each object in the DBMS; it maps back to a system table for the object’s plaintext name (e.g., Employee). Most DBMSes store the object identifier in the page header. Alternatively, PostgreSQL stores the object identifier with each individual record (even though it is redundant, as a single database page can only contain data belonging to one object). The function of the object identifier remains the same despite where it stored. Therefore, DB3F should remove the need to know the specifics of how such metadata is stored.

**Simple to Generate and Ingest**  DB3F should be generated by all database carving tools and used by any searching or advanced analytic tools. Therefore, the DB3F should be easy to generate, and parsing data from DB3F should be trivial.

**Open and Extensible**  DB3F should be publicly available and open sourced. Fields should be easy to add to the public standard. Additionally, given the potentially wide variety of organizations and databases that may use DB3F, custom field addition should be allowed – new custom fields should be easy to introduce. For example, the standard operating procedure for one organization may require chain of custody information that is currently not a field in the DB3F file header. In such cases, it should be easy for an organization to introduce this information into DB3F.

**Scalable**  The amount of database forensic artifacts that may be discovered and will require processing is unpredictable (and projected to continuously increase). An investigation may involve a small (KBs), lightweight DBMS from a personal device, or it may involve
a large (PBs), data warehouse stored on multiple servers. Moreover, an investigation may involve multiple computers (e.g., a network of IoT devices), each with their own individual DBMS. Therefore, the amount of carved data stored in DB3F should not impact the system capabilities.

5.3.2 DF-Toolkit Requirements

Visibility Forensic tools return a wide variety of data and metadata to users. These artifacts should be organized and presented to users in a manner such that the data can be traversed. This is traditionally done using a representative tree structure where the root nodes are the critical data structures (e.g., disk partitions), the next level nodes are used to store the data objects (e.g., stand-alone files), and all other node levels are used to store information about the carved data objects.

Display Data Objects Given that the user can view a logical organization of the forensic artifacts in an evidence tree, the user would most certainly want to view the data objects and their content. Such viewing should be allowed through a user interface.

Object Filtering When a user is presented with a large number of data objects, she may desire to filter these to a relevant subset. For example, a user may only be interested in JPEG files so a corresponding filtering condition (filetype = ‘JPEG’) may be applied. In DBMSes, a user may want to filter objects based on the metadata, such as object type (e.g., table, index, materialized view), number of columns, or object size.

Keyword Searches Keyword searches are commonly used in forensic investigations to find relevant evidence. String matches and regular expressions should be supported for filtering records (e.g., find all mentions of ‘Bob’).

Reporting Reports need to be generated to help analysts make conclusions and present their findings. Furthermore, this reporting should allow for comparison and validation of database forensic carving tool output.

5.4 The Database Forensic File Format

This section presents our storage format for database forensics, DB3F. This is the format that should be used by different database carving tools to output their results.
5.4.1 File Layout

When a database carving tool is passed a file, a carver tool analyzes it for the presence of one or more different DBMSes. Since each DBMS is a self-contained system, data from different DBMSes should not be mixed within the same carver output file. Each DBMS is stored as a separate output file.

Multiple DBMSes may exist on the same piece of evidence. However, it is acceptable for multiple carver output files to be associated with a single DBMS. For example, a series of DBMS files (from a single file system or multiple nodes) belonging to the same DBMS may be passed to the carver as opposed to a single disk image. Moreover, the RAM snapshot(s) will be a separate evidence file for any given DBMS. Therefore, this condition is required if one wants to compare the data from a disk image and a RAM snapshot.

Example 3: File Layout DiskImage01.img is passed to a database carving tool. The carving tool analyzes the evidence for data belonging to PostgreSQL and SQLite DBMSes. This results in two output DB3F files (one for each DBMS): PostgreSQL.json and SQLite.json.

5.4.2 DB3F Files

Each DB3F file stores a series of JSON objects. The first line in a DB3F files contains a JSON object that serves as a header. Every other line in the DB3F contains a JSON object that represents a database page.

Representing the entire carved DBMS with a single JSON object has scalability problems because the amount of data in a DBMS can be arbitrarily large. Therefore, one JSON object per DBMS page allows us to achieve the scalability requirement (see Section 5.3). The physical order of DBMS pages is irrelevant because each page object stores its offset within the disk image.

5.4.3 DB3F Header

The DB3F file header JSON object contains high-level metadata about the artifacts in the file and how they were collected. The list below describes the fields, which should be returned by the database carving tool, stored in the header. Additionally, Figure 5.2 displays a DB3F file header with example data. Since we cannot anticipate all of the header information each specific organization may require, JSON fields can easily be added.

c context (array): namespace information.
name (string): the organization name used to identify custom header information.

uri (string): unique identifier for an organization.

evidence_file (string): the disk image or RAM snapshot from where the forensic artifacts originated.

forensic_tool (string): the database carving tool used to generate the forensic artifacts.

carving_time (string): the system global time when the carver finished generating the DB3F file.

dbms (string): the DBMS vendor and its version.

page_size (number): the page size used by the DBMS. Page size is assumed to be constant across an entire DBMS. It is theoretically possible to use more than one page size in a DBMS. However, we assume the database carving process will extract different page sizes as belonging to different DBMSes.

### 5.4.4 DB3F Database Pages

Each line following the DB3F header contains a single JSON object that represent a database page. Each page stores 1) page header fields and 2) an array of JSON objects that represent records. Figure 5.3 displays an example of how DB3F represents a PostgreSQL DBMS page storing Star Schema Benchmark data [59]. The fields in this figure are defined in this section.
Page Header  The page header stores information that is general to all records within the page. The page header fields are the following:

offset (number): the page address within the evidence.
page_id (string): a unique identifier the DBMS assigns to a page.

object_id(string): an identifier that the DBMS uses to map to the plaintext name of each object (e.g., table or index).

page_type (string): the type of object to which the page belongs.

schema (array): the data types for the record columns within the page.

records (array): a list of JSON objects for each record.

Record A JSON object exists for each record in the page. The record fields are the following:

offset (number): the record address within the page.

allocated (boolean): True indicates the record is allocated, while False indicates the record is deleted (i.e., unallocated storage).

row_id (string): an internal DBMS pseudo-column.

values (array): the individual record values.

The fields defined in this section is not an exhaustive list. We anticipate that new fields will be added to the DB3F standard as the tool use grows and organizations will want to add their own custom fields.

Discussion: Datatype Support While the example data in Figure 5.3 illustrates only strings and numbers, DB3F supports all DBMS datatypes. Each datatype is described in the page header schema field, and the value is stored among the values field for a record. Users may be concerned about storing values that do not fit into a single page, such as Binary Large Objects (BLOBs) and large text fields. To store BLOBs, DBMSes do not directly store the binary data within the page, but rather store a reference to a file containing the binary data. For example, a DBMS would store a reference to a JPEG file in a page rather than the binary JPEG data. DB3F would similarly store a reference to a file, with the actual binary file (e.g., JPEG) stored in a separate dedicated location. It is possible for a text value to span across more than one page. In this instance, each DB3F page object describes the text stored in an individual page, allowing the long text value to be reconstructed independently. Additionally, in some case text field will store pointers to the remainder of the text located in different pages. In such cases, DB3F will store whatever
information is provided by the database page carving tool. Additional analysis is required to rebuild the entire text value – a DBMS pointer can be reconstructed using the metadata already stored in DB3F fields. The work in [89] discusses DBMS pointer reconstruction in more detail.

5.4.5 Evaluation

To verify the reliability of DB3F, we used three DBMSes: PostgreSQL, Oracle, and SQLite. We loaded Star Schema Benchmark data [59] at Scale 1 (600MB) into each DBMS, used DBCarver to carve the DBMS files, and converted the CSV file output into DB3F. We converted the artifacts carved from Oracle and PostgreSQL DBMS files into DB3F without any problems. However, since SQLite does not store an object identifier in the pages, this metadata could not be included in DB3F directly. As an alternative, we used the table schema (i.e., the string with column datatypes) to represent the object identifier. This decision was made because all records for the same table will have the same number of columns and the same datatype for each column. However, we note that more than one table can have the same schema; thus, our decision merged tables with identical columns in SQLite. Table 5.1 summarizes the sizes of the DBMS files passed to DBCarver and our generated DB3F files for the 600MB Scale 1 SSBM data used. The DB3F storage overhead allows for human readability. However, DB3F can be compressed to scale for analysis of larger forensic datasets.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>DBMS(MB)</th>
<th>DB3F(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>625</td>
<td>1329</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>648</td>
<td>1298</td>
</tr>
<tr>
<td>SQLite</td>
<td>445</td>
<td>1308</td>
</tr>
</tbody>
</table>

Table 5.1: File size comparison of DB3F file to the DBMS file for a 600 MB CSV file of raw data.

5.5 The Database Forensic Toolkit

This section presents our toolkit, DF-Toolkit, to view and filter DBMS forensic artifacts stored in DB3F. First, we describe the evidence tree structure that serves as a core concept behind this toolkit. This tree structure allows users to traverse and view metadata and data stored in DB3F files. Next, we discuss how this tree allows carved database metadata and data to be searched and filtered by the user. Finally, our solution to reporting filtered metadata and data in DB3F is described.
Throughout this section we refer to Figures 5.4 and 5.5. As a proof of concept, Figure 5.4 displays our implemented user interface to display the evidence tree. Figure 5.5 contains the relational schema used to store the evidence tree nodes in a SQL database for searching and filtering results. These tables are populated when a tree is first viewed; they can be cached or rebuilt by DF-Toolkit as necessary.

5.5.1 The Evidence Tree

The evidence tree presented in this section follows the same principles as many popular digital forensic tools (e.g., The Sleuth Kit, FTK, EnCase). Similar to these tools, we classify three main node levels in the tree: root, object, and object description. Alternatively in this chapter, the tree nodes are defined to accurately represent database forensic artifacts.

Root The root node serves as a critical storage structure from which all other data can be reached. For example, a disk partition may be a root in commonly used forensic tools. Since DBMSes manage their own storage, a disk partition does not represent a major storage structure in a DBMS. For example, a DBMS may store files across multiple disk partitions. When this is done, system tables and user tables would likely be stored on different partitions. Furthermore, a single table could be stored on multiple disk partitions. Therefore, a DBMS sample (i.e., the complete or partial DBMS storage content taken from a
storage medium) makes an appropriate storage structure for a root. A database root node is not expected to contain an entire DBMS. It is likely that the piece of evidence is a disk partition, RAM snapshot, or contains a corrupt (e.g., partially overwritten) DBMS. Therefore, by “DBMS sample”, we mean all of the data associated with a DBMS for a given piece of evidence.

In Figure 5.4, there are two images that represent evidence, Image01.img and Image02.img. Image01.img contains two root nodes (i.e., DBMS samples), PostgreSQL and MySQL. Since DB3F requires that a carver tool store DBMS samples in separate output files, each root node always corresponds to a single DB3F file.

In Figure 5.5, the DBMS_Sample table stores a record for each root node. DB3F_File is a reference to the DB3F file. This also serves as a unique identifier (i.e., primary key) for each DBMS sample record. DBMS is the DBMS vendor name and version. PageSize is the page size used by the DBMS sample. PageCnt refers to the number of pages associated with the DBMS sample. Therefore, the total DBMS sample storage size can be calculated using $\text{PageSize} \times \text{PageCnt}$. DiskImage is a reference to the evidence (e.g., disk image, RAM snapshot) associated with this DBMS sample. This column also references (i.e., a foreign key) the Evidence table. For every entry in the DBMS_Sample table, a new schema is created containing an Object table, Page table, and Record table.

**Data Objects** The next level in the tree are the data objects for which the root is examined. For example, a stand-alone file (e.g., PDF, Word document) may be a data object in commonly used forensic tools. DBMS files can contain multiple DBMS objects (e.g., tables), and a DBMS object can span across multiple DBMS files. Artifacts belonging to each DBMS object should be associated with each other. Therefore, DBMS files themselves should not be treated as the data objects like traditional stand-alone files. A more suitable candidate for the data object node are the DBMS objects (e.g., customer table, employee table, customer name index). DBMS metadata and data can be associated with DBMS objects by using the object identifier metadata stored within DBMS pages (discussed in Section [5.3.1]). Additionally, viewing the DBMS files themselves does not provide the user with much useful information since they are not stand-alone.

In Figure 5.4, the PostgreSQL root node has four data objects: 1113438, 1113446, 1113441, and 1113440. Statistics and metadata describing the selected object, 1113440, is displayed in the bottom left-hand box. This object is a table with 28 pages (not all displayed) and seven columns (one number and six strings), beginning under the heading, “Record”. 

81
In Figure 5.5, the Objects table stores information about each object. ObjectID is the object identifier used by the DBMS, which also serves as the primary key. Type represents the type of DBMS object (e.g., table, index, or materialized view). PageCnt stores the number of pages associated with the object. ObjectSchema represents the data types for each column in the table.

EVIDENCE

<table>
<thead>
<tr>
<th>DiskImageName</th>
<th>Description</th>
</tr>
</thead>
</table>

DBMS_Sample

<table>
<thead>
<tr>
<th>DB3F_File</th>
<th>DBMS</th>
<th>PageSize</th>
<th>PageCnt</th>
<th>DiskImage</th>
</tr>
</thead>
</table>

DB3F_File.OBJECT

<table>
<thead>
<tr>
<th>ObjectID</th>
<th>Type</th>
<th>PageCnt</th>
<th>ObjectSchema</th>
</tr>
</thead>
</table>

DB3F_File.PAGE

<table>
<thead>
<tr>
<th>Offset</th>
<th>PageID</th>
<th>ObjectID</th>
</tr>
</thead>
</table>

DB3F_File.RECORD

<table>
<thead>
<tr>
<th>PageOffset</th>
<th>RecordOffset</th>
<th>RowID</th>
<th>Allocated</th>
<th>Record</th>
</tr>
</thead>
</table>

Figure 5.5: The relational schema used to store the evidence tree data in a SQL database.

Object Information Two more tree levels are used to recursively store information about each object at the page level and the record level. Storing information about each DBMS page allows for statistics to be quickly collected for an object (or a fast stochastic analysis), and removes data redundancy at the record level.

In Figure 5.4, the pages associated with the selected object, 1113440, are displayed in the right-hand side box. We know there are a total of 28 pages, which are not all displayed in the figure, based on the object information in the bottom left-hand box.

In Figure 5.5, the Page table stores information about each page. Offset refers to the byte address of the page with the evidence file. This also serves as the primary key. PageID is metadata used by the DBMS to uniquely identify pages. Note, that we do not use this as the primary key because multiple copies of a page may exist (e.g., one from the DBMS.
file and one from a paging file on the same disk image). ObjectID is metadata used by the DBMS to identify objects, and this column also references the Object table.

Information about each record within a page is the last node level in our evidence tree. In Figure 5.4, the records associated with the selected page, offset = 3784704, are displayed in the right-hand side box. In Figure 5.5, the Record table stores information about each records. PageOffset refers to the byte address of the page within the evidence file. This column also references the Page table. RecordOffset refers to the byte address of a record within the page. PageOffset and RecordOffset together serve as the primary key. RowID is metadata, which is a DBMS internal pseudo-column. Allocated identifies the record as ‘active’ or ‘deleted’ (i.e., unallocated). Record is a string that combines all record values. Each value within the string has single quotes around it, and all values are separated by a comma.

<table>
<thead>
<tr>
<th>Offset</th>
<th>RowID</th>
<th>Alloc.</th>
<th>Pos.</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>72</td>
<td>True</td>
<td>1</td>
<td>‘430’</td>
</tr>
<tr>
<td>318</td>
<td>72</td>
<td>True</td>
<td>2</td>
<td>‘Supplier#000000430’</td>
</tr>
<tr>
<td>318</td>
<td>72</td>
<td>True</td>
<td>3</td>
<td>‘9eN nRdw0Y4tl’</td>
</tr>
<tr>
<td>318</td>
<td>72</td>
<td>True</td>
<td>4</td>
<td>‘ARGENTINAS’</td>
</tr>
<tr>
<td>318</td>
<td>72</td>
<td>True</td>
<td>5</td>
<td>‘ARGENTINA’</td>
</tr>
<tr>
<td>318</td>
<td>72</td>
<td>True</td>
<td>6</td>
<td>‘AMERICA’</td>
</tr>
<tr>
<td>318</td>
<td>72</td>
<td>True</td>
<td>7</td>
<td>‘11-406-611-4228’</td>
</tr>
</tbody>
</table>

Table 5.2: Sample representation of carved rows on per-value basis.

We stop recursively constructing the tree at the record level. That is, the leaf level of the evidence tree is a database record (e.g., a single row in Figure 5.4) rather than a field (e.g., ‘ARGENTINA’ in Figure 5.4). Logically, another tree level could be added for individual values. For our current version of DF-Toolkit, this step is not needed for plaintext searches. We believe that extending the evidence tree to include individual fields of the database table should be explored in the future to support more advanced analysis; however, the proper execution of such a feature will introduce significant implementation challenges. Continuing to represent data with a proper relational schema (as in Figure 5.5) does not scale well when individual values are considered because each value must now be stored as an entry in the Value table – for example, representing the first row from Figure 5.4 at individual value level as shown in Table 5.2. Therefore, to search for an individual value, an entry from the Value table would need to JOIN with the Record table.

Another possible approach would be to create a new table for each DBMS object from each DBMS. The data would be ingested from a CSV file generated from the DB3F file.
This approach would be similar to Guidance Software’s SQLite Parser (discussed in Section 7.2). While we envision this to be a more viable solution, an incomplete DBMS from evidence such as a RAM snapshot or corrupt DBMS poses an implementation challenge; table columns would be ambiguously defined causing problems when querying data. For example, column names would need be created as Column1, Column2, etc. In general, we do not consider the presence of a complete DBMS to be a safe assumption for DF-Toolkit purposes.

5.5.2 Data Display Filters

Data filtering is performed at the DBMS level; tables (or objects) for each DBMS schema are considered. The following is the basic query need to properly connect a DBMS schema before applying filtering conditions, where DB3F_File is the root node:

```
SELECT *
FROM DB3F_File.Object O,
    DB3F_File.Page P,
    DB3F_File.Record R
WHERE O.ObjectID = P.ObjectID
    AND P.Offset = R.PageOffset
```

This query returns all rows from the Objects, Page, and Record tables for a given DBMS so that the data can be put back into DB3F (this is further explained in Section 5.5.3). Beyond this query, only a basic understanding of SQL is needed to perform custom filtering.

**Objects** Users can filter objects by simply adding `WHERE` clause conditions to the query above. Objects can be filtered based on the following metadata fields: ObjectID, Object Type, Object Page Count, and Object Schema. For example, if the user was only concerned with the object with seven columns (one number and six strings), the following condition would be added:

```
AND O.Schema = ‘NSSSSSS’
```

**Pages** Users can also filter pages with `WHERE` clause conditions. Pages can be filtered based on the following metadata fields: Page Offset, PageID, and Page ObjectID.

**Records** Finally, users can filter record with `WHERE` clause conditions. Records can be filtered based on the following metadata fields: Record PageOffset, Record Offset, Record RowID, Record Allocated/Deallocated, and the data stored in the record. Most importantly,
users would want to apply keyword searches to the data stored in records. All of the values for a carved record are stored as a single string making this feature easy to support. Since SQL supports string matches, wildcards, and regular expressions, keyword searches can be applied by adding another WHERE clause condition(s). For example, to search for all records containing a phone number (in the format of the data from Figure 5.4):

```
AND R.Record REGEXP '\d{2}-\d{3}-\d{3}-\d{4}'
```

Figure 5.6 displays an example interface to apply filtering within our user interface. The JOIN conditions are previously written, simplifying user interaction. The user then adds the two example conditions presented for object filtering and keyword searches.

Figure 5.6: DF-Toolkit filtering implemented with a user interface.

### 5.5.3 Report Generation

After filtering is applied, the results are returned as DB3F. Storing the report back into DB3F allows the data to be viewed within the evidence tree, available for further filtering, and to future advanced analytic tools. We note that DF-Toolkit was able to find every relevant carved artifact in its search (providing a search accuracy of 100%). Report accuracy is thus dependent only on the accuracy of carving provided by the database carving tool(s).

### 5.6 Conclusion and Future Work

This chapter presented a new storage format for database forensic artifacts called the Database Forensic File Format (DB3F), and a toolkit to view and search data stored in DB3F called the Database Forensic Toolkit (DF-Toolkit). Additionally, a user interface was presented to provide a display of DF-Toolkit.
We envision that DB3F and DF-Toolkit will serve as the groundwork for a complete forensic and security analysis system. Future work for this system is discussed below, which includes: incorporating DBMS system data carved from the evidence, carver tool integration, multi-evidence analysis, and non-page data integration.

5.6.1 System Catalog Information

While the metadata presented to users through DF-Toolkit is accurate, some DBMS forensic artifacts may become difficult to interpret for users, especially as the amount of data increases. For example, the object identifiers (e.g., ‘1113440’) alone do not mean as much as the plaintext object name (e.g., ‘Supplier’) to an investigator exploring evidence. Our top priority for future work is to automate the detection and association of DBMS system catalog information, which is stored in DBMS system tables, to replace such metadata with more readable plaintext. We do see two main challenges with this work. First, the system catalog may not always be present (e.g., corruption of data on disk or when using a RAM snapshot). Therefore, DF-Toolkit would need to accurately communicate to a forensic analyst why such metadata is not available. Second, each DBMS has its own system table schema. Therefore, detection and association of this information requires tailored functions for each DBMS vendor.

