Programmable Access to Relational Database Storage

James Wagner
DePaul University

Alexander Rasin
DePaul University

Dai Hai Ton That
DePaul University

Tanu Malik
DePaul University

Jonathan Grier
Grier Forensics

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. 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 database 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 show examples of how to use `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 of the DBMS consists only of a set of relations (i.e., the data). A physical view of the DBMS, however, consists of several objects, such as pages, records, directory headers, etc. Hiding physical representation is fundamental to the design of relational DBMSes: DBMSes transparently control physical data layout and manage auxiliary objects in order to provide efficient query execution. This data independence, however, inhibits several security and performance related applications requiring low-level storage access. We provide a small example here, giving more detailed use cases of application in Section 2.

Consider, for example, a bank or a hospital that handles sensitive customer data using a commercial database but for audit purposes must sanitize deleted customer data to ensure that it *cannot* be recovered and stolen. Very few DBMSes offer support for explicit sanitization of deleted data (e.g., `secure delete` in SQLite exists but provides no guarantees or feedback to the user)[1]. In order to programmatically verify that deleted data cannot be reconstructed, 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.

Enabling comprehensive storage-level access is an inherent DBMS challenge because of the way DBMSes control storage. DBMSes control *allocated* storage objects such as a) physical byte representation of the relations, b) metadata that annotates physical storage of relation data, c) auxiliary objects associated with relations (e.g., indexes, materialized views). The user can manipulate allocated objects exposed by SQL. However, as illustrated in the example, the DBA often also needs access to *unallocated* storage objects not tracked by a DBMS such as deleted data that lingers in DBMS-controlled files, and DBMS-formatted pages that are released back to the OS and no longer under DBMS control (e.g., files deleted by the DBMS, 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 blocks in Oracle storage. This access, however, is not inclusive of all storage.

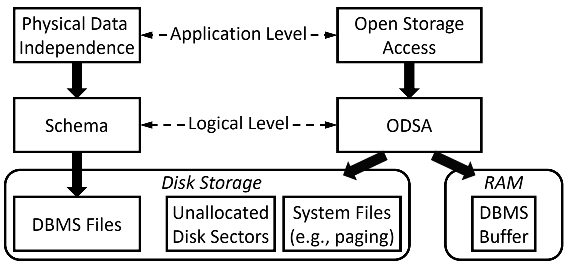

Figure 1: `ODSA` storage access.

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 [7]. We use carving as method to expose physical level details. But carving itself is not sufficient. Carved data consists of disk-level physical details making it difficult to program database storage. `ODSA` abstracts low-level disk-level details with a hierarchical view of database storage that most DBAs are familiar with. In particular it organizes them into pages, records, and values and internally resolves them to physical addresses. `ODSA` also guarantees that the same hierarchy can be used against multiple DBMS storage engines, so as to ensure 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 our API, `ODSA`, and how it includes both allocated and unallocated storage. Section 5 demonstrates how to implement and use the `ODSA`. Finally, Section 6 discusses future work related to `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 information. Unbeknownst to the bank, a disgruntled system administrator modified the DBMS data file bytes at the file system level. This activity bypassed all DBMS access control and logging, and still effectively altered account balances. The system administrator also turned off file system journaling with `tune2fs` to further hide their activity. The bank is not able to determine the cause for inconsistencies with the log files alone. Forensic analysis [8] that detect such malicious activity require per page storage access in order to compare volatile storage with persistent allocated and unallocated 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 must build a container consisting of necessary and sufficient data for Bob to reproduce. If the shared data is much smaller than original database file, Bob will not be able to 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 run and Bob's verification run, data layout specification at the record and page level must itself be ported. Currently, data layouts as part of a shared database file in a container cannot be communicated [4].

2.3 Evaluating Data Retention

Following on the example in Section 1, consider a bank which must validate their compliance with data sanitization regulations (e.g,. EU General Data Protection Regulation or GDPR [5]). To comply the bank, after deleting data, must independently validate that data is indeed deleted. There are no validation guidelines for DBMS beyond a complete overwrite of the entire file [2]. This is considered too coarse a guideline, especially for DBMS files, which may have only deleted a few rows.

Alternatively, consider if the compliance officer has programmatic access to database storage via `ODSA` for validation. The officer can easily access all unallocated storage, determine its location (e.g., the database file or an OS paging file) and then verify if unallocated values cannot be found in OS or memory or any auxiliary structures (e.g., materialized views or B-Tree index).

