Short paper

ODSA: Open Database Storage Access

James Wagner
DePaul University
jwagne32@depaul.edu

Alexander Rasin
DePaul University
arasin@cdm.depaul.edu

Dai Hai Ton That
DePaul University
dtonthat@depaul.edu

Tanu Malik
DePaul University
tanu@depaul.edu

Jonathan Grier
Grier Forensics
jdgrier@grierforensics.com

ABSTRACT

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

In this paper, we present a specification of programmable access to relational DBMS storage. Open Database Storage Access (`ODSA`) is a simple, DBMS-agnostic, easy-to-program storage interface for DBMSes. We formulate novel operations using `ODSA`, such as comparing page-level metadata. We present three compelling use cases that are enabled by `ODSA` and demonstrate how to implement them with `ODSA`.

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 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 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 [3], enabling users to manually fix or skip corrupt pages, but such tools only access DBMS-controlled storage.

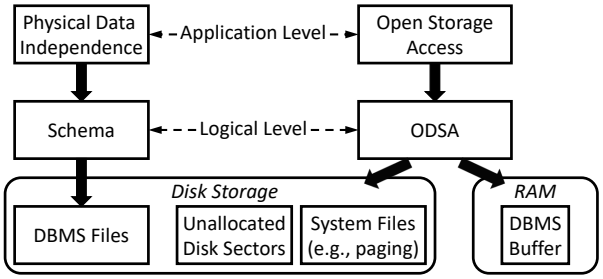

Figure 1: `ODSA` storage access.

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 [8], 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 1 shows the storage access enabled by `ODSA`.

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

The rest of the paper is organized as follows. Section 2 presents three representative uses cases that require storage-level access. Section 3 provides an overview of how applications previously had limited access to internal DBMS storage. Section 4 describes the hierarchy exposed by `ODSA` and how it provides a comprehensive view of storage. Section 5 demonstrates implementation and use of `ODSA`. Finally, Section 6 discusses future work for `ODSA`.

2 USE CASES

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

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 [7, 9] that detects such malicious activity requires comprehensive storage access to compare volatile storage with allocated and unallocated persistent storage.

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 [4].

2.3 Evaluating Data Retention

Continuing with Example 1 (Section 1), the bank validates their compliance with data sanitization regulations (e.g,. EU General Data Protection Regulation or GDPR [5]). 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 [2]. 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).

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. Wagner et al. previously extended carving to interpret DBMS storage with `DBCarver` [8, 10, 11], 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 [12] formatted file. This paper uses `DBCarver` to demonstrate the physical information a DBMS can provide.

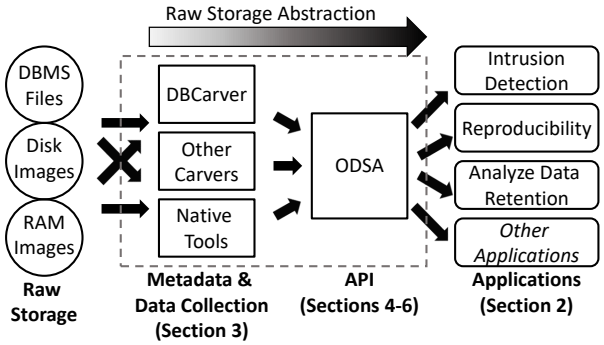

Figure 2: `ODSA` completes raw database storage abstraction in an end-to-end process for storage access.

4 OPEN DATABASE STORAGE ACCESS

Figure 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.

```
#4.A. Root
class Root:
    def __init__(self, db3f):
        #Initialize
    def get_object_ids(self):
        #Return a list of object ids
    #Calls to Other Instance and Namespace Data
#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 3: A sample set of ODSA calls.

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 3) and as a relational schema (Figure 4). The pure object hierarchy is stored as a JSON file in the DB3F format [12]. 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

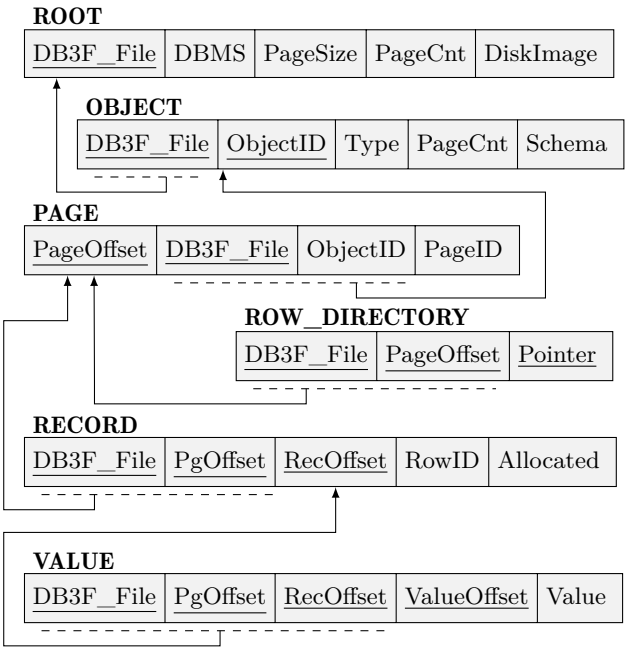

Figure 4: The relational schema used to store ODSA data.

may prefer to access a DBMS storage with SQL rather than calling the ODSA directly. However, as we show in Section 5 the SQL implementation requires several joins and is quite counter-intuitive, despite it being DBMS physical storage.

5 USING ODSA

For use cases in Section 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 5 finds and prints all unallocated (e.g., deleted) records from the *Customer* table. All ODSA calls are highlighted.

The implementation in Figure 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:

