Where Provenance in Database Storage

Alexander Rasin
DePaul University
Chicago, IL, USA
arasin@cdm.depaul.edu

Tanu Malik
DePaul University
Chicago, IL, USA
tanu@cdm.depaul.edu

James Wagner
DePaul University
Chicago, IL, USA
jwagne32@cdm.depaul.edu

Caleb Kim
DePaul University
Chicago, IL, USA
hkim85@mail.depaul.edu

Abstract

Where provenance is a relationship between a data item and the location from which this data was copied. In a DBMS, a typical use of *where* provenance is in establishing a copy-by-address relationship between the output of a query and the particular data value(s) that originated it. Normal DBMS operations create a variety of auxiliary copies of the data (e.g., indexes, MVs, cached copies). These copies exist over time with relationships that evolve continuously – A) indexes maintain the copy with a reference to the origin value, B) MVs maintain the copy without a reference to the source table, C) cached copies are created once and are never maintained. A query may be answered from any of these auxiliary copies; however, this *where* provenance is not computed or maintained. In this paper, we describe sources from which forensic analysis of storage can derive *where* provenance of table data. We also argue that this computed where provenance can be useful (and perhaps necessary) for accurate forensic reports and evidence from maliciously altered databases or validation of corrupted DBMS storage.

Keywords Where Provenance, Database Forensics, DBMS Tampering Detection, DBMS Internals

ACM Reference Format:
Alexander Rasin, Tanu Malik, James Wagner, and Caleb Kim. 2018. Where Provenance in Database Storage. In *Proceedings of TaPP 2018.* ACM, New York, NY, USA, 4 pages. https://doi.org/

1 Introduction

Where Provenance is defined as the addresses of the data values that were used to evaluate the query. It is similar to *Why Provenance* in tracing query inputs, but focuses on the location of that data. In the relational model, value location is defined as the row (tuple) and the value's location within that row. We propose to extend this concept to support database forensic analysis by computing *where* provenance based on the physical address of data copies in DBMS storage.

Database Management Systems (DBMSes) generate a multitude of data copies as part of their normal operation. For example, a materialized view (MV) stores the pre-computed results of a query drawn from the data tables in order to improve query performance. An index contains a copy of values from the indexed column(s) combined with a pointer back to the source table in order to speed up record access. Many other copies of data are created by DBMS engine actions such as caching, log entries, or internal storage defragmentation.

These and other internal copies of data can be extracted from DBMS storage with the help of *database carving* (briefly described in Section 2) and used for evidence of database tampering or storage corruption. Such findings must be supported by a forensic analysis framework that integrates *where* provenance to formalize storage analysis and offer provable results. Recent work by Wagner et al. [2] relied on ad-hoc case analysis (e.g., if the index value does not match the value in table record, report this as a likely indication of tampering) to report malicious activity. Such reports currently require significant effort from forensic analysts – we describe two recent cases that would greatly benefit from integration of *where* provenance into the process of forensic analysis:

Example 1 A consultant from Mandiant/FireEye (a major forensic consultant firm) was working on a case involving a hard drive captured from the suspect. Through manual inspection of drive image, he came to suspect that the drive contained a PostgreSQL database that was uninstalled by the owner. Reconstructing raw data was the first step – but if the case went to court, the analyst could use *where* provenance to prove that the reconstructed data report is accurate. Some of the forensic artifacts are more reliable than others, depending on how they were extracted.

Example 2 A forensic analyst from Royal Canadian Mounted Police was investigating a financial fraud case. One of the sources of evidence was a snapshot of RAM from suspect's computer that contained a MySQL database (the snapshot of the hard drive was never recovered in this case). While

RAM can contain data from DBMS tables, all of the in-RAM values are *copies* of the original tables. In order to establish MySQL data contents from RAM snapshot with a measure of confidence, a *where* provenance derivation could be used.

In addition to these examples, there are other security and audit applications of *where* provenance that we outline in Section 3. Fully deriving and continuously tracking where provenance remains a goal for future work. In this paper, we describe the categories of data copies created within all major DBMSes. We focus on the causal relationship between the tables and auxiliary structures in DBMS storage, describing copies in the context of **active** data (Section 4), **accessible** data (Section 5), and **abandoned** data (Section 6).