5.6.2 Carver Tool Integration

For this chapter, we generated DB3F files from carved output stored in CSV files. This step would be tedious for users, and we believe it should be streamlined. Ideally, we would like to work with the current and future creators of database carving tools (Section 5.2) to return their results in DB3F. Making DB3F publicly available will help to catalyze this effort.

5.6.3 Advanced Analysis

This chapter presented straightforward filter and search examples for single pieces of evidence. However, we envision a more complete toolkit to access and interpret database forensic artifacts. This mostly comes in the form of a database forensic API, which would be a DBMS complement to Garfinkel’s Fiwalk [26]. The primary uses for such work include multi-evidence analysis and integration with non-DBMS page data and other forensic tools.
**Multi-Evidence**  An investigation may involve multiple pieces of evidence when a series of disk images or RAM snapshots was collected, a DBMS was distributed across multiple nodes, or multiple devices contained individual DBMSes. In these cases, metadata and data can be compared to recreate event timelines. Most IoT devices typically store information locally on a lightweight DBMS (e.g., SQLite), send information to a server that uses a more robust DBMS (e.g., MySQL), or both. For example, the Amazon Alexa and Samsung Galaxy images from the DFRWS IoT Challenge 2018 - 2019 [15] each contain a SQLite DBMS. Assuming that these devices had some form of interaction, connecting data and metadata from both devices would help to create an event timeline.

**Integration of Non-DBMS Page Data**  Almost all of the DBMS data and metadata is stored in pages; thus, it can be represented in DB3F and searched with DF-Toolkit. However, connecting metadata and data outside of DBMSes to DB3F files would create more complete timelines. These sources include audit logs, network packets, and files which are referenced by DBMS records. Section 5.2 discussed just some of the tools used to store and searched these data and metadata. We hope that bringing this discussion to the digital forensics community will help bridge the gap between these different domains within digital forensics.
Chapter 6

Database Forensics API

6.1 Introduction

Relational DBMSes adhere to the principle of physical data independence: DBMSes expose a logical schema of the data while hiding its physical representation. A logical schema consists only of a set of relations (i.e., the data). On the other hand, a physical view consists of several objects, such as pages, records, directory headers, etc. Hiding physical representation is a fundamental design of relational DBMSes: DBMSes transparently control physical data layout and manage auxiliary objects for efficient query execution. However, data independence inhibits several security and performance applications requiring low-level storage access. A small example is provided here, while Section 6.2 presents more detailed use cases.

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

Comprehensive DBMS storage-level access is an inherent challenge due to DBMS storage management. DBMSes control *allocated* storage objects such as a) physical byte representation of relations, b) metadata to annotate physical storage of relation data, and c) auxiliary objects associated with relations (e.g., indexes, materialized views). Users can manipulate *allocated* objects exposed by SQL. However, as illustrated in Example 1, the

---

1DBMS encryption is similar in not providing any feedback. Furthermore, encrypted values should still be destroyed on deletion.
DBA may also need access to unallocated storage objects not tracked by a DBMS such as deleted data that lingers in DBMS-controlled files, and DBMS-formatted pages released back to the OS and no longer under DBMS control (e.g., files deleted by the DBMS or OS paging files). These objects are certainly part of the physical view and required for any storage access, but currently not exposed by any DBMS. Vendors such as Oracle incorporate the DBMS_REPAIR package [60], enabling users to manually fix or skip corrupt pages, but such tools only access DBMS-controlled storage.

In order to enable such security and performance applications, we present Open Data Storage Access (ODSA), an API that provides comprehensive access to all DBMS metadata and data in both (unallocated and allocated) persistent and volatile storage. ODSA does not instrument any RDBMS software; it interprets underlying data using database carving methods [91], which we use to expose physical level details. Carving itself is insufficient because the carved data consists of disk-level details making it difficult to program DBMS storage. ODSA abstracts low-level disk-level details with a hierarchical view of DBMS storage that is familiar to most DBAs. In particular it organizes them into pages, records, and values, which are resolved to internal, physical addresses. ODSA also guarantees the same hierarchy applies to multiple DBMS storage engines, ensuring portability of programmed applications. Figure 6.1 shows the storage access enabled by ODSA.

The rest of the chapter is organized as follows. Section 6.2 presents three representative uses cases that require storage-level access. Section 6.3 provides an overview of how applications previously had limited access to internal DBMS storage. Section 6.4 describes the hierarchy exposed by ODSA and how it provides a comprehensive view of storage. Section 6.5 demonstrates implementation and use of ODSA. Finally, Section 6.6 discusses future work for ODSA.
6.2 Use Cases

This section presents three representative use cases that require direct access to different abstractions of storage.

6.2.1 Intrusion Detection

A bank is investigating mysterious changes to customer data. Unbeknownst to the bank, a disgruntled sysadmin modified the DBMS file bytes at the file system level. This activity bypassed all DBMS access control and logging, and still effectively altered account balances. The sysadmin also disabled file system journaling with `tune2fs` to further hide their activity. The bank cannot determine the cause for inconsistencies with the logs alone. Forensic analysis [85, 89] that detects such malicious activity requires comprehensive storage access to compare volatile storage with allocated and unallocated persistent storage.

6.2.2 Performance Reproducibility

Alice, an author, wants to share her computation and data based experiments with Bob so he can repeat and verify Alice’s work. Out of privacy and access constraints, Alice builds a container consisting of necessary and sufficient data for Bob to reproduce. If the shared data is much smaller than original DBMS file, Bob cannot reproduce any performance-based experiment as the data layout of the smaller data will significantly differ from the original layout. To achieve a consistent ratio between Alice’s experiment and Bob’s verification, data layout specification at the record and page level must itself be ported. Currently, data layouts as part of a shared DBMS file in a container cannot be communicated [67].

6.2.3 Evaluating Data Retention

Continuing with Example 1 (Section 6.1), the bank validates their compliance with data sanitization regulations (e.g., EU General Data Protection Regulation or GDPR [72]). After deleting data, the bank independently validates data destruction to ensure compliance. No data sanitization validation guidelines for DBMSes exist beyond a complete file overwrite [38]. This guideline is too coarse, especially for DBMS files containing a few deleted records.

Alternatively, consider a compliance officer that has programmatic access to DBMS storage via ODSA for validation. The officer can easily access all unallocated storage, and determine the location of deleted data that was not destroyed (e.g., DBMS index or table file, OS paging file).
6.3 Related Work

We describe built-in tools and interfaces supported by popular DBMSes, which provide physical storage information at different granularities, but no comprehensive views of storage. The ROWID pseudo-column represents a record’s physical location within DBMS storage (not disk), and is one of the simplest examples of storage-based metadata available to users most DBMSes. Commercial DBMSes typically provide utilities to inspect and fix page-level corruption. Examples include Oracle’s DBMS_REPAIR, Oracle’s BBED (a page editing tool available from Oracle 7 to 10g), and SQL Server’s DBCC CHECKDB. However, even for accessible metadata such as ROWID, built-in tools do not help interpret its meaning; a DBA must manually make such interpretations. Moreover, no DBMS offers access to unallocated storage. Finally, existing tools only consider persistent storage. ODSA offers a universal meaning of DBMS storage (including IBM DB2, Microsoft SQL Server, Oracle, MySQL, PostgreSQL, SQLite, Firebird, and Apache Derby) with support for both persistent and volatile storage.

The term *carving* refers to interpreting data at the byte-level, e.g., reconstructing deleted files without the file system. We previously extended carving to interpret DBMS storage with DBCarver [86][87][91], retrieving both allocated and unallocated data and metadata without relying on the DBMS. DBCarver reads individual files or disk/RAM snapshots and extracts data, including user data and system metadata; it then writes data to a DB3F [88] formatted file. This chapter uses DBCarver to demonstrate the physical information a DBMS can provide.

![Diagram](image)

Figure 6.2: ODSA completes raw database storage abstraction in an end-to-end process for storage access.
6.4 Open Database Storage Access

Figure 6.2 shows how ODSA relies on carving to access raw storage. ODSA abstracts two details from raw storage.

First, it interprets each sequence of raw bytes and classifies it into a physical storage element: Root, DBMS Object, Page, Record, or Value. Thus, given a collection of interpreted raw storage elements, ODSA provides a hierarchical access to these elements by linking them. We provide a brief description of the hierarchy. The root level represents the entry point from all other data to be reached. DBMSes manage their own storage, and a disk partition consisting of both Oracle and PostgreSQL pages, will result in two DBMS roots. The DBMS object level calls return metadata, data, and statistics describing a DBMS object, such as a list of pages or column data types. Pages are uniquely identified by a byte offset in raw storage, rather than the PageID. We also do not rely on the page row directory pointers because deletion may zero out a record’s entry.

Second, the ODSA hierarchy hides DBMS heterogeneity by accessing physical elements (e.g., pages, records) with physical byte offsets, rather than DBMS-specific pointers. Computing a DBMS pointer varies between vendors. For example, Oracle incorporates FileID into index pointer while PostgreSQL does not; index pointers in MySQL differs from both Oracle and PostgreSQL because MySQL relies on index organized tables. Even if all vendors used similar pointer encodings, abstraction is needed in terms of pages since duplicate pages may exist across a storage medium (outside of DBMS-controlled storage, such as paging files). Given Page_A and its physical copy Page'_A, ODSA enables application developers to connect an index pointer referencing Page_A along with Page'_A.

Implementation There are multiple ways to implement the hierarchy. The ODSA hierarchy is currently implemented as a pure object hierarchy (Figure 6.3) and as a relational schema (Figure 6.4). The pure object hierarchy is stored as a JSON file in the DB3F format [88]. The relational schema is a starting representation – it supports basic applications and is normalized to 3NF requirements. A relational schema is realized since application developers may prefer to access a DBMS storage with SQL rather than calling the ODSA directly. However, as we show in Section 6.5 the SQL implementation requires several joins and is quite counter-intuitive, despite it being DBMS physical storage.
# 4.A. Root

```python
class Root:
    def __init__(self, db3f):
        # Initialize
    def get_object_ids(self):
        # Return a list of object ids

# 4.B. Object

class DBMS_Object(Root):
    def __init__(self, parent, object_id):
        # Initialize
    def get_page_offsets(self):
        # Return a list of page offsets
    def get_object_type(self):
        # Return the object type string
    def get_object_schema(self):
        # Return a list of column datatypes

# 4.C. Page

class Page(Object):
    def __init__(self, parent, page_offset):
        # Initialize
    def get_record_offsets(self):
        # Return a list of record offsets
    def get_page_id(self):
        # Return a string for page id
    def get_page_type(self):
        # Return a string for page node type
    def get_checksum(self):
        # Return a string for the checksum
    def get_row_directory(self):
        # Return a list of row pointers

# 4.D. Record

class Record(Page):
    def __init__(self, parent, record_offset):
        # Initialize
    def get_value_offsets(self):
        # Return a list of value positions
    def get_record_allocation(self):
        # Return Boolean allocation status
    def get_record_row_id(self):
        # Return a string for the row id
    def get_record_pointer(self):
        # Return a string for row pointer

# 4.E. Value

class Value(Record):
    def __init__(self, parent, value_offset):
        # Initialize
    def get_value(self):
        # Return string for a data value
```

Figure 6.3: A sample set of ODSA calls.

## 6.5 Using ODSA

For use cases in Section 6.2, two fundamental physical storage access operations are finding unallocated records and matching index pointers to records. ODSA calls enable these operations and show how these operations are achieved in Python and SQL, respectively. The two implementations are shown to contrast programmatic verbosity and maintainability. We focus on ODSA access and do not consider implementation performance.
Example 2: Find Unallocated Records. Use cases 2.1 and 2.3 require a DBA to search and retrieve unallocated records. To retrieve unallocated records, the user must know the carved DBMS file name and the table name (Customer table in this example) from which unallocated records are considered. Figure 6.5 finds and prints all unallocated (e.g., deleted) records from the Customer table. All ODSA calls are highlighted.

The implementation in Figure 6.5 uses ODSA calls to search for unallocated records:

Line 3 retrieves page offsets, which uniquely identify pages. Line 5 then iterates through the pages, Line 6 loads each page, and Line 7 retrieves the record offsets for that page. Finally, Line 7 iterates through records using their identifying offsets within a page. Line 11 retrieves the record allocation status to identify and print unallocated records. The same search and retrieval requires an 8-way join in SQL due to the data hierarchy:

```sql
SELECT PageOffset, RecordOffset, ValueOffset, Value
FROM Object NATURAL JOIN Page
NATURAL JOIN Record NATURAL JOIN Value
WHERE Object.DB_File = 'MyDatabase1.json'
AND Object.ObjectID = 'Customer'
AND Record.Allocated = FALSE;
```
Example 3: Match a Record to an Index Pointer(s). To match a record to pointers in a DBMS object such as an index, the user provides as input specific instances of the record and index objects. In Figure 6.6, Line 5 iterates through all index pages to determine if the input record matches any of the index records. Recall, in an index, records are value-pointer pairs. The code in Figure 6.6 determines offsets of all index pages (Line 7), and for each index page (Line 9) iterates over all index records in that page. Lines 10 fetches the index entry and Line 12 loads the pointer (offset 1 in value-pair) of the current index entry. Finally, for any index pointer match to the record pointer (Line 13), the index entry is printed.

In this example a brute-force iteration over all index pages is necessary, i.e., the program cannot break at the first occurrence of a match in Line 13. In practice, DBMS indexes often contain records of entries that were deleted or updated. For example, consider the record (42, Jane, 555-1234) in the Customer table where name column is indexed. In addition to the expected (Jane, {PAGEID: 12, ROWID: 37}) entry in the index, the index may also contain (Jehanne, {PAGEID: 12, ROWID: 37}) if the customer changed their name from Jehanne to Jane (old index entries will only be purged after the index is rebuilt). Moreover, the index might also contain a (Bob, {PAGEID: 12, ROWID: 37}) entry if another customer named Bob previously deleted their account, free-listing the space for Jane’s record at the same location.

As demonstrated in Figure 6.6, the Python-specific implementation retrieves all records. On the contrary, matching a record to an index in SQL requires a dynamic SQL (shown below) in which after the customary 8-way join to find record values, parameters of each record value must be supplied to match the values. Moreover, this query assumes that there is only one indexed column which is transparently accounted for in the abstraction of the DBMS Object class.
```sql
SELECT V1.Value
FROM Page NATURAL JOIN Record
NATURAL JOIN Value V1 NATURAL JOIN Value V2
AND Page.ObjectId = ? --Index name placeholder
AND V1.ValueOffset = 0 --Indexed value at offset 0
AND V2.ValueOffset = 1 --Pointer is at offset 1
AND V2.Value = ( SELECT Record.Pointer FROM Record
WHERE (DB_File, PageOffset, RecordOffset) =
    (?, ?, ?) /*Record ID placeholders*/);
```

def findIndexEntries(record, Index):
    RecordPtr = record.get_record_pointer()
    IndPageOffsets = Index.get_page_offsets()
    for IndPageOffset in IndPageOffsets:
        IndPage = odsa.Page(Index, IndPageOffset)
        IndROffsets = IndPage.get_record_offsets()
        for IndROffset in IndROffsets:
            IndEntry = odsa.Record(IndPage, IndROffset)
            # IndEntry is a pair (Value, Pointer)
            IndexPointer = odsa.Value(IndEntry, 1)
            if IndexPointer == RecordPtr:
                print IndEntry

Figure 6.6: Using ODSA to find all index entries for one record

### 6.6 Conclusion

ODSA was designed based on the principles and challenges described in [5][74]. In particular, it was designed to be simple and easy-to-use by integrating the terminology used across DBMS documentation. Classes were named based on general concepts giving them an intuitive meaning while abstracting DBMS-specific implementation details. ODSA adheres to single-responsibility principle in that calls focus on single pieces of data and metadata. ODSA supports both 3rd party carving and built-in DBMS mechanisms should vendors choose to expose storage. As a result, ODSA complements physical data independence and enables simple yet powerful implementations of a variety of applications that require access to storage. Additional requirements such as versioning and backward compatibility are future work.
Chapter 7
Log Tampering Detection

7.1 Introduction

Database Management Systems (DBMSes) are commonly used to store sensitive data and, accordingly, significant effort has been invested into securing DBMSes with access control policies. However, once a user has gained elevated privileges in the DBMS (either legitimately or through an attack), the security scheme put into effect can be bypassed, and therefore, can no longer assure that data is protected according to policy. A well-known fact from security research and practice is that it is virtually impossible to create security measures that are unbreakable. For example, access control restrictions 1) may be incomplete, allowing users to execute commands that they should not be able to execute and 2) users may illegally gain privileges by using security holes in DB or OS code or through other means, e.g., social engineering. Thus, in addition to deploying preventive measures such as access control, it is necessary to be able to 1) detect security breaches when they occur in a timely fashion and 2) in event of a detected attack collect evidence about the attack in order to devise counter-measures and assess the extent of the damage, e.g., what information was leaked or perturbed. This information can then be used to prepare for legal action or to learn how to prevent future attacks of the same sort.

When malicious operations occur, whether by an insider or by an outside attacker that breached security, an audit log containing a history of SQL queries may provide the most critical evidence for investigators [51]. The audit log can be used to determine whether data has been compromised and what records may have been accessed. DBMSes offer built-in logging functionality but can not necessarily guarantee that these logs are accurate and have not been tampered with. Notably, federal regulations, such as the Sarbanes-Oxley Act [3] and the Health Insurance Portability and Accountability Act [2], require maintaining an audit trail, yet the privileged user can skirt these regulations by manipulating the logs. In
Figure 7.1: Illustrates that the active records for Peter and Bob can be explained by audit log events, whereas the deleted record Malice can not be explained by any audit log events.

such cases, companies maintaining these systems are, technically, in violation of these regulations. Hence, assurance that security controls have been put into place properly cannot rest merely on the existence of logging capabilities or the representations of a trusted DBA. Internal controls are needed in order to assure log integrity.

**Example 1** Malice is the database administrator for a government agency that keeps criminal records for citizens (an example instance is shown in Figure 7.1). Malice recently got convicted of fraud and decided to abuse her privileges and delete her criminal record by running \texttt{DELETE FROM Record WHERE name = 'Malice'}. However, she is aware that database operations are subjected to regular audits to detect tampering with the highly sensitive data stored by the agency. To cover her tracks, Malice deactivates the audit log before running the \texttt{DELETE} operation and afterwards activates the log again. Thus, there is no log trace of her illegal manipulation in the database. However, database storage on disk will still contain evidence of the deleted row (until several storage artifacts caused by the deleted are physically overwritten). Our approach detects traces of deleted and outdated record versions and matches them against the audit log to detect such attacks and provide evidence for how the database was manipulated. Using our approach, we would detect the deleted row and since it does not correspond to any operation in the audit log we would flag it as a potential evidence of tampering.

In Section 7.3 we showcase, for several databases, how an attacker like Malice can ensure that her operations are not being included in the audit log. Given that it is possible for a privileged attacker to erase log evidence and avoid detection, the challenge is to detect such tampering and collect additional evidence about the nature of the malicious operations (e.g., recover rows deleted by a malicious operation). It may not be immediately clear that this recovery of evidence is possible at all. However, any operation leaves footprints in database storage on disk (writes) or in RAM (both reads and writes). For instance, DBMSes mark a deleted row rather than overwrite it. Thus, if we recover such evidence directly from storage then, at least for some amount of time until the deleted value is overwritten by future
inserts, we would be able to detect that there exists a discrepancy between the content of the audit log and database storage.

Given that evidence of operations exists in database storage, the next logical question to ask is whether Malice can remove this evidence by modifying database files directly. While a user with sufficient OS privileges may be able to modify database files, it is extremely challenging to tamper with database storage directly without causing failures (e.g., DBMS crashes). Direct manipulation of DBMS files will uncover the tampering attempt because: 1) in addition to the actual record data on a page, the database system maintains additional references to that record (e.g., in index structures and page headers). Deleting a record from a page without modifying auxiliary structures accordingly will leave the database in an inconsistent state and will lead to crashes; 2) databases have built-in mechanisms to detect errors in storage, e.g., checksums of disk pages. A tampering attempt has to correctly account for all of these mechanisms; 3) incorrect storage for a value can corrupt a database file. To directly modify a value, an attacker needs to know how the DBMS stores datatypes.

Because it is not only hard but, at times, next to impossible to spoof database storage, it follows that database storage can provide us with valuable evidence of attacks. We use an existing forensic tool called DBCarver [91] to reconstruct database storage. However, we are still left with the problem of matching recovered artifacts to queries in audit log – doing so requires a thorough analysis of how database storage behaves. Our approach automatically detects potential attacks by matching extracted storage entries and reporting any artifacts that cannot be explained by logged operations (summarized in Figure 7.2). Our method is designed to be both general (i.e., applicable to any relational database) and independent (i.e., entirely outside of DBMS control). Our system DBDetective inspects database storage and RAM snapshots and compares what it finds to the audit log; the analysis of this data is then done out of core without affecting database operations. DBDetective can operate on a single snapshot from disk or RAM (i.e., multiple snapshots are not required), but additional snapshots provide extra evidence and improve detection quality. Data that has changed between two snapshots need be matched only.
against audit log entries of commands that were executed during the time span between these snapshots. Thus, more frequent snapshots increase the detection accuracy because it is less likely to match a row against an incorrect operation and the probability that deleted rows are still present is higher. Moreover, frequency of snapshots increase the performance of detection because a smaller number of recovered rows have to be matched against a smaller number of operations. We can reduce storage requirements by only storing deltas between snapshots in the same fashion as incremental backups are used to avoid the storage overhead of full backups.