3 RELATED WORK

We describe built-in tools and interfaces available as part of popular RDBMSes, which provide physical storage information at different granularities, but none provides a comprehensive view of storage. The ROWID pseudo-column represents the physical location of a record within DBMS storage (not disk), and is one of the simplest examples of storage-based metadata users can access through almost all RDBMSes. Most commercial DBMSes offer utilities to inspect and fix page level corruption. Examples include Oracle's `DBMS_REPAIR`, Oracle's `BBED` (page editing tool available from Oracle 7 to Oracle 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 would have to manually make such interpretations. Moreover, no RDBMS offers access to unallocated storage. Finally, existing tools only support analysis of persistent storage. `ODSA` is designed to offer 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.

A comprehensive way to acquire detailed DBMS storage information is carving at byte level. Carving retrieves both allocated and unallocated storage, as we have previously shown with `DBCarver` [7]. However, carving is post hoc in that access to full physical DBMS storage is only available after interpreting storage. In this paper, we use `DBCarver` to demonstrate the types of physical information that an RDBMS can provide.

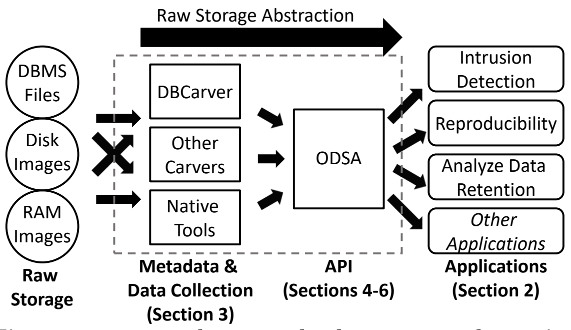

Figure 2: `ODSA` completes raw database storage abstraction in an end-to-end process for storage access.

4 OPEN DATABASE STORAGE ACCESS

As shown in Figure 2, `ODSA` relies on carving to obtain access to raw storage. `ODSA` abstracts two specific details from the raw storage.

First, it interprets each sequence of raw bytes and classifies it into one of physical storage element, *viz.* **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 that describe a DBMS object, such as a list of pages or column data

```
#4.A. Root
class Root:
   def __init__(self, db_file):
      #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.

types. Pages are uniquely identified by their byte offset in raw storage, rather than the PAGEID. We also do not rely on the row directory pointer within the page because record deletion may be performed by writing NULL in the row directory entry or the page itself may be NULLed.

Second, in providing hierarchical access to each physical element, ODSA hides DBMS heterogeneity by identifying pages and records within storage through physical byte offsets instead of DBMS-specific pointers. Computing the value of a DBMS pointer varies between different vendors. For example: Oracle incorporates FileID into index pointer while PostgreSQL does not; index pointer structure in MySQL differs from both Oracle and PostgreSQL because MySQL relies on index organized tables. Even if all vendors encoded the pointer similarly, abstraction is needed in terms of pages since duplicate pages may exist in the 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$_A$ and the contents of Page$'_A$.

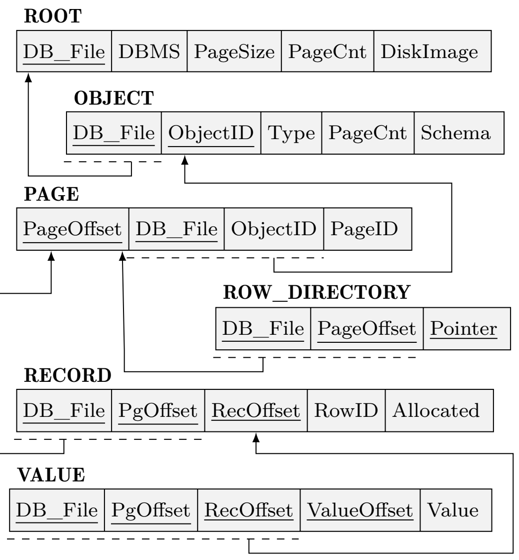
ROOT
DB_File | DBMS | PageSize | PageCnt | DiskImage
OBJECT
DB_File | ObjectID | Type | PageCnt | Schema
PAGE
PageOffset | DB_File | ObjectID | PageID
ROW_DIRECTORY
DB_File | PageOffset | Pointer
RECORD
DB_File | PgOffset | RecOffset | RowID | Allocated
VALUE
DB_File | PgOffset | RecOffset | ValueOffset | Value

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

Implementation. There are multiple ways to implement the hierarchy. We have currently implemented the ODSA hierarchy 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 [9]. 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 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 to find unallocated records and to match index pointers to records in DBMS. We use ODSA calls to enable these operations and show how to achieve these operations 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.

Example1: 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 name of the carved database file and the name of the table (the *Customer* table in this example) from which unallocated records are being sought. 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 three ODSA calls to search for unallocated records: Line 3 retrieves page offsets, which uniquely identify pages. The code then iterates through the pages in Line 5, loads each page in Line 6, and retrieves the record offsets for that page in Line 7. Finally, the code iterates through records using their identifying offsets within a page. The record allocation status is retrieved in Line 11 to identify and print unallocated records.