2 Background and Related Work

All relational databases store data in page units of fixed size – even logs are often stored in system tables. Pages in all major relational databases (known for IBM DB2, SQL Server, Oracle, PostgreSQL, MySQL, Apache Derby, MariaDB, and Firebird) follow the same basic layout structure. Pages are broken down into page header (relevant page metadata such as page ID or page type), row directory (pointers to individual rows in the body of the page), and page body (containing the payload with actual row data). The work in [3] described how this layout can be generally parameterized, reconstructed and even automatically learned by loading synthetic data and observing page storage behavior. Database page carving (implemented as DBCarver [4]) is a method based on this analysis that reconstructs database file contents without relying on the file system or DBMS. Page carving is similar to traditional file carving [1] in that data, including deleted data, is reconstructed from disk images or RAM snapshots without using a live system. As database carving approach does not rely on the DBMS itself, it is also capable of extracting the non-queryable data values, which include: a) index values and pointers, b) deleted records, including partially overwritten records, c) cache contents, including pages and intermediate query results, d) audit logs.

3 Motivating Where Provenance in Database Management Systems

A forensic analysts will seek to discover and prove what is or was previously stored in the database tables, or to determine what actions user may have undertaken within the DBMS. While traditional provenance explains query output by investigating the data sources and the computation process of the query, in forensic cases the target of analysis is the data table itself. For each additional data copy (index, MV, RAM), *where* provenance of that copy will serve as support and evidence for contents of the original table.

Figure 1 represents the overall flow of data copying that occurs inside a DBMS engine. After user data is loaded into tables (data loading process can create extra copies in RAM

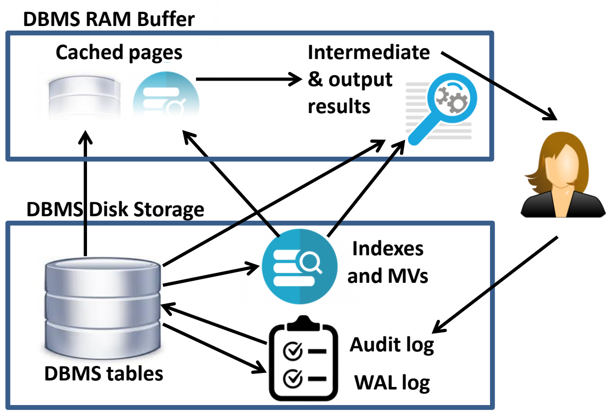

Figure 1. The causality flow of data copying in a DBMS.

or logs, depending on DBMS settings), every access to these tables will cause more copying. A SQL command is initially copied into the audit log; after the query is logged, it proceeds to access (or modify) the tables. Table access affects several parts of DBMS storage: modifications propagate into WAL, both read and write access caches pages in RAM (additionally copied in RAM as intermediate results), and all auxiliary structures are modified and cached in a similar manner.

The goal of this paper is to describe the copies that occur along the flow arrows in Figure 1. Computing *where* provenance (not available in DBMSes) would also require *reversing* the arrows by extrapolating the connection back to the table. For example, a record found in a cached page is evidence of a tuple having been present (at some point) in a source table. The location of the cached record is known, but *where* provenance also needs the link between that copy and the original table record. Note that the original table record may already be *deleted* (deleted values could be restored) or even *erased* (cannot be restored) in which case *where* provenance offers evidence for the source data that ceased to exist.

The second application for *where* provenance is tracing back the arrow between audit log and data tables. The idea is that each forensic artifact (e.g., a deleted row) must have been caused by *some* SQL command. User commands (in Figure 1) recorded in audit log cause changes to data tables. Therefore, if we find a storage artifact (e.g., a deleted row) that does not link back to an audit log entry (a delete or an update), this could be interpreted as a sign of log tampering.

4 Actively Maintained Data Copies

The first category of in-DBMS copies we consider are the values which the DBMS not only copies but also continuously maintained during its normal operations.