Our focus is on identifying the likelihood of database tampering, as well as pointing out specific inconsistencies found in database storage. Determining the identity of the party responsible for database tampering is beyond the scope of this chapter. Due to the volatile nature of database storage, it is not possible to guarantee that all attacks will be discovered. We will discuss how false negatives or positives can occur for particular types of tampering in Sections 7.4 and 7.5. It may sound unsatisfactory that we are not able to detect all attacks. However, these types of attack bypass the database audit log and thus have no chance of being detected natively.

In this chapter, we demonstrate techniques to detect and identify database operations that were masked by the perpetrator through use of our system DBDetective. For each of the major DBMSes we evaluated, we assumed that the DBMS has enabled an audit log to capture SQL commands that are relevant to an investigation. We further assumed an attacker who found a way to prevent logging of executed malicious commands by: a) deactivating audit policies and temporarily suspending logging or b) altering the existing audit log (both discussed in Section 7.3).

By applying forensic analysis techniques to database storage or buffer cache and matching evidence uncovered by these techniques against the audit log, we can:

- Detect multiple types of database access and manipulation that do not appear in the DBMS audit log.
- Classify unattributed record modifications as an obfuscated `INSERT`, `DELETE`, or `UPDATE` command.
- Detect cached data from (read-only) `SELECT` queries that cannot be derived from activity in the audit log.

The rest of the chapter is organized as follows: Section 7.2 covers related work. Section 7.3 discusses DBMS logging mechanisms and how operations can be hidden from logs by an attacker. Section 7.4 details how table modifications that are missing from the log
files can be identified in storage. Section 7.5 discusses how read-only (SELECT) queries hidden from the logs can be detected based on memory snapshots. We evaluate our system in Section 7.6.

7.2 Related Work

7.2.1 Database Forensics

Database page carving [91] is a method for reconstructing the contents of a relational database without relying on file system or DBMS metadata. Database carving is similar to traditional file carving [25, 73] in that data, including deleted data, can be reconstructed from images or RAM snapshots without the use of a live system. The work in [86] presented a comparative study of the page structure of multiple DBMSes. Subsequent work [87] described how long forensic evidence resides within a database even after being deleted and defragmented. While a multitude of built-in and 3rd party recovery tools (e.g., [55, 65, 68]) can extract database storage, none of these tools are helpful for independent audit purposes because they only recover “active” data. A forensic database tool (just like a forensic file system tool) should also reconstruct unallocated pieces of data, including deleted rows, auxiliary structures (indexes) or buffer cache space.

7.2.2 Database Auditing and Security

Peha used one-way hash functions to verify an audit log and detect tampering [64]. They relied on an external, trusted notary to keep track of every transaction. Snodgrass et al. also used a one-way hash function to validate audit logs [80]. Alternatively, their hash function uses the record itself and a last modification timestamp, avoiding the external notary. Pavlou et al. expanded this work by determining when audit log tampering occurred [63]. While this mechanism ensures an accurate audit log with high probability by sending the secure hashes to a notarization service, it is ultimately useless if logging has been disabled by a privileged user. Our approach detects log tampering even if logs files have been disabled.

Sinha et al. used hash chains to verify log integrity in an offline environment [79]. In this situation, communication with a central server is not required to ensure log authenticity. Crosby et al. proposed a data structure called a history tree to reduce the log size produced by hash chains in an offline environment [13]. Rather than detecting log tampering, Schneier and Kelsey made log files impossible to read and impossible to modify [77].
Under this framework, an attacker does not know if his activity has been logged, or which log entries are related to his activity.

An event log can be generated using triggers, and the idea of a `SELECT` trigger was explored for the purpose of logging [19]. This would allow all table access to be logged – but a malicious user could also utilize triggers to remove traces of her activity or simply bypass a `SELECT` trigger by creating a temporary view to access the data.

ManageEngine’s EventLog Analyzer [48] provides audit log reports and alerts for Oracle and SQL Server based on actions, such as user activity, record modification, schema alteration, and read-only queries. However, the Eventlog Analyzer creates these reports based on native DBMS logs. Like other forensic tools, this tool is vulnerable to a privileged user who has the ability to alter or disable logs.

Network-based monitoring methods have received significant attention in audit logging research because they can provide independence and generality by residing outside of the DBMS. IBM Security Guardium Express Activity Monitor for Databases [37] monitors incoming packets for suspicious activity. If malicious activity is suspected, this tool can block database access for that command or user. Liu et al. [46] monitored DBAs and other users with privileged access. Their method identifies and logs network packets containing SQL statements.

The benefit of monitoring activity over the network and, therefore, beyond the reach of DBA’s, is the level of independence achieved by these tools. On the other hand, relying on network activity ignores local connections to the DBMS and requires intimate understanding of SQL commands (i.e., an obfuscated command could fool the system). By contrast, our approach detects both local and network activity because SQL is ultimately run against the database instance affecting database storage state.

### 7.3 Reliability of Database Logs

An attacker can alter two types of logs to interfere with an investigation: write-ahead logs (WAL) and audit logs (event history records). WALs record database modifications at a low level in order to support ACID guarantees, providing a history of recent table modifications. Audit logs record configured user database actions, including SQL operations and other user activity.

**WALs** WALs cannot normally be disabled or easily modified, and require a special-purpose tool to be read (e.g., Oracle LogMiner or PostgreSQL pg_xlogdump). Some
DBMSes allow WALs to be disabled for specific operations, such as bulk load or structure rebuild. Thus inserting records without leaving a log trace can be done through this feature. Since the WAL file format is not human-readable, and requires specific tools for parsing, this would seem to protect it from tampering. However, DBMSes (including Oracle, MySQL, PostgreSQL, and SQL Server) allow the administrator to switch to a new WAL file and delete old WAL files. Therefore, executing a WAL switch and deleting the original WAL can effectively allow a user to perform transactions without leaving a WAL record. For example, an administrator could switch from log file A to log file B, perform the malicious SQL operation(s), switch back to log file A (or a new log file C), and delete log file B. For example, to implement this operation in Oracle:

1) `ALTER DATABASE ADD LOGFILE ('path/logB.rdo')`
2) `ALTER SYSTEM SWITCH LOGFILE`
3) Run the malicious SQL operation(s)
4) `ALTER SYSTEM SWITCH LOGFILE`
5) `ALTER DATABASE DROP LOGFILE MEMBER 'path/logB.rdo'`

Audit Logs  Audit logs store executed SQL commands based on logging policies that are configured by database administrators. Therefore, an administrator can disable logging or modify individual log records as they see fit. For example, records in the Oracle `sys.aud$` table can be modified with SQL commands, and records in the PostgreSQL `pg_audit` log and MySQL general query log are stored as human-readable text files. Table 7.1 summarizes how to modify the audit log for three major DBMSes.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>SQL commands against <code>sys.aud$</code></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Edit files in the <code>pg_log</code> directory</td>
</tr>
<tr>
<td>MySQL</td>
<td>Edit the <code>general_log_file</code></td>
</tr>
</tbody>
</table>

Table 7.1: Commands to edit the audit log.

7.4 Detecting Hidden Record Modifications

When a table record is inserted or modified, a cascade of storage changes occurs in the database. In addition to the affected record’s data itself, page metadata may be updated (e.g., a delete mark is set) and page(s) of an index storing the record may change (e.g., to reflect the deletion of a record). Each of the accessed pages would be brought into RAM if it is not already cached. Row identifiers and structure identifiers can be used to tie all
of these changes together. Furthermore, DBAs can also disable logging for bulk modifications (for performance considerations); this privilege can be exploited to hide malicious modifications. In this section, we describe how we detect inconsistencies between modified records and logged commands.

### 7.4.1 Deleted Records

Deleted records are not physically erased but rather marked as “deleted” in the page; the storage occupied by the deleted row becomes unallocated space, and eventually will be overwritten by a new row. Unlike audit log records, these alterations to database storage cannot be bypassed or controlled – thus if a reconstructed deleted record does not match the \texttt{WHERE}-clause condition of any delete statement in the audit log, then a log record is missing.

\texttt{DBCarver} returns the status of each row as either “deleted” or “active.” Reconstructed deleted rows and the audit log are used in Algorithm \ref{algo:deleted-records} to determine if a deleted row can be matched with at least one \texttt{DELETE} command. Here we use \texttt{cond(d)} to denote the condition of delete \texttt{d}. The conditions of delete operations may overlap, potentially creating false-negative matches (i.e., a delete’s condition may match a row that was \textit{already} deleted by another \texttt{DELETE}). However, we are interested in identifying deleted rows in storage that do not match any delete operation in the log. A false-negative match presents a problem if it hides a missing match with a delete that the attacker attempted to hide. Only if \textit{all} reconstructed deleted rows that the attacker attempted to hide have false-negative matches will the attack go unnoticed, because a single unaccounted for deleted record is sufficient to detect suspicious activity.

\begin{algorithm}
\caption{Accounting for Deleted Records in Log Files}
\begin{algorithmic}[1]
\State \texttt{Deletes} $\leftarrow$ \texttt{DELETE} statements in audit log
\State \texttt{DelRows} $\leftarrow$ deleted records reconstructed by \texttt{DBCarver}
\State \texttt{Unaccounted} $\leftarrow$ \texttt{DeletedRows}
\For{\texttt{d} $\in$ \texttt{Deletes}}
\For{\texttt{r} $\in$ \texttt{DelRows}}
\If{\texttt{r} $\models$ \texttt{cond(d)}}
\State \texttt{Unaccounted} $\leftarrow$ \texttt{Unaccounted} $-$ \{\texttt{r}\}
\EndIf
\EndFor
\EndFor
\State \textbf{return} \texttt{Unaccounted}
\end{algorithmic}
\end{algorithm}

Figure \ref{fig:deleted-rows} gives an example for detecting unaccounted deleted rows. \texttt{DBCarver} reconstructed three deleted rows from the \texttt{Customer} table: (1,Christine,Chicago), (3,Christopher, Seattle), and (4,Thomas,Austin). The log file contains two operations:

\begin{verbatim}
T1 DELETE FROM Customer WHERE City = 'Chicago'
\end{verbatim}
In Algorithm 2, DeletedRows was set to the three reconstructed deleted rows. Algorithm 2 returned (4, Thomas, Austin), indicating that this deleted record could not be attributed to any of the deletes. We cannot decide which operation caused deletion of (1, Christine, Chicago) row (T1 or T2), but that is not necessary for our purpose of finding that record #4 is an unattributed delete.

![Figure 7.3: Detection of unattributed deleted records.](image)

### 7.4.2 Inserted Records

New inserted rows are either appended to the end of the last page (or a new page if the last page is full) of a table or overwrite free space created by previously deleted rows. A new row has to be smaller than or equal to the old deleted row to overwrite its previous storage location; some databases (Oracle and PostgreSQL) explicitly delay the overwriting unallocated page space. When an inserted row is smaller than the deleted row, only a part of the deleted row is overwritten leaving traces of the old row behind. If an “active” new table row does not match any of the insert operations from the audit log, then this row is a sign of suspicious activity. These “active” records are used in Algorithm 3 to determine if a reconstructed row can be attributed to an insert from the audit log.

**Algorithm 3** Accounting for Inserted Data in Log Files

1: $\text{InsertedRows} \leftarrow$ rows created by \text{INSERT} log entries
2: $\text{Rows} \leftarrow$ reconstructed active rows
3: $\text{Unaccounted} \leftarrow \text{Rows}$
4: for each $r \in \text{Rows}$ do
5: \hspace{1em} if $r \in \text{InsertedRows}$ then
6: \hspace{2em} $\text{Unaccounted} \leftarrow \text{Unaccounted} - \{r\}$
7: return $\text{Unaccounted}$
Figure 7.4 shows an example for detecting an **INSERT** operation that does not match any commands in the audit log. The log contains six operations. As rows are inserted from T1 to T4, they are appended to the end of the table. At T5, (3, Lamp) was deleted followed by an insert of (5, Bookcase) at T6. Since row (5, Bookcase) is larger than the deleted row (3, Lamp), it is appended to the end of the table. **DBCarver** reconstructed five active records, including (0, Dog) and (2, Monkey). **Rows** was initialized to the five reconstructed active rows for Algorithm 3. Algorithm 3 thus returned (0, Dog) and (2, Monkey) because these records could not be matched to logged inserts (only the latter is an **INSERT** as we will see in Section 7.4.3). The character p found with (0, Dog) was not part of the record, indicating that this record overwrote a previously deleted row. Since (0, Dog) is one character smaller than (3, Lamp) and the last character from (3, Lamp) was found, it was likely that (0, Dog) overwrote the deleted record (3, Lamp). We describe how to confirm this in Section 7.4.4.

<table>
<thead>
<tr>
<th>Log File</th>
<th>DBCarver Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, INSERT INTO Furniture VALUES (1, ‘Chair’);</td>
<td>✔ 1, Chair</td>
</tr>
<tr>
<td>T2, INSERT INTO Furniture VALUES (2, ‘Desk’);</td>
<td>✗ 2, Desk</td>
</tr>
<tr>
<td>T3, INSERT INTO Furniture VALUES (3, ‘Lamp’);</td>
<td>✔ 0, Dogp</td>
</tr>
<tr>
<td>T4, INSERT INTO Furniture VALUES (4, ‘Dresser’);</td>
<td>✔ 4, Dresser</td>
</tr>
<tr>
<td>T5, DELETE FROM Furniture WHERE Name LIKE ‘Lamp’;</td>
<td>✔ 5, Bookcase</td>
</tr>
<tr>
<td>T6, INSERT INTO Furniture VALUES (5, ’Bookcase’);</td>
<td>✗ 2, Monkey</td>
</tr>
</tbody>
</table>

**Figure 7.4:** Detection of unattributed inserted and updated records.

### 7.4.3 Updated Records

An **UPDATE** operation is essentially a **DELETE** operation followed by an **INSERT** operation. To account for updated rows, we use unmarked deleted rows returned by Algorithm 2 and unmarked inserted rows returned by Algorithm 3 as the input for Algorithm 4. If a deleted row can be matched to the **WHERE** clause of an update, then this deleted row operation is marked as present in the log. Next, if an unmarked inserted row can be matched to the value from the **SET** clause, and the inserted row matches all values in the deleted row except for the **SET** clause value, then this inserted row operation is present in the log. Currently,
our implementation is limited to simple set clause expressions of the form $A = c$ for an attribute $A$ and constant $c$. In the algorithm, we use $cond(u)$ for an update $u$ to denote the update’s where clause condition and $set(u)$ to denote the its set clause. Furthermore, we use $APPLY(r,s)$ to denote evaluating set-clause $s$ over row $r$.

**Algorithm 4** Accounting for Updated Data in Log Files

1. $\text{Deleted} \leftarrow \text{unmarked deleted rows from Alg. 2}$
2. $\text{Inserted} \leftarrow \text{unmarked inserted rows from Alg. 3}$
3. $\text{Updates} \leftarrow$ set of updates from the audit log
4. for all $r_{del} \in \text{Deleted}$ do
5. if $\exists u \in \text{Updates} : r_{del} \models cond(u)$ then
6. $\text{Deleted} \leftarrow \text{Deleted} - \{r_{del}\}$
7. for all $r_{ins} \in \text{Inserted}$ do
8. if $APPLY(r_{del}, set(u)) = r_{ins}$ then
9. $\text{Inserted} \leftarrow \text{Inserted} - \{r_{ins}\}$
10. return $\text{Deleted}$, $\text{Inserted}$

Figure 7.4 also shows an example of how we detect an update operation not present in the log. Algorithm 2 returned the row $(2,\text{Desk})$, and Algorithm 3 returned the row $(0,\text{Dog})$ and $(2,\text{Monkey})$. Using these sets of records, Algorithm 4 returned $(2,\text{Desk})$ as the list of deleted records, and $(0,\text{Dog})$ and $(2,\text{Monkey})$ as the list of inserted records. Additionally, Algorithm 4 recognized the shared value, $2$, for the first column in $(2,\text{Desk})$ and $(2,\text{Monkey})$. While this does not confirm an update operation by itself, it is reasonable to conclude that $(2,\text{Desk})$ was updated to $(2,\text{Monkey})$.

### 7.4.4 Indexes

In some cases, records from table pages are insufficient to draw reliable conclusions about record modification. For example, in Figure 7.4 we did not have enough information to confirm that $(3,\text{Lamp})$ was overwritten by $(0,\text{Dog})$. Reconstructed index pages provide additional information because deleted index values have a significantly longer lifetime compared to deleted records themselves [87]. Using the pointer associated with deleted (but still recoverable) index entry allows us to determine values previously stored at a particular location within a page.

Figure 7.5 demonstrates how old index values supply evidence of a deleted record that was overwritten by new values. The index stores the furniture table ID and a pointer to the row address. Using index pointers, we can be certain that the overwritten row once stored record with ID of $3$. This allows us to extrapolate a partial deleted record, $(3, \_ \_ \_)$, that we can include in Algorithms 2 and 4. If a secondary index on the second column (furniture name) is available, we could also extrapolate Lamp from the index.
DBMSes use a component called buffer manager to cache pages from disk into memory. Data is read into the buffer pool in units of pages, that can be reconstructed by DBCarver. In this section, we describe how artifacts carved from the buffer pool can be matched to read-only queries in the audit log. A database query may use one of two possible ways of accessing a table: a full table scan (FTS) or an index scan (IS). An FTS reads all table pages, while an IS uses an index structure (e.g., B-Tree) to retrieve a list of pointers referencing particular table pages (or rows) to be read based on a search key. All accessed index pages and some of the table pages (depending on access type) are placed in the buffer pool by the DBMS.

### 7.5.1 Full Table Scan

When a query uses an FTS, only a small part of a large table will be cached. A small table (relative to the buffer pool size) may be cached in its entirety. Every database stores a unique page identifier within the page header which allows us to efficiently match cached pages to their counterpart on disk. The particular number of pages cached by a FTS can be derived from the size of the table, although it is not always proportional (e.g., a larger table may result in fewer cached pages). Thus, after FTS is executed, typically pages from the physical end of table storage would be in the cache (i.e., a few pages including the one where new inserts would be appended). In Section 7.6.3 we analyze caching behaviour for multiple DBMSes.

Figure 7.6 provides an example of an FTS over the Employee table. We can identify pages that belong to Employee by the structure identifier 131, which is stored in the page header. DBCarver can return just the page structure identifiers (without parsing page
content) at a much faster speed. Both Q2 and Q4 access Employee via an FTS. Each time the Employee table is scanned, the same four pages (identifiers: 97, 98, 99, and 100) from the table are loaded into the buffer pool. Therefore, when four pages with the page identifiers 97, 98, 99, and 100 and a structure identifier of 131 are found in memory, a FTS on the Employee table can be assumed.

### 7.5.2 Index Access

DMBSes use IS to optimize performance for queries that access data based on the key attributes of an index. Caching of index pages identifies what attribute was queried (a query posed conditions over this attribute) and provides a rough estimate of what value range was selected for an indexed attribute (since values stored in index pages are ordered). Cached index pages are more precise in determining what the query accessed because cached table pages contain the entire table row (regardless of which columns were accessed), but index pages contain only the relevant columns. A sequence of index pages in the buffer pool that does not correspond to any logged query can present evidence of hidden access.

Algorithm 5 describes how to use the minimum and maximum values of index pages to determine if a cached index page can be attributed to logged query. Again, cond(q) denotes the conditions used by query q (OR’ed together).

Figure 7.6 shows examples of index accesses on the Customer table. The Customer table’s structure identifier is 124, and the secondary index on the C_City column has a structure identifier of 126. Q1 filters on the city Dallas, and it caches the index page with identifier 2. This page has a minimum value of Chicago and a maximum value of Detroit. Q3 filter on the city Jackson, and it caches the index page with the page identifier of 4. This page has a minimum value of Houston and a maximum value of Lincoln. If a query
Algorithm 5 Accounting for Index-Access Queries

1: \textit{IndexPages} $\leftarrow$ set of all cached index pages
2: \textit{Queries} $\leftarrow$ queries from the audit log
3: \textbf{for each} $i \in \text{IndexPages}$ \textbf{do}
4: \hspace{1em} if $\exists q \in \text{Queries} : \exists r \in i : r \models \text{cond}(q)$ \textbf{then}
5: \hspace{2em} \text{IndexPages} $\leftarrow$ \text{IndexPages} $\setminus \{i\}$
6: \textbf{return} \text{IndexPages}

in the audit log filters on a values within the minimum and maximum range of values for an index page, then that page can be attributed to that query.

7.5.3 Data Lifetime in Memory

As new data is read into cache, old data is evicted (using a buffer replacement strategy such as LRU) providing us with an approximate timeline of recent accesses. A malicious user can not directly control the buffer pool; regardless of one’s permission level, there are no direct APIs to control buffer pool behavior. Assuming that the attacker cannot do something as conspicuous as powering down the computer, the only available command is to flush the cache (only available in Oracle, SQL Server and MySQL). Interestingly, flushing buffer cache will mark pages as unallocated instead of physically evicting any data from RAM.