```
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;
```

```
1  DBRoot = odsa.Root('MyDatabase1.json')
2  CustomerTable = odsa.Object(DBRoot, 'Customer')
3  PageOffsets = CustomerTable.get_page_offsets()
4
5  for PageOffset in PageOffsets:
6    CurrPage = odsa.Page(CustomerTable, PageOffset)
7    RecordOffsets = CurrPage.get_record_offsets()
8
9    for RecOffset in RecordOffsets:
10     CurrRecord = odsa.Record(CurrPage, RecOffset)
11     allocated = CurrRecord.get_record_allocation()
12     #print unallocated (e.g., deleted) record
13     if not allocated:
14       print CurrRecord
```

Figure 5: Using ODSA to find deleted records.

```
1  def findIndexEntries(record, Index):
2    RecordPtr = record.get_record_pointer()
3    IndPageOffsets = Index.get_page_offsets()
4
5    for IndPageOffset in IndPageOffsets:
6      IndPage = odsa.Page(Index, IndPageOffset)
7      IndROffsets = IndPage.get_record_offsets()
8
9      for IndROffset in IndROffsets:
10       IndEntry = odsa.Record(IndPage, IndROffset)
11       # IndEntry is a pair (Value, Pointer)
12       IndexPointer = odsa.Value(IndEntry, 1)
13       if IndexPointer == RecordPtr:
14         print IndEntry
```

Figure 6: Using ODSA to find all index entries for one record

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, 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 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, 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.

```
SELECT V1.Value
FROM Page NATRUAL 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*/);
```

6 CONCLUSION

ODSA was designed based on the principles and challenges described in [1, 6]. 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 3^{rd} 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.

ACKNOWLEDGMENTS

This work was partially funded by the US National Science Foundation Grants CNF-1656268 and CNS-1846418.

REFERENCES

[1] Joshua Bloch. 2006. How to design a good API and why it matters. In *ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 506–507.
[2] Intl. Data Sanitization Consortium. 2019. Data Sanitization Terminology. https://www.datasanitization.org/ data-sanitization-terminology/.
[3] Oracle. 2019. Database Administrator's Guide: Repairing Corrupted Data. https://docs.oracle.com/database/121/ADMIN/repair.htm#ADMIN022
[4] Quan Pham, Tanu Malik, Boris Glavic, and Ian Foster. 2015. LDV: Light-weight database virtualization. In *IEEE International Conference on Data Engineering*. IEEE, 1179–1190.
[5] General Data Protection Regulation. 2016. Regulation (EU) 2016/679. *Official Journal of the European Union (OJ)* 59, 1-88 (2016), 294.
[6] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.
[7] James Wagner et al. 2017. Carving database storage to detect and trace security breaches. *Digital Investigation* 22 (2017), S127–S136.
[8] James Wagner et al. 2017. Database forensic analysis with DBCarver. In *Conference on Innovative Data Systems Research*.
[9] James Wagner et al. 2018. Detecting database file tampering through page carving. In *21st International Conference on Extending Database Technology*.
[10] James Wagner, Alexander Rasin, and Jonathan Grier. 2015. Database forensic analysis through internal structure carving. *Digital Investigation* 14 (2015), S106–S115.
[11] James Wagner, Alexander Rasin, and Jonathan Grier. 2016. Database image content explorer: Carving data that does not officially exist. *Digital Investigation* 18 (2016), S97–S107.
[12] James Wagner, Alexander Rasin, Karen Heart, Rebecca Jacob, and Jonathan Grier. 2019. DB3F & DF-Toolkit: The Database Forensic File Format and the Database Forensic Toolkit. *Digital Investigation* 29 (2019), S42–S50.