4.1 Index over Table Values

An index is a structure containing value-pointer pairs used for faster data look up; thus, each index entry already contains a pointer to the value(s) in the source table. Moreover, in response to updates the DBMS will automatically update the pointer (we discuss deletes in Section 6).

4.2 MV Values

Materialized views are a computed query results which are stored and maintained by the DBMS over time. After a user updates one of the source tables from which MV is derived, the DBMS will (eventualy) propagate the change into the MV. The refresh behavior of the MV is dependent on both the DBMS and its settings. There are three types of MV refresh options: 1) a custom refresh function, 2) refresh on each transaction commit, 3) refresh on demand.

4.3 Index over MV Values

Indexes created over MVs behave like indexes over tables. From *where* provenance perspective, as our goal is to generate evidence regarding the contents of the original table, an MV index is 2 steps away from the source table. That is, an MV index refers to where it was copied from – however, the MV itself is also a copy of the original table. To compute *where* provenance, a connection from an MV index to the MV needs to be established, after which the MV to table connection (as in Section 4.2) needs to be traced as well.

4.4 Cached Pages

When a page is modified, a "dirty" copy is created in RAM to hold pending data changes. Additional changes to the same page will be applied in RAM (assuming transactional restrictions are not violated). Thus, the entire dirty page is an automatically maintained data structure while it remains active in RAM and before it is flushed to disk.

5 Accessible Data Copies

In this section we discuss the middle-ground copies that are not kept current by the DBMS (i.e., can differ from the original value after having been copied) but are explicitly queryable by the user. Such data items include old MV values, audit and WAL logs – intuitively, these are user-accessible copies of data that can become "outdated" reflecting a previous value that has since changed.

In most DBMSes, an MVs may refresh with some delay (see Section 4.2). If the refresh is not immediate, the old pre-update or pre-delete values will persist in the MVs for a period of time and can be linked back to the source table in the same way as in Section 4.2. These data copies are ephemeral as the MV will eventually be refreshed. Note that once a table delete is propagated into the MV, it remains a source of evidence but stops being queryable (i.e., after MV refresh such value transitions from **accessible** to **abandoned**).

Audit and WAL logs are typically represented as system tables (although in some DBMSes, such as PostgreSQL or MySQL, they are merely text files) and can be explicitly queried by users with sufficient privileges. In an audit log, the entire text of the SQL query is stored – which includes the operation in question, and some of the affected values. For insert operation the entire row would normally be available; for an update operation the `WHERE` clause and the new copy of the new values (`SET` clause) is available; finally, for delete operations only the `WHERE` clause is available. In a WAL log, every modified value is stored for maintaining ACID properties of transaction. Each available copy of the logs can be traced back to the original record in the table, unless the affected values have since changed.

6 Abandoned Data Copies

In this section, we consider a multitude of value copies that remain physically in storage (disk or RAM), but are no longer accessible by users. These copies are the richest source of provenance because all DBMS operations generate abandoned data copies; however, they are also the most volatile because they can be overwritten without notice.

6.1 Deleted Table Values

When a record is deleted from a DBMS table, the values are not erased; rather, the value is flagged as deleted within the page. An update operation may be performed in-place or translated into a physical delete (acting as a regular delete) followed by an insert of the updated row.

Deleted values can serve as evidence of records that previously existed in the table, but will eventually be overwritten by other data. The survival duration of deleted records in storage is based on the DBMS and its settings. For example, Oracle offers a `percent utilized` threshold which determines when in-page storage is reclaimed by incoming inserts. In contrast, MySQL is much more aggressive about reclaiming de-allocated page space.

6.2 Deleted MV Values

An MV follows a pattern similar to that of table storage. Both deletes and updates are propagated from original table and applied to the MV. The difference between the MVs and tables is in the refresh policies – as discussed in Section 4.2, MV update propagation lags behind tables and thus offers a different timeline of events. Moreover, a deleted value that was purged (overwritten or rebuilt) from the table may still remain in the MV. We note that refresh settings of MV discussed in Section 4.2 do *not* have any effect on purging or overwriting deleted values stored in the MV. The deletion of the row is the action that is eventually propagated into the MV; once the row is deleted in the MV, it would follow the rules of deleted table values in Section 6.1.