7.6 Experiments

Our experiments use three databases (Oracle, PostgreSQL, and MySQL) that we consider representative (both open- and closed-source, all three very widely used) due to space limitations. We have used data and queries from TPCC [43, 71] and SSBM [59] benchmarks. These benchmarks were used because they were designed to measure the performance of DBMSes.

Our experiments were carried out on servers with an Intel X3470 2.93 GHz processor and 8GB of RAM running Windows Server 2008 R2 Enterprise SP1 or CentOS 6.5. Windows OS memory snapshots were generated using a tool called User Mode Process Dumper (version 8.1). We used regular SATA magnetic drives for storage and processing. Linux OS memory snapshots were generated by reading the process’ memory under /proc/$pid/mem.
7.6.1 Experiment 1: *DBDetective* Performance Evaluation

The objective of this experiment is to explore the costs associated with *DBDetective* and the estimated reaction time to detect tampering. In Part-A of this experiment, we provide cost estimates to perform memory snapshots. In Part-B, we test the carving process performance against database files. In Part-C, we test the carving speed against memory snapshots.

**Part A** To estimate the cost to perform memory snapshots, we copied a 2.5GB snapshot from an Oracle database process to a magnetic disk. This operation took approximately 31 sec. In practice, the snapshot cost is dominated by the cost of writing the result to disk but can be sped up significantly by shipping data to a remote machine or using a faster drive (e.g., PCIe). As long as snapshots are taken as often as the entire buffer pool is replaced by query activity, we expect to detect most activity.

**Part B** To obtain a performance estimate for file carving, we ran *DBCarver* tool against five Oracle database files ranging in size from 1MB to 3GB. All Oracle files contained 8KB database pages. We observed that *DBCarver* parsed the files at an average rate of 1.1 MB/s and continued to scale linearly with respect to the file size (using SATA magnetic disk).

**Part C** Finally, we tested the performance of the carving tool against memory snapshots of Oracle buffer cache. We collected a 2.5GB snapshot taken from the Oracle database process and an 8GB snapshot of the entire RAM content. Each of the snapshot required detecting and parsing the contents of roughly 80,000 pages (600MB). The 2.5GB snapshot was carved at a rate of 4.2 MB/s, and the 8GB snapshot was carved at a rate of 13.2 MB/s. We can thus conclude that the runtime of page parsing depends solely on the number of database pages rather than raw file size.

7.6.2 Record Modification Detection

7.6.2.1 Experiment 2: Deleted Record Detection

The objective of this experiment is to identify deleted rows from storage that could not be matched to commands in the log files. We also evaluate the situation where a row deleted by a malicious query was overwritten or was attributed to a non-malicious query (a false-negative match).
Part A  For this experiment we use MySQL. By default, MySQL creates an IOT when a primary key is declared for a table. MySQL uses the primary key as the row identifier, and all rows are physically ordered within index (leaf) pages by the row identifier. If no primary key is declared, MySQL will synthesize a unique row identifier for each row. MySQL stores the row identifier as the pointer in the index value-pointer pairs.

We initially started with the Item table (100K records) from the TPCC benchmark. We created a primary key on the I_ID column, a secondary index on the I_Name column, and a secondary index on the I_IM_ID column. Next, we issued two delete commands:

(Delete 1) `DELETE FROM Item WHERE I_Name LIKE ‘w2G%'`
(Delete 2) `DELETE FROM Item WHERE I_IM_ID = 8563`.

Delete 1 represents malicious activity, and was therefore removed from the log. Delete 1 deleted records with the I_ID values of 92328 and 95136. Delete 2 is in the log and was responsible for deletion of 10 records. We used DBCarver to reconstruct deleted rows from Item in storage: and 12 deleted rows were reconstructed.

Algorithm 2 returned one record with the I_ID value of 92328. 11 of the deleted records were matched with the logged Delete 2 command: the 10 records it deleted and the record with I_ID 95136. Even though the 11th record was caused by Delete 1, it resulted in false-negative match to Delete 2 because it happened to have a I_IM_ID value of 8563. However, false-negatives are only problematic if they prevent all maliciously deleted records to be detected.

Part B  Realistically, investigators may not be able to perform forensic analysis at the most opportune time. We next consider what determination can be made if the trace of the maliciously deleted record has been overwritten.

To instrument an overwrite of a deleted record in an IOT, a record with the same primary key value had to be inserted. We inserted the record `(92328,100,DBCaver1,0.0, This is a trick1)`. The original deleted record with the I_ID value of 92328 was permanently overwritten. However, the secondary indexes on I_Name and I_IM_ID columns retain traces of this record until something causes an index rebuild. The pointers stored with index values are the row identifiers (or primary key) for table records.

We found that the row identifier 92328 had two entries in the I_Name index: the value for the current (new) record, w2GSyVRavpUbCr2bEzqOb for the old record, and two entries in the I_IM_ID index: the value for the current record and 4763 for the overwritten record. This allowed us to extrapolate a partial deleted record as an input to Algorithm 2 `(92328,4763,w2GSyVRavpUbCr2bEzqOb,?,?)`. Since Algorithm 2 could not match
the partial record to any of the logged commands, it also provides evidence of the missing log record.

7.6.2.2 Experiment 3: Updated Record Detection

The objective of this experiment is to identify the by-product of an UPDATE operation in persistent storage that can not be matched to commands in the log. Similar to Experiment 7.6.2.1-B, we evaluated records that were overwritten by an in-place UPDATE.

Part A We again use MySQL and the Item table with 100K rows and indexes defined as in previous experiments. Records in Item include (53732, 1004, Name\textsuperscript{Val}_53732, 14.55, Data\textsuperscript{Val}_53732) where Name\textsuperscript{Val}_53732 is Us65fCVCfROMDT6bpDDE and Data\textsuperscript{Val}_53732 is mPSxHpzo2ftrF2P0rXpZhdYSakGcqrSqeL6a6p2cE4Q. All of INSERT commands creating the table were logged. Next, we issued an update, 

\texttt{UPDATE Item SET I\_Name = 'DBCarver' WHERE I\_ID = 53732}

to simulate malicious activity, and removed this operation from the log. We then passed the database files containing the Item table and the I\_Name secondary index to DBCarver. Algorithm 3 returned the record (53732, 1004, DBCarver, 14.55, Data\textsuperscript{Val}_53732) since it does not match any logged INSERT command. DBCarver did not return deleted rows because when the row was updated, the new version of the row physically overwrote the old version. Two pieces of evidence help classify the row 53732 as an overwrite of a deleted row: table pages and the pages for the index on I\_Name. In the table page, the new row used less storage space than the old overwritten row. Therefore, part of the old row was still present – 13 characters from the last column were reconstructed:
mPSxHpzo2ftrF2P0rXpZhdYSakGcqrSqeL6a6p2cE4Q

These 13 characters could be distinguished from new row because new row metadata specifies where the current row ends. This behavior is illustrated in Figure 7.4. In the secondary index page, the pointer (or row identifier) 53732 had two entries, both with the new value (DBCarver) and the old value (Name\textsuperscript{Val}_53732). Since the value DBCarver was present in the current active record, we could assert that DBCarver overwrote Name\textsuperscript{Val}_53732. This allowed us to extrapolate a partial pre-update record, (53732, ?, Name\textsuperscript{Val}_53732, ?, ?) despite the fact that it was destroyed.

Part B Having detected unmatched active record (53732, 1004, DBCarver, 14.55, Data\textsuperscript{Val}_53732), and a partially reconstructed deleted record, (53732, ?, Name\textsuperscript{Val}_53732, ?, ?), we can link them as evidence of an update in Algorithm 4. First, we use Algorithm 2

113
which returned our partially deleted record as not a non-match. We next added our partially deleted record 53732 to **Deleted** and our active record to **Inserted** in Algorithm 4.

Algorithm 4 returned (53732, 1004, DBCarver, 14.55, Data_Val53732) as an active record and (53732, ?, Name_Val53732, ?, ?) as a deleted record. Since they share the 53732 primary key value, it is reasonable to conclude that these records should match with an **UPDATE** command, rather than both a **DELETE** and **INSERT**. Technically, this behavior could be caused by a hidden combination of **DELETE** and **INSERT** – either way, we uncovered a maliciously hidden modification. We can also determine that the third column was changed from Name_Val53732 to DBCarver.

### 7.6.2.3 Experiment 4: Modified Record Detection

We now explore the objectives of Experiments 1 and 2 in an Oracle setting. In Part A of this experiment, we identify the by-products of **DELETE** and **UPDATE** commands in storage that do not match any logged operations. In Part B, we simulated a scenario in which deleted records are overwritten. We then determined what malicious **DELETE** and **INSERT** commands could still be detected. In Part C, we used available indexes and results from Part B to match **UPDATE** operations.

Unlike MySQL, Oracle does not create an IOT by default when a primary key is declared (IOTs must be created explicitly). Instead, a regular B-Tree index is created on the primary key. Without IOT, unique row identifiers are not stored with each row. Instead, Oracle uses physical row identifiers consisting of a structure identifier, page identifier, and row’s position within the page.

#### Part A

We use the TPCC NewOrders (NO_O_ID, NO_D_ID, NO_W_ID) table with 9K rows. We declared a primary key on the NO_O_ID column and a secondary index on the NO_D_ID column. Next, we issued the following queries to simulate malicious activity:

(Command 1) **DELETE FROM** New_Orders  
WHERE NO_O_ID = 2500 AND NO_D_ID = 1

(Command 2) **UPDATE** New_Orders SET NO_D_ID = 777  
WHERE NO_O_ID = 2700 AND NO_D_ID = 1.

We removed both Command 1 and Command 2 from the log. We then passed the database files containing the NewOrders table and both indexes to DBCarver.

We reconstructed the deleted record (2500, 1, 1) caused by Command 1. A copy of the indexed values for this record were reconstructed from the primary and secondary index. DBCarver also reconstructed the active record (2700, 777, 1) – Command 2 caused an
in-place update and overwrote the old version, (2700, 1, 1). However, the old NO_D_ID value is still present in the index, and could be mapped back to the overwritten row.

Part B To continue this experiment, we simulated normal database activity to observe what causes commands from 7.6.2.3-A to be no longer immediately detectable. This was done by repeatedly deleting 10 records using the NO_O_ID column, and inserting 20 records. We passed the database files containing the NewOrders table and indexes to DBCarver after each set of operations. We passed the carved output to Algorithms 2 and 3 after each set of operations.

After the first and second sequence of 30 commands, Algorithm 2 returned (2500, 1, 1), and Algorithm 3 returned (2700, 777, 1). This meant that we had detected a DELETE command and an INSERT command missing from the log file. After the third set of commands, Algorithm 2 did not return any records because (2500, 1, 1) was overwritten by an inserted record, and Algorithm 3 returned (2700, 777, 1). Now, only an INSERT command was only detected as missing from the log file.

Part C While we detected missing operations during our simulation, we wanted to see if indexes can serve as an extra source of evidence of malicious activity. The unidentified DELETE command was no longer detected after the third set of database activity commands, and the unidentified INSERT command could have actually been an in-place update that we demonstrated in Experiment 7.6.2.2.

The third set of database activity commands overwrote the deleted record of interest, seemingly avoiding detection. However, we found multiple values for the pointer to this record in both the primary key index and the secondary index. We then reconstructed a partial deleted record using the index values that weren’t found in the current record: (2500, 1, ?). Algorithm 2 did not associate this partial record with any DELETE command in the log file since all of the DELETE commands were on the primary key. Therefore, we had found evidence of a DELETE operation not recorded in the log files.

Throughout all of the database activity, we detected that the record (2700, 777, 1) was part of an INSERT command removed from the log files. However, more conclusions could be derived from the index values. We found the one value for the pointer in the primary key index, but we found two values for the same pointer in the secondary index. This indicated that the record was likely updated by a previous command. Given the one value in the primary key index and the two values in the secondary index, we could reconstruct the partial deleted record: (2700, 1, ?). Finally, Algorithm 4 identified the commonality, 2700, between the unattributed active record, (2700, 777, 1), and the partial deleted record, (2700,
1, ?). Based on this result, it was reasonable to assume that the record with the NO_O_ID value of 2700 was involved in a hidden UPDATE command.

### 7.6.3 Read-Only Query Detection

#### 7.6.3.1 Experiment 5: Full Table Scan Detection

**Part A** The objective of this experiment is to demonstrate full table scan (FTS) detection. FTSes leave a consistent pattern of pages in the buffer cache for each table they access which can be detected in RAM.

We used a PostgreSQL DBMS with 8KB pages and a buffer cache of 128MB (or 16,000 pages). We evaluated FTS for two tables: the Item table (1284 pages) from the TPCC benchmark and the LineOrder table (77K pages) from the SSBM. To do this, we ran three queries that all used an FTS. The first query accessed Item, and the second and third queries accessed LineOrder.

In Snapshot 1, we observed 32 pages from the Item table. The 32 pages reconstructed by DBCarver represented the 32 highest page identifiers for Item table (i.e., the last 32 pages in the physical database file), just as described in Figure 7.6. We verified that this is the case by inputting the Item database file into DBCarver. We did not observe any other cached table pages or cached index pages related to the Item table in the buffer cache. In Snapshot 2, DBCarver reconstructed the same 32 pages from Item and an additional 32 pages from LineOrder. The by-product from scanning Item was still detectable in memory, although it is unallocated space from DBMS’s perspective. Similar to the Item FTS, the 32 pages cached for LineOrder had the highest page identifiers from the database file where LineOrder was stored. For Snapshot 3, DBCarver returned 32 pages from Item and 64 pages from LineOrder. The Item pages were the same pages from Snapshots 1 and 2. The new set of 32 pages from LineOrder had the exact same page identifiers, found at a different location in the memory snapshot. Each FTS access demonstrated a consistent caching pattern in PostgreSQL, 32 pages for every table, producing a new set of pages at a location in memory adjacent to the previous pattern thereby creating a crude timeline of queries in buffer cache. Note that other DBMSes exhibit their own (consistent) caching pattern for an FTS. For example, the exact number of pages cached for a table in Oracle is not constant, but relies on a predictable pattern for each table.

**Part B** To demonstrate that FTS caching depends on buffer cache size, we increased buffer cache to 256MB in PostgreSQL and performed the same sequence of queries. As a result, we observed that the FTS(Item) query switched to caching the whole table (all
1284 pages). However, the FTS(LineOrder) query cached 32 pages each in the exact same pattern as before. In general, DBMSes use an internal heuristic threshold to decide when a whole table is “small enough” to be fully read into the buffer cache.

### 7.6.3.2 Experiment 6: Index Access Detection

The objective of this experiment is to demonstrate index access detection. When a table is accessed using an index, both the index pages and table pages are cached in memory. The ordered values stored in the index pages (leaves and intermediate nodes) provide a rough estimate of the range of values accessed by a query.

For this experiment, we used a PostgreSQL DBMS with 8KB pages and a buffer cache of 128MB (or 16,000 pages). We created the Item table with a secondary index on the I_NAME column. Next, we issued two queries that used an index access for the Item table:

(1) **Query 1**

```sql
SELECT * FROM Item
WHERE I_Name BETWEEN 'aa' AND 'ab'
```

(2) **Query 2**

```sql
SELECT * FROM Item
WHERE I_Name BETWEEN 'ba' AND 'bb'.
```

Query 1 selected 105 rows (0.08 selectivity) and Query 2 selected 109 rows (0.08 selectivity). After each query, we captured a cache snapshot that we passed to DBCarver.

DBCarver reconstructed 102 table pages and 2 leaf index pages from the memory snapshot after Query 1. Since Query 1 used a secondary index (the table is not organized on this column), almost every accessed row cached a new table page. DBCarver reconstructed 94 new table pages and 2 new index leaf pages from the memory snapshot after Query 2, while the pages cached by Query 1 remained in memory. Similar to Query 1, Query 2 cached a page for almost every row selected. Since the indexes stored ordered values, they provided an estimate of how the table was accessed. Table 7.2 summarizes the detailed breakdown of index page contents returned by DBCarver. Table 7.2 shows that a value range between ‘a6j3’ and ‘AaBD’ must have been read to cache index page 1, a value between ‘AaBd’ and ‘ac5U’ was accessed to cache index page 2, a value between ‘b76G’ and ‘bAGT’ must have been to read to cache index page 3, and a value between ‘BaGW’ and ‘bcDi’ was accessed to cache index page 4. These index value ranges matched to Query 1 and Query 2 in Algorithm 5.

117
Audit logs and other build-in DBMS security mechanisms are designed to detect or prevent malicious operations executed by an attacker. An inherent weakness of such mechanisms is that attackers with sufficient privileges can bypass them to hide their tracks. We present and thoroughly evaluate DBDetective, an approach for detecting database operations that were hidden by an attacker by removing them from the audit log and collecting evidence about what data was accessed and modified by an attacker. Our approach relies on forensic inspection of database storage and correlates this information with entries from an audit log to uncover evidence of malicious operations. Importantly, database storage is nearly impossible to spoof and, thus, is a much more reliable source of tampering evidence than, e.g., audit logs.

Given that storage snapshots provide incomplete information, we will explore probabilistic matching that determines the likelihood of a storage artifact being caused by the operations in the audit log, exploit additional constraints based on temporal ordering of operations, simulate partial histories of SQL commands from an audit log for more precise matching, and dynamically adapt the frequency of taking snapshots based on detected anomalies.
Chapter 8

File Tampering Detection

8.1 Introduction

DBMSes use a combination of defense and detection mechanisms to secure access to data. Defense mechanisms, such as access control, determine the data granularity and system access granted to different database users; defense mechanisms, such as audit logging, monitor all database activity. Regardless of the defense mechanisms, security breaches are still a legitimate concern – sometimes due to unintentional granting of extra access control and sometimes due to outright hacking, such as SQL injection. Security breaches are typically detected through analysis of audit logs. However, audit log analysis is unreliable to detect a breach that originated from privileged users.

Privileged users, by definition, already have the ability to control and modify access permissions. Therefore, audit logs fundamentally cannot be trusted to detect suspicious activity. Additionally, privileged users commonly have access to database files. Consider a system administrator who maliciously, acting as the root, edits a DBMS data file in a Hex editor or through a programming language, such as Python. The DBMS, unaware of external file write activity taking place outside its own programmatic access, cannot log it, and thus the tampering attack remains undetected.

Current DBMSes do not provide tools against insider threats – in general, a built-in security mechanism is vulnerable to insider attacks. While a DBMS will not be able to detect direct storage changes, file-level modifications potentially create inconsistencies within the auxiliary data structures maintained by a DBMS. Forensics tools that examine file contents can be used to detect such inconsistencies, and determine if insider threats have taken place. We proposed the first database forensic tool, DBCarver, that can be used to detect deleted data from database pages (Chapter 3). However, database forensic tools such as DBCarver merely extract forensic artifacts but do not search for inconsistencies within the data structures maintained by a DBMS.
In this chapter, we propose a system, DBStorageAuditor, that detects database file tampering by identifying inconsistencies in storage through a direct inspection of internal database structures. DBStorageAuditor utilizes existing database forensic techniques and expands them to extract additional necessary storage artifacts. These artifacts are then used to detect inconsistencies within indexes and between indexes and tables. The underlying premise of our approach is that all relational databases follow patterns in storage over which the privileged user has little or no control. We inspect these storage patterns to detect unusual activity. We motivate DBStorageAuditor through an example:

**Example 2** Malice is the system administrator for a shipping company, FriendlyShipping. Malice is bribed by a competing company to interfere with the orders going to Seattle. Malice does not have access to the DBMS, but she does have access to the server where the database files reside.

Malice writes a Python script that will open and directly modify the database file containing the Orders table. The script then opens the database file, finds all records containing the string ‘Seattle’, and explicitly overwrites entire records with the NULL ASCII character (decimal value 0).

Figure 8.1 illustrates the result of Malice’s script actions. Since the record was erased without the DBMS (API has never seen that command) all DBMS security was bypassed, and the operation was never recorded in the log file. When FriendlyShipping investigates the missing Seattle orders, the audit log can only explain deleted orders for (2, Chair, New York) and (6, Chair, Detroit). The audit logs contain no trace of the Seattle order being deleted because it was not deleted but rather wiped out externally.

To simplify in the above example, we have omitted some details of database file tampering, which we expand on later in Section 8.4. Barring those details in Example 1, the
value in the City index still exists in index storage even though the entire record is erased. Therefore, an inconsistency can be identified by mapping back the index value to the empty gap in table storage. The empty gap in table storage exists because a database only marks a record when it is deleted, and only overwrites the record with data from a newly inserted record. However, making the mapping from the index value to the associated record must be based on the behavioral rules of database storage, such as page and record layout. We use database forensic tools to understand database layout, and using that layout, perform the necessary mapping.

It is not impossible for a scrupulous system administrator to (i) tamper with the index and create a cascade of inconsistencies throughout the index structure, or (ii) for an attacker who has privileges to modify database files to acquire privileges to suspend or kill logging mechanisms at the operating system level if necessary, or (iii) for a knowledgeable adversary to easily avoid corrupting storage and keep checksum values consistent. However, in spite of increased level of threat, we repeatedly show that accurate knowledge about data layout can be used to gather evidence and prove if any malicious activity has taken place.

Previously we developed an approach to detect malicious activity when DBMS logging is disabled [85]. In this approach we analyzed unlogged activity (executed through a proper DBMS API) but strictly assumed that database files were not exposed to tampering. In this chapter, we address the tampering vulnerability where the database files are physically altered. Developing an auditing system for DBMSes is part of our larger goal to open up the database system and its storage to users, for performance and forensics investigation.