```
DBRoot = odsa.Root('MyDatabase1.json')
CustomerTable = odsa.Object(DBRoot, 'Customer')
PageOffsets = CustomerTable.get_page_offsets()

for PageOffset in PageOffsets:
  CurrPage = odsa.Page(Table, PageOffset)
  RecordOffsets = CurrPage.get_record_offsets()

  for RecOffset in RecordOffsets:
    CurrRecord = odsa.Record(CurrPage, RecOffset)
    allocated = CurrRecord.get_record_allocation()
    #print unallocated (e.g., deleted) record
    if not allocated:
      print CurrRecord
```

Figure 5: Using ODSA to find deleted records.

The same search and retrieval requires an 8-way join in SQL due to joining the hierarchy between `Object`, `Page`, `Record`, and `Value` tables:

```sql
SELECT PageOffset, RecordOffset, ValueOffset, Value
FROM Object, Page, Record, Value
AND Object.DB_File = Page.DB_File
AND Object.ObjectID = Page.ObjectID
AND Page.DB_File = Record.DB_File
AND Page.PageOffset = Record.PageOffset
AND Record.DB_File = Value.DB_File
AND Record.PageOffset = Value.PageOffset
AND Record.RecordOffset = Value.RecordOffset
AND Object.DB_File = 'MyDatabase1.json'
AND Object.ObjectID = 'Customer'
AND Record.Allocated = FALSE;
```

Example2: Matching a Record to Index Pointers. To match a record to a database object such as an index, the user must provide as input specific instances of the record and index objects. The code in Figure 6 iterates through all index pages (Line 2) and index records to determine if the input record matches. Recall, in an index, records are value-pointer pairs. So the code determines offsets of all index pages (Line 4), and for each index page (Line 5), determine record offsets (Line 6 and 7) to finally load the value-pointer pair index record (Line 10), and its corresponding pointer (Line 12). Finally, for any index pointer match to the record pointer (Line 13), the index entry is printed.

In this example 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. This is necessary because DBMS indexes often contain records of entries that have been 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 customer changed their name from Jehanne to Jane as well as *(Bob, {PAGEID: 12, ROWID: 37})* if another customer named Bob had previously deleted their account freeing up the space for Jane's record in 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 exactly match the values. Moreover, this query assumes that that there is only one indexed column which is transparently accounted for in the abstraction of the

```
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(Table, 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

DBMS Object class.

```sql
SELECT V1.Value FROM Page, Record, Value V1, Value V2
WHERE Page.DB_File = Record.DB_File
AND Page.PageOffset = Record.PageOffset
AND Record.DB_File = V1.DB_File
AND Record.PageOffset = V1.PageOffset
AND Record.RecordOffset = V1.RecordOffset
AND Record.DB_File = V2.DB_File
AND Record.PageOffset = V2.PageOffset
AND Record.RecordOffset = V2.RecordOffset
AND Page.ObjectID = ? --Index name placeholder
AND V1.ValueOffset = 0 --Indexed value is 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.

REFERENCES

[1] Joshua Bloch. 2006. How to design a good API and why it matters. In *OOPSLA*.
[2] Intl. Data Sanitization Consortium. 2019. Data Sanitization Terminology. https://www.datasanitization.org/.
[3] Oracle. 2019. Using DBMS_REPAIR. https://docs.oracle.com/cd/B19306_01/server.102/b14231/repair.htm.
[4] Quan Pham et al. 2015. LDV: Light-weight database virtualization. In *ICDE*.
[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* (2009).
[7] James Wagner et al. 2017. Database Forensic Analysis with DBCarver. In *CIDR*.
[8] James Wagner et al. 2018. Detecting Database File Tampering through Page Carving. In *EDBT*.
[9] James Wagner et al. 2019. DB3F & DFToolkit: The Database Forensic File Format and Database Forensic Toolkit. In *DFRWS*.