The rest of this chapter is organized as follows: Section 8.2 covers related work. Section 5.4 discusses concepts of database storage used throughout the chapter. Section 8.3 defines the adversary we seek to defend against. Section 8.4 details how to perform database file tampering. Section 8.5 provides an overview of DBStorageAuditor. Section 8.6 describes how we utilize database forensics. Section 8.7 addresses index tampering. Section 8.8 proposes a method to organize carved index output making our system scalable. Section 8.9 discusses how to detect file tampering using inconsistencies between carved index data and table data. Section 8.10 provides a thorough evaluation of our system.

### 8.2 Related Work

This chapter focuses on the detection of database file tampering. Therefore, we discuss work related to protecting DBMSes against privileged users as well as work that detects regular (non-DBMS) file tampering. We outline why existing file tampering and anti-forensic methods are inapplicable to database files.
8.2.1 Database Auditing and Security

Database audit log files are of great interest for database security because they can be used to determine whether data was compromised and what records were accessed. Methods to verify log integrity have been proposed to detect log file tampering [80] [64]. Pavlou et al. expanded upon this work to determine the time of log tampering [63]. Sinha et al. used hash chains to verify log integrity in an offline environment without requiring communication with a central server [79]. Crosby et al. proposed a data structure, history tree, to reduce the log size produced by hash chains in an offline environment [13]. Rather than detecting log tampering, Schneider and Kelsey developed an approach to make log files impossible to parse and alter [77]. An event log can be generated using triggers, and the idea of a \texttt{SELECT} trigger was explored for the purpose of logging [19]. ManageEngine’s EventLog Analyzer provides audit log reports and alerts for Oracle and SQL Server based on actions, such as user activity, record modification, schema alterations, and read-only queries [48]. We previously described a method to detect inconsistencies between storage and log files, allowing tampering detection when logging was disabled (i.e., when an operation was excluded from the log) [85]. All of this work assumes that database storage cannot be altered directly – an action which bypasses logging mechanisms.

Network-based monitoring methods have received attention in audit log research because they provide independence and generality by residing outside of the DBMS. IBM Security Guardium Express Activity Monitor for Databases [37] monitors incoming packets for suspicious activity. Liu et al. [46] monitored DBAs and other privileged users by identifying and logging network packets containing SQL statements. The benefit of monitoring activity over the network and, therefore, beyond the reach of DBA’s, is the level of independence achieved by these tools. On the other hand, relying on network activity ignores local DBMS connections and requires intimate understanding of SQL commands (i.e., an obfuscated command can fool the system).

8.2.2 Database Forensics

Stahlberg demonstrated the retention of deleted data and proposed techniques to erase data for a MySQL DBMS [82]. While this work was only ever implemented for MySQL, it validates our threat model by imposing custom DBMS file modifications.

Database page carving [91] is a method for reconstructing the contents of a relational database without relying on the file system or DBMS. Page carving is inspired by traditional file carving [73] [25], which reconstructs data (active and deleted) from disk images or RAM snapshots without the need for a live system. The work in [86] presented a comparative
study of the page structure for multiple DBMSes. Subsequent work in [87] described how long forensic evidence resides within a database even after being deleted or reorganized. While a multitude of built-in and third party recovery tools (e.g., [55, 65, 68]) aim to extract database storage, none of these tools are helpful for forensic analysis because they can only recover “active” data. Forensic tools, such as Sleuth Kit [7] and EnCASE Forensic [20], are commonly used by digital investigators to reconstruct file system data, but they are not capable of parsing database files. A database forensic tool (just like a forensic file system tool) should also reconstruct unallocated pieces of data, including deleted rows, auxiliary structures (indexes) or buffer cache space.

8.2.3 File Tampering and Anti-Forensics

One-way hash functions have been used to detect file tampering at the file system level [41, 28]. However, we expect database files to be regularly modified by legitimate operations. Distinguishing a malicious tampering operation and a legitimate SQL operation would be nearly impossible at the file system level without knowledge of metadata in DBMS storage. Authenticating cached data on untrusted publishers has been explored by Martel [49] and Tamassia [83]. Their threat model defends against an untrusted publisher that provides cached results working with a trusted DBMS and, while our work addresses an untrusted DBMS.

Anti-forensics is defined as a method that seeks to interfere with a forensic process [35]; file tampering threat model we address in this chapter exhibits anti-forensics behavioral properties. Two traditional anti-forensics techniques are data wiping and data hiding [21, 40]: 1) data wiping explicitly overwrites data to delete it rather than mark it as deleted, 2) data hiding seeks to hide the message itself. We are not aware of any existing literature that addresses anti-forensics within DBMSes [75]; we consider adding or erasing data through file tampering (that bypasses DBMS itself) to be the equivalent of anti-forensics for DBMSes.

8.3 Threat Model

In this section, we define the attack vectors, different possible adversary types, and the privileges we expect them to wield. We consider two types of privileged users: database administrator (DBA) and system administrator (SA). A DBA can issue privileged SQL commands against the DBMS including disabling logs or granting privileges to users. However, a DBA would not have administrative access to the server OS. The SA has administrative access to the server OS including the ability to suspend processes and read/write access to all files,
but no access to privileged SQL commands in the DBMS. The SA can still have a regular DB user account without affecting our assumptions.

Since a DBA can bypass DBMS defense mechanisms, detection mechanisms are best suited to identify anomalous behavior. An audit log containing a history of SQL commands is accepted as one of the best detection mechanisms for a DBMS. In Section 8.2, we discussed prior work designed to prevent audit log tampering and detect malicious behavior in the event that logging was disabled. In this chapter, we focus on a detection mechanism for a user often ignored in DBMS security, the SA.

The SA can bypass all DBMS security defense and detection mechanisms by reading and editing a database file with a tool other than the DBMS. For example, a SA could use Python to open a file and change the value ‘Hank’ to ‘Walt.’ In Section 8.4 we discuss additional steps that must be considered to successfully perform such an operation, but it can ultimately be achieved. Since this operation occurs outside of the DBMS, it bypasses all DBMS access control, and it will not be included any of the DBMS log files. Furthermore, one can assume that the SA would have the ability to suspend any logging mechanism in the server OS. Although changes to a file will also be recorded in the file system journal, the SA has the ability to turn off journaling to the file system by using tune2fs on Unix or the FSCTL_DELETE_USN_JOURNAL control code on NTFS (Windows). However, the file system must be shutdown first in order to prevent possible corruption. Therefore, the SA may have to effect a shutdown of the DBMS before making changes to the database files. The shutting down and restarting of the database instance and the system will generate events that are logged; however, as mentioned earlier, the SA can turn off system logging easily. Moreover, the SA could revise the DBMS log in order to hide evidence of the shutdown and restart. Hence, it would be somewhat involved but not difficult for a SA to cover his/her tracks when tampering with a DBMS file.

8.4 File Tampering

The threats to data we consider in this chapter occur at the OS level outside of DBMS control. In this section, we formulate the threat and introduce concepts and categories of tampering.

A DBMS allows users and administrators to access and modify data through an API. Access control guarantees that users will be limited to data they are privileged to access. In this section, we discuss how an adversary can perform file tampering. To limit the scope of this chapter, we assume that file tampering involves user data and not metadata (changing metadata can easily damage the DBMS but that will not alter any of its records).
We define user data as records created by the user or copies of record values that may reside in auxiliary structures (e.g., indexes). File tampering actions that we discuss in this section ultimately produce one of two results in storage: 1) **Extraneous data** is a record or a value that has been added through file tampering or 2) **Erased data** is a record that has been explicitly overwritten (rather than marked deleted by a command as described in Chapter 3).

Three things must be considered when performing database file tampering: 1) page checksum, 2) write lock on files, and 3) dirty pages. In Chapter 2, we discussed the functionality and placing of the page checksum. Figure 8.2 shows three different page alterations, in all of which the checksum is (also) updated. Some DBMS processes hold write locks on the database files. Therefore, tampering would require that the attacker release or otherwise bypass OS file write locks. DBMSes do not immediately write pages back to disk after they are modified in the buffer cache. That is significant because a maliciously altered page on disk can be overwritten when a dirty page is flushed to disk – or, alternatively, a dirty page could be altered directly in RAM instead (bypassing file locks that way).

**Write-Locks** The file locking system API, through the `fcntl` system call in Unix, is set up so that a process can prevent writes to (as well as reads from) a file that it has locked successfully. An attacker can potentially cause the process holding the lock, in this case the DBMS, to release the lock. Otherwise, a sophisticated attacker with root privileges can release the lock without involvement of the process by using kernel code. Once the lock is released, the attacker would lock the file, tamper with its content, and then release the lock. The DBMS would not receive any signal or other indication of the tampering and could continue to use the file as if it were locked after the attacker releases the lock. While the attacker holds the lock, however, DBMS access to the file would be suspended. In order to prevent the DBMS from discovering this condition, the attacker could suspend the DBMS process temporarily until the tampering has been completed. An attacker with root privileges could also mark memory used by the DBMS as shared and tamper directly with memory.

**Data Encryption** Different levels of encryption can be employed to protect database files, but they can ultimately be bypassed by an adversary with SA privileges. It is reasonable to assume that the SA would have the ability to decrypt any data that has been encrypted at the OS level. The SA would most likely not have the privileges to decrypt any internal database encryption. However, individual (value or record based encryption)
is still subject to tampering since the metadata describing the encrypted values is still readable. Furthermore, column-level encryption values are decrypted when they are read into memory making it possible to map the decrypted values in memory back to the encrypted values in persistent storage.

### 8.4.1 Value Modification

The first category of file tampering action we consider is value modification. Value modification is logically similar to a SQL `UPDATE` command; this type of tampering results in extraneous data. Storage space and value encoding (see Chapter 3) are the main considerations when modifying a value.

If a modified value requires the same storage space as the original entry, no metadata needs to be updated. If the newly modified value requires less storage than the original, then metadata needs to be modified, and other values in the record may need to be shifted. For example, many DBMSes explicitly store string sizes on page – e.g., changing ‘Hank’ to ‘Gus’ requires metadata value with the size of the string to be changed from 4 to 3. Furthermore, if the modified value is not the last column in the record, all other columns must be shifted by one byte. Only the columns in the modified record need to be shifted; other records in the page can remain as-is, leaving a gap (1 byte in our example). Shifting all other records in the page to close the gap would require all of the corresponding row directory addresses and relevant index pointers to be updated. If a value is modified to a value that requires more storage space, the old version of the record must be erased and the new version of the record must be appended to the table. These operations are discussed in the remainder of this section. Shifting the following records to accommodate a large value modification is not practical – unless the modified value happens to be in the last record on the page (and there is free space at the end of the page).

Figure 8.2.2 shows an example of a value changed to a smaller size. Since ‘Andy’ is one byte smaller than ‘Alice’, the column size must be changed from 5 to 4. Furthermore, the name is not the last column so next column (‘Austin’) is shifted by one byte, which overwrites the ‘e’ at the end of ‘Alice’ and leaves an unused ‘n’ character from ‘Austin’.

### 8.4.2 Record Addition

The next file tampering action we consider is new record addition, which is logically similar to a SQL `INSERT` command. This type of file tampering results in extraneous data generated within the DBMS. When adding a record to a file, metadata in the row data, row directory, and page header must be considered along with the correct value encodings.
When a record is appended to an existing page, the structure of the record must match
the proper active record structure for that DBMS. Section 5.4 discusses metadata that a
DBMS uses to store records. For the DBMS to recognize a newly added record, a pointer
must be appended to the page row directory. Finally, the free space pointer must be updated
and the active record count (if used by the DBMS in question) must be incremented.

Figure 8.2.3 shows an example of the record (‘Carl’, ‘Chicago’) added to the page. Along with the values themselves, additional metadata is included in the row data. The
size of each column, 4 and 7 bytes, is included, the column count, 2, and the row delimiter, 44. Next, a pointer, 8050, is added to the row directory, and the record count is updated to 3. Finally, the free space address is updated since the record was added to free space of the page.

Figure 8.3: Architecture of the DBStorageAuditor.

127
8.4.3 Record Wiping

The final tampering action category we discuss is record wiping. Record wiping is logically similar to a SQL `DELETE` command, except that it fully erases the record. A proper SQL `DELETE` command will merely mark a record as deleted; record wiping explicitly overwrites the record to destroy the data, even from a forensic recovery tool. Record wiping erases data with no forensic trace as there is no indication that a record existed in a place where it was overwritten. Wiping a record from a file is essentially the reverse operation of adding a record to a file: the metadata in the row data, row directory, and page header must all be altered.

When a record is overwritten in a page, the entire record (including the metadata) is overwritten with the NULL ASCII character (a decimal value of 0). Next, the row directory pointer must also be overwritten in the same way. Finally, the free space pointer must be updated and the active record count (if used by the DBMS) must be decremented.

Figure 8.2.4 shows an example of the record (‘Alice’, ‘Austin’) erased from the page. Every byte used for the values and their metadata (column sizes, column count, and row delimiter) is overwritten with the decimal value 0. The row directory address for that row is erased and the row directory is defragmented. Finally, the record count is updated to 1.

Record Removal  Rather than explicitly overwriting a record, the record metadata could also be marked to mimic a SQL `DELETE`. We define such changes as a record removal (versus record wiping). We do not address record removal in this chapter because such unlogged action can be detected by our previous work in [85] by comparing and flagging inconsistencies between DBMS storage forensic artifacts and the audit logs.

8.5 Approach Overview

Our goal in this chapter is to eliminate a major security vulnerability stemming from file tampering; our solution is envisioned as a component of a comprehensive auditing system that employs database forensics. We have previously built a tool that detects malicious activity when database logging was disabled [85] by comparing forensic artifacts and database logs. That approach relied on forensic artifacts left by SQL commands and assumed no OS level file tampering. DBStorageAuditor finds inconsistencies that were done by direct file modification. Future work, such as recovering a time line of events or user attribution, would involve expanding upon the current components to the system.

The remainder of the chapter describes our system to detect database file tampering, DBStorageAuditor, followed by an experimental evaluation in Section 8.10.
8.3 provides an overview of DBStorageAuditor, which consists of four components: forensic extraction(A), index integrity verification(B), carved index sorting(C), and tampering detection(D).

The forensic processing component is based on the forensic tool DBCarver [91] described in Section 8.2. DBCarver retrieves from storage all table records (including deleted records), record metadata, index value-pointer pairs, and several additional storage artifacts. We discuss new functionality that was added to DBCarver for this chapter in Section 8.6 (e.g., a page checksum extraction and comparison, a generalized approach to pointer deconstruction for several RDBMSes).

We first verify the integrity of indexes (discussed in Section 8.7) because indexes are used later to detect tampering of table data, so it is critical to verify index structure integrity. To achieve that, we evaluate the B-Tree in storage, consider corrupt data that matches B-Tree organization, and check for traces of an index rebuild (e.g., REORG, VACUUM – depending on a DBMS).

We cannot assume that index artifacts can be fully stored in RAM while matching index values to table records. Therefore, the carved index sorting component discussed in Section 8.8 pre-processes index artifacts to make DBStorageAuditor approach scalable. We approximately sort the index values based on their pointers which correspond to the physical location of records in a file and improves the runtime the matching process.

Finally the tampering detection component discussed in Section 8.9 detects cases of extraneous and erased data in storage. If a record and its artifacts can not be reconciled with index value-point pairs, such entries are flagged and returned to the user as suspected file tampering.

8.6 Forensic Analysis

Our proposed analysis relies on an expanded version of DBCarver (Chapter 3) to extract database storage artifacts that can not be queried using the DBMS. These artifacts include record metadata, deleted records, and index value-pointer pairs. In this section, we discuss the addition of a checksum comparison and generalized pointer deconstruction to DBCarver.

8.6.1 Checksum Comparison

In Section 5.4, we defined the checksum stored in the page header. Whenever data or metadata in a page is updated, either legitimately or through data tampering, the checksum must be updated accordingly. If the checksum is incorrect, the DBMS will recognize the
page as corrupt. This will result in warnings as well as data loss ranging from page to the
table or the entire database instance. Therefore, we can assert that if a checksum did not
change between time $T_1$ (previous inspection) and $T_2$ (current inspection), then the page
has not been modified and the records have not been exposed to tampering.

We implemented a dictionary of checksums taken from the DBMS pages that are to be
evaluated by DBCarver (it is possible to inspect any subset of the DBMS for tampering
signs – focusing only on data-sensitive tables). Our dictionary stores the checksum values,
where the object identifier and page identifier (Section 5.4) were the key and the checksum
was the value. The checksum dictionary should be stored off-site so it is not at risk of
tampering.

If the checksum has changed for a given page, the entire page must be inspected and
validated by DBCarver. If the checksum did not change for a page, only page metadata
was necessary to reconstruct. The metadata is needed to avoid false-positives in Alg.

8.6.2 Index Carving and Pointer Deconstruction

DBStorageAuditor uses index value-pointer pairs to identify inconsistencies in DBMS
storage. Therefore, the value-pointer pairs must be inspected. DBMSes do not allow in-
dexes to be queried directly (i.e., indexes can not appear in the FROM clause) which is why
we use DBCarver to retrieve index contents. However, the pointer parsing by DBCarver
was limited and specific to each DBMS; we developed a generalized approach to pointer
deconstruction allowing DBStorageAuditor to be compatible with any investigated
RDBMS.

We performed an analysis of pointers for 7 commonly used RDBMSes. Table 8.1 lists
these RDBMSes and summarizes our conclusions. We found that all of these DBMSes,
except for MySQL, stored a PageID and a Slot#. By default, MySQL creates an indexed
organized table (IOT) so the pointer deconstruction process is slightly different. We address
index pointers for IOTs later in this section. The PageID refers to page identifier that is
stored in table page header (Section 5.4). The Slot# refers to a records position within a
page. SQLServer and Oracle both store a FileID, which refers to file in which the page
is located. The DBMSes that do not include a FileID in the pointer, use a file-per-object
storage architecture (i.e., each table and index are stored in different files). The FileID for
these pointers is the ObjectID or it can be mapped back to the ObjectID if the object name
is the file name. Thus, an index pointer can be deconstructed into a FileID, PageID, and
Figure 8.4: An example of mapping index values to a record.

Slot# to map a value back to a table record location. Index pointers are typically the same as the internal DBMS row identifier pseudo-column.

<table>
<thead>
<tr>
<th>DBMS Version</th>
<th>FileID</th>
<th>PageID</th>
<th>Slot#</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLServer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Oracle</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ApacheDerby</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Firebird</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DB2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MySQL</td>
<td>No</td>
<td>Yes*</td>
<td>No</td>
</tr>
</tbody>
</table>
*The pointer references the second level of an IOT.

Table 8.1: Pointer Deconstruction.

Figure 8.4 demonstrates how index values are mapped back to the table records through our generalized pointer deconstruction. For each index value(A), the pointer stores a PageID(B) and Slot#(C). The pointer PageID(B) corresponds to the page identifier(D) in the table page header. The pointer Slot#(C) corresponds to the row directory address(E) in the table page. For example, the pointer for ‘Austin’ stores PageID = 8 and Slot# = 12. To find the record, the table page with identifier = 8 is found and the 12th row directory address is used to locate the record (68, ‘Alice’, ‘Austin’) within the page.

Index Organized Tables While MySQL was the only evaluated DBMS that created IOTs by default, IOTs are commonly used in other DBMSes under different names (e.g., IOT in Oracle, Included Columns in SQL Server) so we incorporated their pointer deconstruction. The pointer for a secondary index built on an IOT is made of a PageID that references a
page one level above the IOT B-Tree leaf page, and the primary key value. The PageID for the IOT leaf page can then be retrieved from the pointer stored in the second level of the B-Tree. After performing this additional IOT B-Tree access, we can associate every secondary index value with a PageID and a primary key value, where the PageID references an IOT leaf page and the primary key value replaces the Slot#. Figure 8.5 illustrates how a secondary index value can be mapped back to an IOT record. We have the same index on City and the same records from Figure 8.4. However, the records are now stored in an IOT, and we now have a B-Tree page one level above the IOT leaf pages. The City index values(A) now store the PageID for IOT B-Tree page(B) and the primary key values(C) as the pointer. The IOT B-Tree page stores primary key values(F) and leaf PageIDs(G) as the pointer. For example, the pointer for ‘Austin’ stores PageID 20 and the primary key 68. This directs us to the IOT B-Tree page with PageID 20 and the value-pointer pair (57, 8). The IOT B-Tree pointer tells us ‘Austin’ is in the leaf page with PageID 8 and the primary key value 68.

Figure 8.5: Mapping index values to an IOT record example.

8.7 Verifying Index Integrity

It is plausible for an adversary to tamper with the relevant index values in an attempt to conceal evidence of file tampering. In this section, we address several types of index tampering, and how to detect such activity.
8.7.1 B-Tree Integrity Verification

If the attacker changes a value, adds a record, or wipes a record from a table, he may also perform a complimentary operation in the index. For example, ‘Dan’ was changed to ‘Jane’ in a table record could also be similarly modified in the index leaf node.

Interestingly, this type of activity creates inconsistencies in the index B-Tree that do not arise in the table. We consider the case where an index value is changed in-place and the case where index value was erased (and possibly reinserted into the correct position in the B-Tree). If the index value was changed in-place, it would appear out-of-order in the leaf of the B-Tree. If the index value was erased, it creates an uncharacteristic blank space between values within the leaf page, which never occurs naturally.

8.7.2 The Neighboring Value Problem

An index value may sometimes be altered without violating the correct ordering of the B-Tree. For example, in Figure 8.6 ‘Dan’ is changed to ‘Dog’ preserving a correct value ordering of the Name index. This example shows how a table and an index can be altered without producing an inconsistency.

We build a function-based index that stores the hash value of column(s) to thwart tampering that involves neighboring range values. The values in hash-index will have a different ordering than the values in the secondary index so a neighboring value can occur in one, but not both. Figure 8.6 shows an example of how a hash index can be used to detect index tampering the involves neighboring values. In both the table and the Name index, the value was changed to ‘Dog.’ Changing the value in the Name index preserved the correct ordering. However, changing the value in the hash-index would result in an incorrect ordering since the values are organized differently. Function-based indexes are supported by many major DBMSes (e.g., IBM DB2, Oracle, and PostgreSQL); a computed column can be used for DBMSes that do not support function-based indexes (e.g., MySQL and Microsoft SQL Server).

<table>
<thead>
<tr>
<th>Name Index</th>
<th>Table</th>
<th>Hash Value Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>Jane</td>
<td>P2</td>
</tr>
<tr>
<td>Bob</td>
<td>Detroit</td>
<td>P4</td>
</tr>
<tr>
<td>Carl</td>
<td>Alex</td>
<td>P6</td>
</tr>
<tr>
<td>Dog</td>
<td>Lincoln</td>
<td>P8</td>
</tr>
<tr>
<td>Jane</td>
<td>Chicago</td>
<td>P1</td>
</tr>
<tr>
<td>Joe</td>
<td>Austin</td>
<td>P7</td>
</tr>
<tr>
<td>Kate</td>
<td>Joe</td>
<td>P3</td>
</tr>
<tr>
<td>Pat</td>
<td>Dog</td>
<td>P5</td>
</tr>
<tr>
<td>Sam</td>
<td>Sam</td>
<td>P9</td>
</tr>
</tbody>
</table>

Figure 8.6: Preventing the neighboring value problem.
8.7.3 SQL Index Rebuild

Although we assume that the attacker does not have privileges to rebuild an index through SQL, the index may nevertheless be rebuilt as part of routine maintenance. If an index is rebuilt post tampering, the reconstruction of the index will eliminate any inconsistencies (extraneous or erased data) between the table and the index because indexes will be built anew using the current table state. However, when an object is rebuilt, a new object is created and artifacts (discarded pages) from the old object are left behind in storage. Many of the pages from the old index are likely to be overwritten, but some pages are going to persist in storage following the rebuild [87].

Pages left behind from an index rebuild can serve as separate evidence to detect tampering. The old index version (or the parts recovered) can be treated as a separate index ($I_{t-1}$) from the newly rebuilt version ($I_t$). While the old index version does not contain a complete set of values due to having been partially overwritten, it can still be used to detect tampering. This would be applicable when auditing is not performed at regular intervals relative to the frequency of index rebuilds.

8.7.4 Manual Index Rebuild

In order to deceive DBStorageAuditor, an attacker would have to completely rewrite the entire index (or at least several different pages in it). While such operation is possible, performing it successfully poses several major challenges. We emphasize that typical security solutions are designed to greatly increase the level of difficulty to perform an attack, rather than create an absolute defense.

Section 8.4 discussed cached dirty page problem when physically modifying a page. Moreover, dirty index pages can introduce additional complications. First, a given index page is more likely to have a dirty version cached compared to a table page. An index page is not only modified when the indexed column is updated, but the index pointer must also be updated if an update causes a record to be written to a new location. Furthermore, index pages store significantly more values than table pages, increasing their chance to be modified. Second, as the index changes, the database may reorganize the B-Tree structure (e.g., page split). As parts of the index are rebuilt, pages are likely to be written to new locations in a file. We note that the physical order of a B-Tree does not reflect the logical order of the B-Tree. Third, the attacker may have to discover the physical location of other connected index pages (i.e., just finding the page with needed value is insufficient, several parts of the B-Tree would need to be reconstructed). Index leaf pages point to the next logical page in the B-Tree and sometimes to the previous page as well. This means that if a
logically adjacent page is rebuilt and written to a new location, then a modified index page would need to reflect that change. Therefore, the attacker would need to be aware of all internal B-Tree structure changes to guarantee a successful manual index rebuild. Finally, if a function-based index storing a hash value exists, we assume that an SA would not have knowledge of this function. Therefore, inconsistencies would still arise in any attempts to manually rewrite the index.

### 8.8 Index Sorting

When tables and indexes are carved, the data is extracted based on the physical location within the files. Therefore, the relationship between the ordering of the carved table records compared to the index values is random, with a possible exception of a clustered index (it is common for a clustered index to be manually updated, such as PostgreSQL with VACUUM command). Assuming that the index can not be fully loaded into into RAM, expensive random seeks must be performed to map index values to table records. In this section we propose a method to reorder the index to make the process of matching index values and table records scalable.

As demonstrated in Section 8.6, index pointers correspond to the physical position of the table records. Therefore, sorting the index values by the pointers produces the same ordering for index values and table records. Carved table records and index values are then read sequentially, similar to a merge join process.

For an index that is too large to fit into memory, sorting the index pointers can be a costly operation. If we assume that $N$ table pages will be read into memory when detecting table tampering (Section 8.9), then index values need to be sorted across every $N$ pages, but values do not need to be sorted within $N$ pages. We call each set of index values that belong in $N$ table pages a **bucket**. We perform approximate sorting by re-ordering index values across buckets but not within buckets.

For each index bucket, we record the minimum and maximum table page identifier. If an index value is in the range of page identifiers for a bucket, the page identifier, slot number, and index value are stored in that bucket. When table pages are read for table tampering detection, the relevant bucket(s) are read into memory using the table page identifier and the index bucket minimum and maximum values.

Figure 8.7 shows an example of an index that is approximately sorted on the pointer. For each value in the index, there is a pointer that contains a PageID and a Slot#. We first create a set of buckets where each bucket contains 1000 PageIDs. We read the carved index data, and assign a value to the appropriate bucket using the pointer. For example, the
first and second index values ‘Alex’ and ‘Bob’ belong in bucket #2 because their PageIDs, 2000 and 1002 are between the minimum and maximum PageID range for the bucket. We then store the PageID, Slot#, and Value in the bucket. ‘Carl’ has a PageID 5 so that value belongs in bucket #1. Bucket #2 demonstrates that PageIDs do not need to be sorted within the bucket. Furthermore, we see that PageID 2000 in bucket #2 has two values. This can occur as a result of legitimate SQL operations that create stale index values.

### Approximate Sorted Index

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>PageID Min/Max</th>
<th>PageID</th>
<th>Slot#</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 - 1000</td>
<td>5</td>
<td>33</td>
<td>Carl</td>
</tr>
<tr>
<td>2</td>
<td>1001 - 2000</td>
<td>2000</td>
<td>1</td>
<td>Alex</td>
</tr>
</tbody>
</table>

### Carved Index

<table>
<thead>
<tr>
<th>Value</th>
<th>PageID</th>
<th>Slot#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>2000</td>
<td>1</td>
</tr>
<tr>
<td>Bob</td>
<td>1002</td>
<td>2</td>
</tr>
<tr>
<td>Carl</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>Dan</td>
<td>4400</td>
<td>12</td>
</tr>
<tr>
<td>Jane</td>
<td>3050</td>
<td>20</td>
</tr>
<tr>
<td>Joe</td>
<td>1001</td>
<td>1</td>
</tr>
<tr>
<td>Kate</td>
<td>1002</td>
<td>1</td>
</tr>
<tr>
<td>Pat</td>
<td>1001</td>
<td>2</td>
</tr>
<tr>
<td>Sam</td>
<td>2000</td>
<td>1</td>
</tr>
</tbody>
</table>

### Index Organized Tables

Approximately sorting secondary indexes for index organized tables (IOT) is a slightly different process. When an IOT is used, the secondary index pointer is made up of a PageID that references a second level B-Tree page and the primary key value instead of a PageID that references the table and a Slot#. To sort the secondary index values, the second level BTree pages from the primary key is used to retrieve the table PageIDs for each value. Furthermore, the primary key value is now used in place of the Slot#.

The cost of approximate sorting is dependent on the amount of available memory. A bucket must fit into memory. Fewer buckets results in quicker bucket assignment for values, but buckets will be larger requiring more memory. In Section 8.10.2 we provide costs of approximately sorting an index.

### 8.9 Detecting Table Tampering

In Section 8.4 we discussed how database files, specifically tables, are vulnerable to tampering. We propose using the validated indexes (Section 8.7) to verify the integrity of table
records in storage. Earlier in this chapter, we classified data tampering that involves changing a value or adding records as extraneous data, and we classified data tampering that involves wiping records as erased data. In this section we present and discuss algorithms to detect both extraneous and erased data.

8.9.1 Extraneous Data Detection

Extraneous data is a record or a value that has been added to a table through file tampering. Since extraneous data is not added using the DBMS, it is not reflected in the indexes. Therefore, if a record does not have any corresponding index pointer, then the entire record is suspected of having been added through file tampering. Any table with a primary key can be tested because an index is automatically created for a primary key constraint. Similarly, if a table value does not match an index value with the corresponding pointer, then the value is assumed to have been modified through file tampering. This validation test does require that an index exist on the column(s). We use the carved data from Section 8.6 and an approximately sorted index (Section 8.8) that was not been subject to tampering (Section 8.7).

Algorithm 6 describes how to detect extraneous data. First, we read $N$ table pages at a time for evaluation; we then scan the approximately sorted index buckets for the relevant table page identifiers and read the index pages from the relevant bucket(s). For every record in the $N$ table pages, we find the corresponding index pointer. If an index pointer does not exist, this record is added to a list of likely extraneous data. If an index pointer does exist for a record, the indexed column is compared to the index value(s) for that pointer (there may be more than one index value per pointer for legitimate reasons). If the table value is not in the set of index values, then this value is added to a list of likely extraneous data. This is evidence of a value that has been changed. After all table pages have been read and all records evaluated, the resulting extraneous data list is returned to the user.

8.9.2 Erased Data Detection

Erased data is data explicitly wiped from table storage through file tampering. Deleted records are likely to be overwritten by new records over time as the DBMS runs. However, a deleted record will never be overwritten by something that is not another record of the same structure. Therefore, if an index value points to an area in storage that does not contain a proper record (including metadata), then record wiping is suspected. We are not concerned with matching the specific index value since this is done in Algorithm 6 but rather that a pointer must reference an area in storage that resembles a record.
Algorithm 6 Extraneous Data Detection

1: Table ← carved table data: PageIDs, Slot #s, and Records.
2: N ← the number of table pages to be read.
3: SortedIndex ← the approximately sorted index (Section 8.8).
4: Flag ← an empty list to store extraneous data.
5: for each NPages ∈ Table do
6:    MinPID ← the minimum page ID from NPages.
7:    MaxPID ← the maximum page ID from NPages.
8:    Indexes ← an empty list to store index pages.
9:    for each Bucket ∈ SortedIndex do
10:       if (MinPID ∈ Bucket) ∨ (MaxPID ∈ Bucket) ∨ (MinPID < Bucket ∧ MaxPID > Bucket) then
11:          Indexes.append(Bucket)
12:       for each Rec ∈ NPages do
13:          RecPtr ← Rec.PageID.Slot#
14:          if RecPtr ∈ Indexes.PageID.Slot#.Vals then
15:             if Rec.Val /∈ Indexes.PageID.Slot#.Vals then
16:                Flag.append(['ModVal', RecPtr, Rec, Val])
17:          else
18:             Flag.append(['HiddenRecord', RecPtr, Rec])
19:    return Flag

Algorithm 7 describes how to detect erased data. First, we read each bucket from the approximately sorted index. When a bucket is read, the table pages with the relevant page identifiers are also read. We iterate through each index value in the bucket. If the pointer for an index value does not match any record in the table pages, then the index value is appended to a list of erased data. After all index buckets have been evaluated, the list of erased data is returned to the user.

Adjacent Deleted Records It is possible that multiple deleted records can exist adjacent to one another in a page. When this happens it is also possible the a single record could overwrite all of one record and part of another. For example, (1, ‘Ed’) and (2, ‘Tom’) are deleted records that are next to each other in storage. The inserted record (3, ‘Karen’) could overwrite all of (1, ‘Ed’) and part of (2, ‘Tom’). This presents a problem because any old index value for (2, ‘Tom’) would now point to the middle of the inserted record, rather than to a full record. In this scenario, Algorithm 7 would return a false-positive for the index value from (2, ‘Tom’). These false-positives can be eliminated by comparing these results to audit log entries. For example, if a delete command in the log could explain (2, ‘Tom’), then this could be declared as not malicious. This functionality is not currently
Algorithm 7 Erased Data Detection

1: $Table \leftarrow$ carved table data: PageIDs, Slot #s, and Records.
2: $SortedIndex \leftarrow$ the approximately sorted index (Section 8.8).
3: $Flag \leftarrow$ an empty list to store erased data.
4: for each $Bucket \in SortedIndex$ do
5:     $NPages \leftarrow$ pages from $Table$ where $PageID \in Bucket$
6:     for each $IndexValue \in Bucket$ do
7:         $Ptr \leftarrow IndexValue.PageID.Slot#$
8:     if $Ptr \notin NPages$ then
9:         $Value \leftarrow$ the index value
10:        $Flag.append([\textquoteleft ErasedRecord\textquoteright, Ptr, Value])$
11: return $Flag$

supported by DBStorageAuditor, and it would be explored in future work to achieve a more complete auditing system.

8.10 Experiments

In this section, we present a set of experiments that evaluate the performance, accuracy, and limitations of DBStorageAuditor. Table 8.2 summarizes the experiments in this section.

<table>
<thead>
<tr>
<th>#1</th>
<th>Forensic analysis (Sec 8.6) cost evaluation. DB files were carved at a rate of 1.2 MB/s. A checksum comparison can improve carving costs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2</td>
<td>Approximate sorting (Sec 8.8) cost evaluation. Fewer buckets improves runtime, but requires more memory.</td>
</tr>
<tr>
<td>#3</td>
<td>Algms 6 and 2 (Sec 8.9) cost evaluation. Both algorithms increase linearly with table size.</td>
</tr>
<tr>
<td>#4</td>
<td>DBStorageAuditor detection evaluation. Algms 6 detects an added record, Algms 6 detects a modified value only for an indexed column, and Algms 2 reconstructs erased data that was indexed.</td>
</tr>
<tr>
<td>#5</td>
<td>DBStorageAuditor detection limitations after an index rebuild (Sec 8.7). DBStorageAuditor can use the old version of an index depending on the DBMS.</td>
</tr>
</tbody>
</table>

Table 8.2: Summary of experiments.

MySQL 5.7, PostgreSQL 9.6, and Oracle 11g R2 DBMSes were used in these experiments. We believe these three RDBMSes are a good representative selection from the commonly used RDBMSes. Not only are they widely used commercial and open-source DBMSes, but they also represent the spectrum of different storage decisions across about ten DBMSes we have studied. For example, PostgreSQL does not support IOTs, Oracle
offers an option to create IOTs, and MySQL automatically uses IOTs. The default page sizes for each DBMS were used: 8K for Oracle and PostgreSQL and 16K for MySQL. Data from the Star Schema Benchmark (SSBM) \cite{59} was used to populate our DBMS instances. Table 8.3 can be used to reference table sizes used throughout this section. DBMS instances ran on servers with an Intel X3470 2.93 GHz processor and 8GB of RAM running Windows Server 2008 R2 Enterprise SP1 or CentOS 6.5.

<table>
<thead>
<tr>
<th>Table</th>
<th>Scale</th>
<th>DB File Size(MB)</th>
<th>Values(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lineorder</td>
<td>1</td>
<td>600</td>
<td>6</td>
</tr>
<tr>
<td>Lineorder</td>
<td>4</td>
<td>2400</td>
<td>24</td>
</tr>
<tr>
<td>Lineorder</td>
<td>14</td>
<td>8300</td>
<td>84</td>
</tr>
<tr>
<td>Supplier</td>
<td>1</td>
<td>&lt;1</td>
<td>2K</td>
</tr>
</tbody>
</table>

Table 8.3: SSBM table sizes used through the experiments.

The different DBMS storage-altering operations that we are seeking to detect are discussed in Section 8.9. When modifying files, we re-calculated and updated the page checksum value for the PostgreSQL pages; in MySQL and Oracle we disabled the page checksum validation. Before modifying files, we first shutdown the DBMS instance.

### 8.10.1 Forensic Processing

The objective of this experiment is to evaluate the computational cost associated with the forensic processing component of DBStorageAuditor discussed in Section 8.6. In Part-A, we provide DBCarver runtimes against database files of various sizes from MySQL, Oracle, and PostgreSQL DBMSes. In Part-B, we repeat the same evaluation, further including a checksum re-computation.

**Part-A**  We created a series of database files for each DBMS to pass to DBCarver. We created three LINEORDER tables: Scale 1, 4, and 14. Each table was stored in a separate file. The PostgreSQL files were carved at an average rate of 1.0 MB/s, the MySQL files were carved at a rate of 1.2 MB/s, and the Oracle files were carved at a rate of 1.5 MB/s.

**Part-B**  We used the PostgreSQL LINEORDER Scale 4 table from Part-A to evaluate the checksum comparison we added to DBCarver. We modified pages that induced a checksum change for 1%, 5%, 10%, and 100% of the pages in the database file. The carving rate for each percent modification was 100% → 1MB/s, 10% → 9 MB/s, 5% → 18 MB/s, and 1% → 58 MB/s. The cost of forensic pre-processing is thus proportional to the number of modified pages rather than the total size of the DBMS storage.
8.10.2 Index Sorting

The objective of this experiment is to evaluate the costs associated with approximately sorting the index values on the pointers. The output produced by the forensic analysis is similar for all DBMSes so this component of DBStorageAuditor is not tested for DBMS-specific features. In Part-A, we vary the size of bucket; in Part-B, we vary the size of the indexes.

Part-A  To evaluate approximate sorting with respect to bucket size, we used the carved output from PostgreSQL database files containing a LINEORDER Scale 4 table, a secondary index on LO_Revenue, and a secondary index on LO_Orderdate. Table 8.4 summarizes the performance results. As the number buckets decreases the time to sort the data decreases. However, a bucket must fit into memory, so increasing of bucket sizes is limited by available RAM.

<table>
<thead>
<tr>
<th>Bucket Size (Pages)</th>
<th>Bucket Count</th>
<th>Orderdate (sec)</th>
<th>Revenue (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>63</td>
<td>1366</td>
<td>1380</td>
</tr>
<tr>
<td>10,000</td>
<td>32</td>
<td>1121</td>
<td>1131</td>
</tr>
<tr>
<td>50,000</td>
<td>7</td>
<td>932</td>
<td>945</td>
</tr>
<tr>
<td>100,000</td>
<td>4</td>
<td>909</td>
<td>926</td>
</tr>
<tr>
<td>200,000</td>
<td>2</td>
<td>903</td>
<td>918</td>
</tr>
</tbody>
</table>

Table 8.4: Index sorting costs with varying bucket sizes.

Part-B  To evaluate approximate sorting with respect to the size of an index, we used the carved output from PostgreSQL database files containing LINEORDER Scale 1, 4, and 14 tables and a secondary index on LO_Revenue for each table. Table 8.5 summarizes the results. If the bucket size is increased proportionally for the table size, the approximate sorting cost increases linearly.

<table>
<thead>
<tr>
<th>Bucket Size (Pages)</th>
<th>Index sorting time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scale 1</td>
</tr>
<tr>
<td>10,000</td>
<td>239</td>
</tr>
<tr>
<td>50,000</td>
<td>231</td>
</tr>
<tr>
<td>100,000</td>
<td>n/a*</td>
</tr>
<tr>
<td>200,000</td>
<td>n/a*</td>
</tr>
</tbody>
</table>

*Bucket size is larger than the table.

Table 8.5: Approximate sorting costs for varying table sizes.
8.10.3 Tampering Detection Costs

The objective of this experiment is to evaluate the costs associated with Algorithms 6 and 2. For this experiment we used the LINEORDER Scale 4 table. We used one index on the LO_Revenue and multiple indexes on the LO_Revenue and LO_Orderdate. We approximately sorted the index using buckets with 50K pages.

**Part-A: Algorithm 6**
To evaluate the costs associated with Algorithm 6, we used the output from two different secondary indexes (LO_Revenue and LO_Orderdate) on LINEORDER Scale 4 and one secondary index (LO_Revenue) on LINEORDER Scale 14. Table 8.6 summarizes the runtime results. The runtime for Algorithm 6 was the same for LO_Revenue and LO_Orderdate on LINEORDER Scale 4, and the cost increased linearly for LO_Revenue on LINEORDER Scale 14.

<table>
<thead>
<tr>
<th>Table</th>
<th>Index</th>
<th>Part-A (sec)</th>
<th>Part-B (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale 4</td>
<td>LO_Revenue</td>
<td>966</td>
<td>503</td>
</tr>
<tr>
<td>Scale 4</td>
<td>LO_Orderdate</td>
<td>961</td>
<td>476</td>
</tr>
<tr>
<td>Scale 14</td>
<td>LO_Revenue</td>
<td>3482</td>
<td>1773</td>
</tr>
</tbody>
</table>

Table 8.6: Algorithm 6 and 2 runtimes.

**Part-B: Algorithm 2**
We used the same tables in indexes from Part-A of this experiment to evaluate the costs associated with Algorithm 2. Table 8.6 summarizes the runtime results. Similar to Algorithm 6, the cost for Algorithm 2 was nearly the same for LO_Revenue and LO_Orderdate on LINEORDER Scale 4, and the cost increased linearly for LO_Revenue on LINEORDER Scale 14.

8.10.4 Detection Capabilities

The objective of this experiment is to demonstrate the file tampering activity that DBStorageAuditor is capable of detecting. For each part in this experiment, we simulate one defined type of malicious activity and explain how it was detected. We manually add records to the database file (Part-A), change values in the database file (Part-B), and erase records from the database file (Part-C). We present results only for PostgreSQL because our results for Oracle and MySQL were very similar.
**Setup**  We created a LINEORDER Scale 4 table for a PostgreSQL DBMS. An index existed on the primary key (LO.Orderkey, LO.Linenumber) and we created a secondary index for LO.Revenue and LO.Orderdate.

We also created a function-based index on LO.Revenue that used the 32-bit version of the MurmurHash2 hash function.

**Part-A**  We manually added 5 records (shown in Figure 8.8) to the file containing the LINEORDER table. We added a record to five different pages (with PageIDs 11, 12, 13, 14, and 15). Existing primary key values were included in each of the five records. For each of these records, all of the data was the same as the existing records with the same primary key except we used LO.Suppkey -5 and LO.Revenue -100000.

<table>
<thead>
<tr>
<th>Primary Key</th>
<th>LO_Suppkey = -5</th>
<th>LO_Revenue = -100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>108733</td>
<td>7417</td>
</tr>
<tr>
<td>4001</td>
<td>38143</td>
<td>210370</td>
</tr>
<tr>
<td>12001</td>
<td>2303</td>
<td>391486</td>
</tr>
<tr>
<td>100001</td>
<td>102599</td>
<td>383999</td>
</tr>
<tr>
<td>200001</td>
<td>85157</td>
<td>130108</td>
</tr>
</tbody>
</table>

Figure 8.8: Records added to the LINEORDER file.

The addition of these five records produced several interesting outcomes. First, these records bypassed the primary key constraint since they contained primary key values that previously existed in the table. The DBMS only checks constraints when executing API-based load commands, and it does not retroactively check the table for constraint violations. Adding the record to the file bypasses all official channels and is thus never checked for constraint violations. Second, these records also bypassed referential integrity since the LINEORDER table references the SUPPLIER table, and LO.Suppkey -5 did not exist in the SUPPLIER. Similar to the primary key violation, the constraint violation was never caught by the DBMS. Finally, table access for the same query could produce different results because the indexes were not updated after we added these five records. For example, the two versions of the following query returns different results:
• Query 1 → 34600180980
  
  ```sql
  SELECT SUM(LO_Revenue) FROM Lineorder
  WHERE LO_Orderdate = 19960319;
  ```

• Query 2 → 34600180980 - 100000
  
  ```sql
  set enable_seqscan=true;
  SELECT SUM(LO_Revenue) FROM Lineorder
  WHERE LO_Orderdate = 19960319;
  ```

Query 1 uses the `LO_Orderdate` index to access the table while Query 2 uses a full table scan. Record #1 from Figure 8.8 was included in Query 2, but it was not included in Query 1.

Algorithm 6 successfully detected the fact that five new records do not have corresponding pointers in the primary key index, the two secondary indexes, and in the function-based index. Problem was flagged by a `False` value for the line 14 `If` condition resulting in the malicious records being added to the list of invalid data at line 18. Each existing index serves as an additional validation to detect table tampering – and the function-based makes sure that small incremental changes are not possible.

**Part-B** Next, we changed `LO_Revenue` for all 41 records where the `LO_Custkey` 4321 and `LO_Orderdate` between 19930101 and 19931231. To simulate a neighboring value problem (a small change that does not violate index ordering), we changed the record with `LO_Custkey` 4321 and `LO_Revenue` 3271986 to 3271987 in both the table and the `LO_Revenue` index. For all other records we subtracted 100000 from `LO_Revenue` in the table.

Algorithm 6 reported that 40 records had an inconsistent value based on the `LO_Revenue` index and 41 records had an inconsistent value based on the function-based index on `LO_Revenue`. The difference of the one additional record was due to the neighboring value attack which regular index may fail to detect. These values were detected by a `False` value for the line 15 `If` condition resulting in the malicious values being added to the list of invalid data at line 16. We can conclude that the primary key and `LO_Orderdate` columns were not tampered with for these and all records since they were not included in the invalid data. However, we can not make any conclusion if any other of the non-indexed columns for these or any records were tampered.
Part-C Next, we erased all 3085 records with the LO_Suppkey 123 from the file. For data erasure, we explicitly overwrote the records and their metadata with the NULL ASCII character.

Algorithm returned the primary key index, the function-based index, and two secondary indexes values. Each had 3085 values that did not point to a valid record structure. These were detected by a True value for the line 8 condition in Algorithm resulting in malicious data being added to the list of invalid data at line 10. By combining the values for each pointer we reconstructed partial records containing the index columns to explain the missing data. However, the data for the non-indexed columns was unable to be reconstructed since it was not indexed.

8.10.5 Long-Term Detection

The objective of this experiment is to evaluate the artifacts produced by an index rebuild that can used by DBStorageAuditor. We evaluate a different DBMS for each part of this experiment: Oracle in Part-A, MySQL in Part-B, and PostgreSQL in Part-C.

We performed the following steps for each DBMS. After each step, we copied the database file for analysis. Table 8.7 summarizes the results.

- **T₀**: Started with the Supplier Scale1 (2K records) table and a secondary index on S_Name.
- **T₁**: Erased/wiped all 829 records where S_Region equaled ‘ASIA’ or ‘EUROPE’.
- **T₂**: Rebuilt the index. Each DBMS used a different index rebuild command:
  - **Oracle**: ALTER INDEX Supp_Name REBUILD ONLINE
  - **MySQL**: DROP and CREATE commands
  - **PSQL**: REINDEX TABLE Supplier

<table>
<thead>
<tr>
<th>DBMS</th>
<th>T₀(pgs)</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>1 root, 9 leaf</td>
<td>no change</td>
<td>All index pages from the old index</td>
</tr>
<tr>
<td>MySQL</td>
<td>1 root, 5 leaf</td>
<td>no change</td>
<td>remained in DB storage.</td>
</tr>
<tr>
<td>PSQL</td>
<td>1 root, 10 leaf</td>
<td>no change</td>
<td>2 leaf pages from the old index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>remained in DB storage.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>None of the old index remained in DB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>storage.</td>
</tr>
</tbody>
</table>

Table 8.7: Index rebuild summary.
Part-A: Oracle  The index contained 1 root page and 9 leaf pages after creation at $T_0$. No changes were made to the index after the table records were erased at $T_1$. After the index rebuild at $T_2$, the new index contained 1 root page and 5 leaf pages. All of the pages from the original version at $T_0$ remained in the database file. The DBMS assigned a new ObjectID to the new version of the index so index pages between versions were easily distinguished. Since the entire version of index was found, it could be used by DBStorageAuditor. The old version of the index still contained pointers to the erased records, whereas the new version only contained pointers to active records in the table.

Part-B: MySQL  The index contained 1 root page and 5 leaf pages after creation at $T_0$. No changes were made to the index after the table records were erased at $T_1$. After the index rebuild at $T_2$, the new index contained 1 root page and 3 leaf pages. 2 out of the 5 leaf pages from the original index remained in database storage. This demonstrates that the DBMS immediately reclaimed the pages from the dropped index. Since the new index version used less storage space, 2 pages from the old version remained in the file. In this scenario, a B-Tree could not be fully validated with only 2 leaf pages making them less useful as evidence for DBStorageAuditor. It is likely that copies of the index could be carved from a disk image due to activity such as writes that do not occur in place and paging files. DBStorageAuditor does not currently reconstruct entire B-Tree indexes from disk images. Future work will seek to reconstruct objects from disk images, which requires multiple versions of pages to be considered.

Part-C: PostgreSQL  The index contained 1 root page and 10 leaf pages after creation at $T_0$. No changes were made to the index after the table records were erased at $T_1$. After the index rebuild at $T_2$, the new index contained 1 root page and 6 leaf pages. The new version of the index was assigned a new ObjectID and a separate file. All pages belonging to the old version of the index were disassociated with its file, and this storage was reclaimed by the file system. In this scenario, DBStorageAuditor can no longer detect that the records were erased. As discussed in Part-B, it is likely that the index could be carved from a disk image. This will be explored in future work since a logical timeline would need to be recreated to account for multiple page versions.

8.11 Conclusion

Database file tampering can be used to perform malicious operations while bypassing database security mechanisms (logging and access control) and constraints. We presented
and evaluated DBStorageAuditor component that detects database file tampering. Our approach relies on a forensic inspection of database storage and identifies inconsistencies between tables and indexes.

Future work plans to expand upon this chapter and work from [35] to create a complete database auditing framework. This future work would include creating a timeline of events and user attribution of storage artifacts. Our auditing framework relies on inherent characteristics of database storage that users, including privileged users, are incapable of controlling.
Chapter 9

An Independent Analysis of Data Organization

9.1 Introduction

Indexing is a primary technique in Relational Database Management Systems (RDBMS) to logically order data. Therefore, it is a key factor for scalable query processing. When the underlying data is clustered in index order, query processing scales up. However, in the absence of clustering a regular secondary index merely improves search performance while potentially incurring random I/O for each page access. In the worst case for an indexed query predicate on an unclustered index, a cost-based optimizer may resort to full sequential table scan for query selectivity as low as 0.01 (1%).

It is well recognized that moving object or sensor data warehouses that continually ingest data, which requires clustered and unclustered indexes to support analytical workloads, face query response time degradation due to indices becoming severely unclustered, i.e., incurring random I/O on each disk access. It is not uncommon for a database warehouse to undertake a downtime to recluster the entire data and improve performance.

However, such slowdowns due to random disk I/O can be reduced if the database shares some information of the physical location of attributes on a disk. We illustrate this reduction through an example in Figure 9.1. Consider Table T in Figure 9.1 with attributes \{ID, Name\}. The table is physically clustered on attribute \{ID\} into seven pages, i.e., the pages are in sequential order on the disk. The table also records the physical location of each row which is marked with an internal \{RowID\} column. Clustering this column on \{ID\} will sort the attribute \{ID\} and physically cluster the sorted result. Note that in order to minimize maintenance costs, the clustering on \{ID\} in Figure 9.1 example is not strict but rather approximate. Consider a query that accesses values based on ID BETWEEN #1 and #6. The secondary index will look up the matching keys, reading a number of index
pages (intermediate levels) and two pages from leaf level of the index (incurring several seeks before accessing the table itself). First three pointers (Row1, Row3, Row2) will access the first page, which will be cached after the Row1 lookup. Fourth match (Row19) will require a seek and a read of a seventh page at the end. Finally, fifth and sixth match will correspond to pointers (Row4, Row5) causing yet another seek and reading of the second page in the table. A more efficient access path would recognize that five out of six matched values are in fact co-clustered in first two pages, with one outlier (#4) that resides in the overflow page and avoid seeking back and forth.

The only way to take advantage of this seek reduction is by determining the level of physical co-clustering within attribute \{ID\}, an information which is maximally available through the RowID column of the table. Thus for instance, if the database was indexed on RowID, with each range of RowID values consisting of six table rows, then such an index will quickly determine the physical co-clustering and lead to two seeks instead of three seeks. In general, the difference can be much larger. The sparse index on the right of table T illustrates that fewer seeks are possible with physical clustering.

Figure 9.1: Storage layout of PLI and native database indexes (secondary and index-organized).
To create an index based on physical location ordering of data rows, one must precisely determine the physical location, i.e., the values of the **RowID** column. While this internal column maps the queryable tuple to the physical location on the disk, commercial databases, due to physical-logical independence do not provide this information transparently. In fact, while the physical location of a row can be determined through a SQL query (e.g., `SELECT ROWID` in Oracle), given two physical locations, there is no SQL query that determines if the two physical locations are strictly ordered on disk. Ordering a table on the internal **RowID** attribute, and inserting the result into a new table will also not guarantee that the resulting table is strictly ordered. To obtain the precise physical locations of the table rows in this chapter we use a forensic tool that can read most commercial database storage files and output rows as they are physically ordered on disk for each table.

We describe a bucket based mechanism that exploits the approximate sorting inherent to Table T in Figure 9.1. Instead of traditional index that maps one value to one storage location (e.g., `{ID}=3` to Row2 pointer) **PLI** utilizes range-of-values to range-of-addresses mapping. In our example, there is no entry for `{ID}=3` specifically; instead, there is Bucket1 that represents a value range [1–10] (min and max for `{ID}`) which is mapped to a storage location range (in our example, Bucket1 corresponds to a physical pointer range of [Row1–Row6]). What Bucket1 tells us is that physical address range [Row1–Row6], which corresponds to first two pages, contains only values between 1 and 10. The index enables us to include (or exclude) this particular bucket without knowing the exact set of values or their specific ordering within the bucket.

Bucket mapped index has significant advantages over a typical secondary index. First, it provides the advantages inherent to a sparse index. That is, it requires one record per bucket (instead of one record per row) and is easier to maintain and use for lookup. Furthermore, bucket ranges defined in terms of **RowID** can be externally used in any database that exposes **RowID** values. This approach can be effectively generalized to multiple databases and attached to a live DBMS. Second, the value range associated with each bucket makes it easy to build expression-based variations of the same structure. For example, in Figure 9.1 we have a second **PLI** structure constructed on `{ID-1}` rather than `ID`. The only change that such structure requires is re-mapping the value ranges (e.g., Bucket1 [1–10] becomes Bucket1 [0–3]) and the new **PLI** can be used on matching expression. This is far easier for order-preserving functions and we plan to explore other mappings and effects of inter-column correlation.

Our experiments show that a live database can be augmented with **PLI** using existing **RowID** and achieving query performance competitive to that of a native clustered index (or even exceed native performance because clustered indexes are not implemented to act as
a true sparse index). Furthermore, PLI is also associated with surprisingly low overhead and higher tolerance to storage fragmentation (due to inserts) because of its sparse nature and approximate (rather than strict) ordering.

9.2 Related Work

Some DBMSes (e.g., Oracle and MySQL) implement an Index Organized Table (IOT) as a replacement for clustered table. Figure 9.1 includes an example of IOT compared to a clustered index or PLI. While a traditional clustered index is still an additional structure that happens to be aligned with the sorting order of the table (table and index are two distinct structures), IOT is a merged structure with rows of the table spliced into the leaf nodes of the index itself. IOTs do achieve clustering (textbook definition) in that the table data is now kept sorted as new rows are inserted. However, this solution comes at a price. The leaves of the BTree data structure are logically sorted forming a linked list (each leaf node points to the next sibling). However, such linked list is not guaranteed to maintain a physical ordering as a clustered table usually does. Furthermore, even if a physical ordering of index leaves exists initially, BTree maintenance algorithm cannot maintain such continuity as the tree splits and merges (nor is this the goal of BTree structure).

Kimura et al. proposed dividing a table into buckets as a scan unit with correlation maps (CMs) index [42]. Using buckets to scan a table allows for a compressed index structure, but can result in false positives. A compressed index structure can be cached, reducing I/O operations for index maintenance just like PLI. Similar to CMs, our method records the ranges of values stored for each bucket. Unlike CMs, our method only requires access to internal row identifier – while CMs require a built-in clustered index to operate. As we show in our experiments, built-in clustered index has some practical (database-specific) limitations.

Generalized partial indexing builds unclustered indexes around records defined by the user, leaving some records not indexed [78]. A physical location index is similar in that the user defines which sections, i.e. buckets, of the table to reorder possibly leaving some buckets unordered. However, PLI provide the benefit of indexing to most records, and approximately sorts data across buckets. In both methods, index maintenance cost is reduced by only recording access or reorganizing data that benefits queries.

Database cracking expands on generalized partial indexing by reorganizing a cracker index in pieces accessed by queries [39]. Our work allows the user to reorganize data across units of buckets, where the size of the bucket is determined by the user instead of a query. Similar to database cracking, data is only organized across, not within, a piece or bucket.
The major difference between the two is that database cracking requires significant rewrite of the DBMS engine, while we add PLI to a live database.

Cheng et al. implements predicate introduction to improve query performance [10]. Predicate introduction can be used to improve a query by accessing a column with an index, or by reducing the tuples scanned for a join. Instead of rewriting queries based on structures created by the user, our work rewrites queries with constraints on the database internal row identifier in the WHERE clause of the query. Since the row identifier is typically used to access rows when a full table scan is not used, there is no benefit to using user created structures.

9.3 How to Build a Custom Clustered Index

To create a database-independent clustered index in a DBMS, we augment the DBMS with a module that consists of a sparse index structure, a maintenance component and an automated SQL query re-writer. Section 9.4 presents results using a PostgreSQL and Oracle DBMSes.

9.3.1 Architecture

The architecture of PLI operation is shown in Figure 9.2. We rely on the native database table(A) with no modifications or assumptions about DBMS engine features (e.g., DBMS may not even support clustering). Initially, we use DBCarver to inspect table layout as it currently exists. As shown in [91], looking for specific pages in a table is orders of
magnitude faster compared to full reconstruction of disk image. If the table is sufficiently (approximately) organized in the desired fashion and can be represented as a sequence of bucket ranges (e.g., first 10 pages contain function values [0 – 10], next 10 pages contain function values [9 – 12], etc.), then PLI can be built immediately; otherwise, we need to reorganize the table (by creating a replacement table with custom ORDER BY clause). Note that any sorting function supported by DBMS can be chosen (e.g., income-expenses or $\sqrt{\text{income}}$). The PLI structure is then constructed as a substitute for the native DBMS index – Figure 9.1 outlines different index choices. PLI is orders of magnitude cheaper to maintain when compared to a regular index for two reasons: 1) PLI is sparse and thus very small, and 2) PLI does not need to track DELETE operations (we explain why in this section). Finally, in order to query with PLI, we use a simple query rewrite process that occurs outside of the database. An incoming query is augmented to include a predicate on database-specific implementation of the RowID to instruct the DBMS engine which pages to read. The rest of this section discusses creation and use of PLI in more detail.

9.3.2 Initial Setup

The first step in using PLI is to inspect the table and organize it (if necessary) according to the desired access pattern. Note that reorganization refers the table data itself not to creation of an additional index. Secondary index does not permit sequential access and introduces significant overhead in addition to the original table. Databases clustering functionality is severely restricted in practice (e.g., in many DBMSes clustering index key must be unique). Despite the fact that sparse access is the distinguishing feature of clustering indexes, they are never truly sparse (e.g., using only 1 index entry per 80 rows on a page) when used in practice.

If the table is not already sorted as we prefer, we impose the ordering by recreating that table structure. In either case we discard the existing secondary index (as PLI will replace it). Database user can choose arbitrary ordering that need not be unique or strict; any function or rule supported by ORDER BY clause would be acceptable. To order table T on function of columns (A-C), we create a new structure as `CREATE TABLE T_PLI AS SELECT * FROM T ORDER BY (A-C)`.

Once the sorted table is created, we use DBCarver to validate table’s storage sorting at the physical level. The table is likely to be sorted (or at least mostly-sorted) as the ORDER BY clause specified as non-clustered tables are generally stored in order of insertion. However, although such sorting is not guaranteed – in practice, new table may be
stored differently on disk (most notably in Oracle). Using the underlying sorting, we next generate a bucket mapping structure, recording RowID boundaries for each bucket.

9.3.3 PLI Structure and Maintenance

The structure of PLI is similar to that of a traditional sparse primary index. A regular sparse index will direct access to the correct page or sequence of pages instead of referencing particular rows. For example, in Figure 9.1, PLI consists of 3 buckets of approximately sorted data and an overflow bucket for a total of 20 rows in the table. Instead of storing 20 index entries, PLI only contains 4; the first bucket covers first two pages with six rows – PLI structure knows that all indexed values in that range are between #1 and #10 (without knowing the exact order) and can direct the query to scan this range if the predicate matches. The following two pages belong to bucket two which includes range between #7 to #14; note that approximate nature of sorting can result in overlap between buckets, e.g., PLI does not know whether #8 is in the first or second bucket and will direct the query to scan both buckets for this value. Thus, PLI can conceptually tolerate any amount of out-of-orderness, but performance will deteriorate accordingly. In addition to the indexed buckets, we also include the overflow bucket (values [5–13]) which contains recent inserts.

We next discuss maintenance costs. Interestingly, PLI's approach requires no maintenance for deletes. Sparse bucket-based indexing knowingly permits false-positive matches that will be filtered out by the query after I/O was performed. Therefore, the index does not change when rows are deleted (e.g., in Figure 9.1, deletion of #6 will not change the first bucket in any way). Update queries can be viewed as DELETE + INSERT, permitting us to treat updates as insert as well.

A new insert would typically be appended at the end of table storage, unless there is unallocated space on one of the existing pages and the database is willing to make in-place overwrite (Oracle has a setting to control page utilization, while PostgreSQL avoids in-place overwrite inserts). If the insert is appended, the overflow bucket needs to be updated only if the range of values in the bucket changes. For example, in Figure 9.1 overflow bucket is [5–13] and thus does not need to be changed when #10 is inserted into overflow.

There are several ways to determine the location of the newly inserted row to update PLI (RowID is the internal database identifier that reflects location of the row). Our current prototype queries the DBMS for it (SELECT CTID in PostgreSQL or SELECT ROWID in Oracle). However, for bulk inserts it we can also use DBCarver to inspect the storage and determine the RowID ourselves. The new insert may overwrite a previously deleted row at any position (as we are avoiding maintenance overheads of clustering), which could
potentially widen range of values in that bucket creating more false-positives. The degrada-
dation is gradual, but eventually the table will need to be reorganized. The comparison
of different reorganization algorithms is beyond the scope of this chapter, but storage re-
organization can be done by targeting specific rows (executing commands that will cause
out-of-order rows to be re-appended) or by resorting the whole table.

The storage size and the cost to maintain the PLI structure is proportional to the number
of buckets that it uses. We have experimented with different granularities and bucket sizes
– and, in practice, having a bucket of fewer than 12 disk pages does not improve query
performance. Assuming about 80 rows per page, PLI structure only needs one bucket
per one thousand (1000) rows. A structure of this size can be kept in RAM and used or
maintained at a negligible overhead cost.

9.3.4 Query rewrite

In order to use PLI index, incoming SQL queries are rewritten to take full advantage of
the current layout of the table. Additional PLI-based predicates are added to the query to
restrict the disk scan range; bucket-based indexing is approximate by nature and provides
a superset range within which data of interested resides. For example, consider Figure 9.1
– a query predicate:

\[ id \text{ BETWEEN } \#1 \text{ AND } \#6 \]

is rewritten into:

\[ id \text{ BETWEEN } \#1 \text{ AND } \#6 \]
\[ \text{AND CTID BETWEEN Row1 AND Row6} \]
\[ \text{AND CTID BETWEEN Row19 AND Row20}. \]

In Oracle ROWID will be used instead of CTID in PostgreSQL. The first added condition
matches the range of buckets (in that case the first bucket from PLI) and the second condi-
tion corresponds to the overflow bucket. This access range results in a more efficient pattern
of disk reading by minimizing seeks and by removing the overhead of secondary index use.
While this PLI condition does include false-positives (specifically, id #10 at Row6 and #13
at Row20), the original query predicate (\( id \text{ BETWEEN } \#1 \text{ AND } \#6 \)) will eliminate false
positives.

9.4 Experiments

Due to limitations of available user access to the database-internal RowID attribute, our
experiments were limited to two databases, PostgreSQL and Oracle. We used data from the
Unified New York City Taxi Data Set \[76\]. The experiments reported here were performed
on servers with an Intel X3470 2.93 GHz processor and 8GB of RAM running Windows Server 2008 R2 Enterprise SP1 or CentOS 6.5.

9.4.1 Experiment 1: Regular Clustering

The objective of this experiment is to compare the performance of a table with a native clustered index and a table with a PLI. In Part-A, we collected query runtimes using a predicate on the sorted attribute. In Part-B, we compare the time to batch insert data into each table. In Part-C, we repeat the queries from Part-A.

Part A  We began with 16M rows (2.5GB) from the Green_Trips table sorted by the Trip_Distance column. For each DBMS, we created one table that implemented the native clustering technique and another table that implemented PLI. Since an Oracle IOT can only be organized by the primary key, we prepended the Trip_Distance column to the original primary key. We then ran three queries, which performed sequential range scans, with selectivities of 0.10, 0.20, and 0.30.

Table 9.1 summarizes the runtimes, which are normalized with respect to a full table scan (i.e., 100% is the cost of scanning the table without using the index). Since our goal is to evaluate a generalized database approach, the absolute time of a table scan is irrelevant; we are concerned with the runtime improvement resulting from indexing. In PostgreSQL, both approaches exhibited comparable performance, a few percent slower than the optimal runtime (e.g., for 0.20 selectivity the optimal runtime would be 20% of the full table scan). PLI remained competitive with native PostgreSQL clustering – the slight edge in PLI performance is due to not having the overhead of accessing the secondary index structure. PostgreSQL has to read the index and the table, while PLI access only reads the table (PLI structure itself is negligible in size). In Oracle, PLI significantly outperformed the IOT for the range scans. The queries that used a PLI were about three times faster than those that used an IOT. Oracle performance is impacted by lower average page utilization (and unused space) in the nodes of the IOT B-Tree.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Index Type</th>
<th>Query Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>Clustered</td>
<td>15% 26% 38%</td>
</tr>
<tr>
<td></td>
<td>PLI</td>
<td>13% 25% 36%</td>
</tr>
<tr>
<td>Oracle</td>
<td>Clustered</td>
<td>31% 57% 86%</td>
</tr>
<tr>
<td></td>
<td>PLI</td>
<td>12% 21% 32%</td>
</tr>
</tbody>
</table>

Table 9.1: Query runtimes as percent of a full table scan (clustered on attribute vs PLI).
Part B  Next, we bulk loaded 1.6m additional rows (250MB or 10% of the table) into each \texttt{Green\_Trips} from Part-A. In PostgreSQL, the records were loaded in 263 seconds for the table that implemented native clustering and 62 seconds for the table that implemented a \texttt{PLI}. Clustering is a one-time operation in PostgreSQL and ordering is not maintained as inserts are performed. Therefore, the observed overhead was primarily associated with the clustered index itself. A \texttt{PLI} does not have a significant maintenance cost due to its sparse and approximate nature. In Oracle, the records were loaded in 713 seconds for the IOT, and 390 seconds for the table that implemented a \texttt{PLI}. Since IOT used a B-Tree to order records, the observed high overhead was caused by maintenance of the B-Tree as new records were inserted.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Index Type</th>
<th>Query Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Clustered</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>\texttt{PLI}</td>
<td>23%</td>
</tr>
<tr>
<td>Oracle</td>
<td>Clustered</td>
<td>123%</td>
</tr>
<tr>
<td></td>
<td>\texttt{PLI}</td>
<td>20%</td>
</tr>
</tbody>
</table>

Table 9.2: Query runtimes as percent of a full table scan (clustered on attribute vs \texttt{PLI} after bulk insert).

Part C  To evaluate the maintenance approach for each index, we re-ran the queries from Part-A. Table 9.2 summarizes the resulting runtimes. For both DBMSes, the queries that used a \texttt{PLI} incurred a penalty of 10% or less compared to Part-A, which is consistent with Part-B inserting 10% worth of new rows. All newly inserted records were appended to the end of the table and were therefore incorporated into the overflow bucket (requiring minimal maintenance in the process and causing limited query performance deterioration). In PostgreSQL, the queries using the native clustered index slowed down by a factor of about 4 due to the interleaving seeks inefficiency discussed in Section 1. In Oracle, the queries using native clustering also slowed down by a factor of about 4, albeit for a different reason. While the IOT maintains logically sorted records within the leaf node pages, these leaf node pages are not necessarily ordered on disk during B-Tree re-organization resulting in an increased number of seeks for the queries.

9.4.2 Experiment 2: Expression Clustering

The objective of this experiment is to expand upon Experiment 1, and evaluate an expression based (rather than attribute-based) index to demonstrate the extendability and flexibility of the \texttt{PLI} approach. In Part-A, we collected query runtimes using a predicate on the
sorted attribute. In Part-B, we compare the time to batch insert data into each table. In Part-C, we re-run the same queries from Part-A.

**Part A** We began with 16M rows (2.5GB) from the `Green_Trips` table, and we sorted the table on `Tip_Amount / Trip_Distance` function (i.e., tip-per-mile for each trip as our order-preserving function). For each DBMS, we created one table that implemented the native clustering technique and another table that implemented PLI. As Oracle does not support function-based indexes, we created a computed column, and prepended this computed column to the primary key so an IOT could be built. We then ran three queries, which performed sequential range scans with selectivities of 0.10, 0.20, and 0.30.

Table 9.3 summarizes the runtimes, which are again normalized with respect to full table scan. These baseline performance results are very similar the result from Experiment 1: Part-A demonstrating that query access for the function based index does not impose a significant penalty for any of the approaches. The runtimes for the Oracle IOT were slightly higher, which we believe were caused by additional storage space used by the computed column.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Index Type</th>
<th>Query Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>Clustered</td>
<td>13% 25% 37%</td>
</tr>
<tr>
<td></td>
<td>PLI</td>
<td>15% 25% 37%</td>
</tr>
<tr>
<td>Oracle</td>
<td>Clustered</td>
<td>30% 62% 100%</td>
</tr>
<tr>
<td></td>
<td>PLI</td>
<td>11% 21% 32%</td>
</tr>
</tbody>
</table>

Table 9.3: Query runtimes as percent of a full table scan (clustered on expression-based index vs PLI).

**Part B** Next, we bulk loaded 1.6m additional rows (250MB or 10% of the table) into each `Green_Trips` from Part-A. For the Oracle IOT containing the computed column, we previously generated the value, and we stored it in the raw data file. In PostgreSQL, the records were loaded in 917 seconds for the table that implemented native clustering, and 70 seconds for the table that implemented a PLI. This demonstrates that a traditional expression-based index is far more expensive to maintain than a regular index, producing much higher overheads. PLI requires very minimal maintenance – same as in Experiment 1, without an expression-based clustering. The insert cost into the table itself is using append and is thus comparable for both. In Oracle, the records were loaded in 1527 seconds for the IOT, and 408 seconds for the table that implemented a PLI. This drastic overhead increase in the time to load the data (compared to Experiment 1: Part-B) can be explained
by data distributed. The data in Experiment 1 was more uniform requiring less B-Tree rebuilding, while computed ordering was much more scattered resulting in more B-Tree restructuring.

**Part C** To evaluate the maintenance penalties for each index, we re-ran the queries from Part-A as summarized in Table 9.4. Just as in Experiment 1, the queries that used PLI increased in cost by about 10% of a full table scan – as expected because inserted records were appended to the overflow bucket causing queries to scan additional 10% of overflow data. In PostgreSQL, the runtimes for the native expression-based clustered index increased by about a factor of 3 due to interleaving seeks as in Experiment 1. Interestingly, the penalty caused by computed index and storage fragmentation was not nearly as significant as regular built-in clustered index. We expect that PostgreSQL makes some additional effort to mitigate the overhead of interleaving seeks when utilizing an expression-based clustered index. In Oracle, the queries using the IOT increased by a factor of about 7, which is significantly more than Experiment 1: Part-C. This difference can be attributed to a greater amount of fragmentation caused by the B-Tree restructuring in Part-B.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Index Type</th>
<th>0.10</th>
<th>0.20</th>
<th>0.30</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>Clustered</td>
<td>52%</td>
<td>79%</td>
<td>93%</td>
</tr>
<tr>
<td></td>
<td>PLI</td>
<td>22%</td>
<td>31%</td>
<td>44%</td>
</tr>
<tr>
<td>Oracle</td>
<td>Clustered</td>
<td>259%</td>
<td>461%</td>
<td>706%</td>
</tr>
<tr>
<td></td>
<td>PLI</td>
<td>19%</td>
<td>30%</td>
<td>40%</td>
</tr>
</tbody>
</table>

Table 9.4: Query runtimes as percent of a full table scan (clustered on expression-based vs PLI after bulk insert).

9.5 Conclusion

We have presented PLI – a generalized clustered indexing approach that can be added to a live relational database using rowid column. This indexing approach uses a bucket-based sparse indexing structure, which results in a very lightweight and easy-to-maintain index. The sparse pointers into the table can easily tolerate approximate clustering (i.e., reordering within the bucket is irrelevant) and trivially allows PLI variations to use an expression-based index to match query predicate. DBMSes could expose rowid column further to make custom clustered index creation simple for the user – or this approach can be used to create a generation of better clustered indexes inside the database engine, as existing engines do not implement true (i.e., textbook-like) sparse clustering indexes.
Chapter 10

Future Work

10.1 Testing framework

10.1.1 Introduction

Prior to a digital examination, investigators select the proper forensic tool(s) based on the tool’s capabilities. For evidence to be usable (e.g., admissible in court), forensic tools must be tested using well-accepted benchmarks and repeatable environments. Additionally, testing is of the utmost importance for newly developed forensic tools, which have unproven functionality.

Database forensics is an emerging sub-field of digital forensics. As DBMSes are increasingly used in all facets of everyday life (e.g., cell phones, web browsers, web stores, banking), there is a growing demand for database forensic tools. However, due to the relatively young age of the field, datasets, workloads, and test environments are yet to be developed. Stable, controlled, and repeatable experiments are an undeniable requirement for scientific validation. Having a set of representative corpora enhances the scientific evaluation of forensic methods; they can be used as a baseline to evaluate the success of new database forensic tools and methods using objective metrics.

Testing environments should support benchmarks consisting of both a dataset and a corresponding workload to measure tool metrics. A good benchmark allows users to model real-world requirements. For example, law enforcement wants to know the data that can be collected from a cell phone where a suspected criminal receives, sends, and deletes SMS messages. Therefore, there is an obvious need for a comprehensive corpus that supports both a variety of DBMSes and a comprehensive range of benchmarks that model real-world scenarios.

As future work we propose to address this problem and contribute to repeatability and comprehensive database forensic tool testing. We propose two main contributions: 1) An
automated framework to setup, run, and collect storage for repeatable and controlled corpora creation to be used for database forensic tool evaluation. 2) We will use our framework to generate a public, multi-modal corpus. Our generated corpus will then be applied to a set of forensic software to evaluate the metrics of these tools and demonstrate the impact of our testing framework.

10.1.2 Related Work

We account for the work by Yannikos et al. [94] and Diffallah et al. [16] when defining our framework requirements and design. Yannikos et al. presented a framework to generate synthetic disk images for testing file carvers [94]. Similarly, we automatically capture storage based on user specified models. Differently, we capture disk images, RAM snapshots, network packets, and individual files (in addition to disk images), we incorporate accepted benchmarks (rather than generate user-specified models) within our system, and we collect statistics and metadata for testing database carvers. Diffallah et al. presented a testbed to benchmark relational DBMSes [16]. Similarly, our framework uses a workload manager to run a specified benchmark and collects statistics and metadata about the DBMS and server being tested. Differently, the statistics and metadata we collect are intended for forensic testing (rather than transaction processing), and we collect storage to maintain a forensic corpus.

Several real-world corpora exist for digital forensics [24, 23], but full knowledge of what forensic artifacts exist on them makes tool evaluation (especially in the emerging field of database forensics) difficult. Recently, Nemetz et al. presented a synthetic corpus to test SQLite DBMS forensic tools [53]. This corpus is the only publicly available database forensics corpus; our proposal makes some significant extensions and improvements to Nemetz’s work. First, they only collected the SQLite data files, whereas, we collect all storage (e.g., disk images, RAM snapshots) that contain other forensic data and metadata. Second, their corpus contains only the very basic database operations, whereas our framework supports any type of user-specified benchmark that model desired scenarios. Third, their work was done for only SQLite DBMS, whereas, our framework is designed to support all relational DBMSes.

10.1.3 Framework Description

Our proposed framework contains a synthetic workload manager to incorporate realistic datasets allowing users to model real-world scenarios, a set of global actions to create multi-modal corpora and control the rate of storage collection from a VM, and a defined
set of statistics and metadata that is automatically collected and included with a corpus for accurate and comprehensive tool testing. Figure 10.1 displays the framework overview. In a configuration file, the user specifies the benchmark for the workload manager (A) to use and the set of global action (B) to perform. While forensic data is generated, (C) statistics and metadata are collected to be included for later forensic tool evaluation.

The workload manager (A) is responsible for reading the user-provided benchmark (specified in the config file) and the benchmark parameters (e.g., transaction throughput, dataset size). Given these inputs, the workload manager generates a synthetic workload based on these parameters. The work in this proposal assumes a single client; future work will expand support to simulating a workload from multiple clients.

The global actions (B) are functions accessible anytime while the workload manager is in operation. These commands will include the ability to capture a disk image, RAM snapshot, files, and network files (i.e., a multi-modal set of forensic evidence). The ability to capture a set of heterogeneous forensic data allows for a wide variety of tools to be tested using our framework.

As the workload manager executes transactions and global actions are performed, the statistics and metadata collector (C) captures information to summarize and represent this activity. We will define the metadata and statistics that must be collected. Examples of relevant information include the number of records affected by a transaction, the amount of file system and database file slack space during a disk image export, and the buffer cache utilization during a RAM snapshot export.
We presented a prototype of this framework in [45] as SysGen. SysGen currently executes custom workloads and generates a corpus, which includes the DBMS files, full disk image, RAM snapshot, and network packets, for most relational DBMSes.

10.1.4 Public Corpus Generation and Framework Demonstration

We plan to use our framework to create public corpora to be hosted by the DePaul Data System Laboratory. These corpora will include several established benchmarks in database systems research (e.g., SEATS [18], TPC-C [19], LinkBench [20]) and at least two additional benchmarks created by us. Each benchmark will be run against multiple DBMSes (e.g., Oracle, PostgreSQL, SQLite, MySQL). Most database systems benchmarks do a good job of simulating OLTP (e.g., webstores) and OLAP (e.g., data warehouses) environments, but do not simulate the real-world environments experienced by most of our collaborators in law enforcement (e.g., the Chicago Regional Computer Forensics Lab, MITRE, Royal Canadian Mounted Police). We will create new benchmarks to simulate these environments. Specifically, our new benchmarks will model personal data uses, such as cell phones and web browsers.

10.2 Systematic Reverse Engineering

10.2.1 Introduction

Digital forensic tools enable investigators to reconstruct data (e.g., deleted files) and event timelines (e.g., sequence of actions in remote hack) following a security breach. A forensic analyst can use the reconstructed artifacts to detect subtle advanced persistent threats that skirt preventative security measures (e.g., access control or encryption), particularly in cases of insider attack. Accurate event timelines inform investigators on how to implement future preventative measures and what data was compromised.

The current state-of-the-art approach to digital forensic analysis is to reverse engineer specific systems, often for a specific purpose. Developing a dedicated forensic solution for each system (e.g., for each database, for each file system) is obviously problematic, but the reality is much worse. As new, unforeseen analysis approaches emerge or new versions of the same system are released (changing the internal storage structure), forensic investigators develop and publish multiple approaches for the same system. For example, SQLite database is of particular significance to forensics because it is used by mobile devices (e.g., SMS messages) and Internet browsers (e.g., preferences or bookmarks). Over
the last decade, researchers published three papers focusing only on algorithms for recovering deleted records from a SQLite database (starting from Pereira [66] in 2009). That does not include a generalized approach to reconstructing relational databases that included SQLite among others [86] and at least half a dozen SQLite recovery tools with varying capability (explored in [53]) and of unknown scientific provenance (not explored by anyone to our knowledge).

Reverse engineering a system (particularly without access to the source code) is an arduous task. However, it is usually not a new or a novel technique and creates a bottleneck for tool development. This means current digital forensic tools are in a constant state of ‘catch-up’ with new technologies. Despite all of the reverse engineering in digital forensics, little effort has been devoted to developing automated or systematic reverse engineering approaches. Complete (or even partial) automatic reverse engineering advances digital forensics through quicker tool development, composability, abstraction for analysis tools, and alternate analysis methods using heterogeneous data sources. Furthermore, automation anticipates new systems or system changes rather than waiting for new releases and constantly playing ‘catch-up’ – a great business model to keep forensic tool developers employed, but it is not always ideal for end users.

Future work makes a case for what we believe to be one of the most promising ways to design systematic approaches (and innovation) in digital forensics: tool parameterization. The goal of this work is to bring to light that parameterization is possible for forensic tools and that it should be explored more by serious digital forensic researchers. To describe this process, we broke it down into four steps: 1) define parameters that describe metadata and data layout, 2) create synthetic data to identify parameter values, 3) collect parameter values as the synthetic data forces changes to data structures, and 4) define how to use these parameters in carving tools.

10.2.2 Overview

DBCarver (and page carving) is our strongest example of systematic reverse engineering with parameterization. DBCarver describes all DBMS metadata and data with a set of parameters, rather than a tool for each DBMS. It was also designed to anticipate storage architecture changes in existing DBMSes or new DBMSes.

We also described a systematic framework to reverse engineer database memory in [84]. Rather than reconstructing data at the page-level as with DBCarver, this work describes how to reverse engineer four major DBMS memory areas regardless of the system configurations: I/O cache, sort area, query cache, and transaction buffer.
We propose that our work with DBCarver and in [84] can be translated to all systems (e.g., file systems or NoSQL DBMSes) and data structures (e.g., process control blocks or OS page tables). While some of these systems and data structures are used to manage unstructured data, the metadata used to describe the data will be structured and therefore has potential to translate to our systematic approaches in reverse engineering.
Chapter 11

Conclusion

This dissertation presented our novel database forensic method of page carving and an implementation as DBCarver. We then presented a standard storage format for database forensics called DB3F, and an API called ODSA to access data stored in DB3F. Next, we presented three applications that require low-level storage information that cannot be obtained through DBMS APIs and must use database forensics to obtain this data. The first application, DBDetective, seeks to detect malicious database activity that bypasses audit logs. The second application, DBStorageAuditor, seeks to detect illegitimate modifications to DBMS files made at the OS level. The third application, PLI, builds an index that supports approximate clustering. Finally, we presented future work that addresses more thorough testing of DBCarver, the applications that use its output, and database forensics in general. Future work also discusses expanding our novel methods in database forensics to all of digital forensics.
References


[81] SQLite. Sqlite database file format. 