6.3 Deleted Index Values

A deleted index value behaves differently from all other structures. Although theoretically indexes follow the B-Tree maintenance protocols, in practice *all* DBMSES [3] are implemented to leave the deleted values in the index storage. This saves on the maintenance cost (the B-Tree does not need to shrink and re-balance), but introduces the extra cost through increased B-Tree size. The decision on whether an index entry has been deleted can only be made by checking the row to which the index points (which, if still present in the storage, would be explicitly flagged as "deleted").

Deleted index entries serve as evidence of values that previously existed in the table or MV. Index values have perhaps the longest lifespan of all abandoned data because hey are not free listed in the traditional sense.

6.4 Cached Values

Nearly every access to a DBMS structure will generate cached pages in RAM. There are only two exceptions to that rule: 1) a query access that reads "too many" (subject to DBMS internal caching policies) pages will avoid caching pages, and 2) a "direct" data load in some DBMSes (e.g., in Oracle) will leave no trace of the loaded data in RAM.

Cached pages that are being modified (dirty pages) will be maintained while the cached copy is active – the new changes will be applied to active dirty pages in memory. Eventually, modified cached pages are flushed to disk and become free-listed (i.e., discarded). Cached pages that result from read-only (e.g., `SELECT`) access will eventually become discarded as well subject to DBMS caching policies or due to page contents changing (making it outdated).

6.5 Discarded Pages

Discarded pages are pages that have been free-listed and may be overwritten by other pages, similarly to a deleted row that may be reclaimed by new inserts. Defragmenting operations (e.g., `VACUUM`, `REBUILD`) leave behind old page copies on disk because they often create a new structure and then discard the old one. The specific pages left behind depends on a particular DBMS. Some of the pages may be discarded into Operating System custody – for example, dropping a table in PostgreSQL deletes the file corresponding to that structure.

7 Forensic Evidence in Where Provenance

Once *where* provenance of data copies is computed, it will be unified into a report describing 1) the data values contained within the target of the investigation, 2) the relative confidence in each reported value, and 3) an extrapolated timeline information for each data value.

The target of the investigation can be either user data tables or WAL log – due to space limitations we only discuss the former. For data tables, Part-#1 would include *every value and every record* for which some evidence of existence (at any time) was identified. This will include data from primary evidence sources (data tables), secondary evidence sources (cached table pages, indexes, MVs), tertiary evidence sources (indexes over MVs, cached index pages), and so on. In cases like Example 2 in Section 1 (only RAM data is available), the entire report will be based on secondary evidence or lower.

A reported value may derive from conflicting facts (e.g., on-disk table page and in-RAM cached page disagreeing on what the value was). Part-#2 would therefore seek to unify multiple reports about each value. A value with multiple agreeing sources would have higher confidence; a value with disagreeing or lower tier (e.g., tertiary) sources would have a relatively low confidence. Most importantly, confidence report should include reasons for how it was derived.

Finally, Part-#3 would further annotate all reported values with known timeline information. Evidence of each reported value will be associated with the time range during which it (likely) existed. For example, audit logs may help determine the exact time when the value was created and subsequently deleted. Alternatively, a deleted record in a page would indicate that the values were deleted in the past, but the time of the deletion could only be approximated.

8 Conclusion

DBMS storage is a rich source of data copies created during normal operations and accessible through forensic analysis techniques. These copies can serve as evidence of database state or proof of data tampering or log tampering. *Where* provenance is the mechanism that can create a formal analytical framework to explain and quantify accuracy and of the forensic evidence reliability drawn from storage analysis.

A report of all known data augmented with confidence rating and timeline knowledge will no doubt greatly help forensic and security analysts in their job. Copies of the data are available – but these copies lack the connection to their source; in order to reason about the evidence they offer, copy flow in DBMS storage must be reverse engineered.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1656268. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Golden G Richard III and Vassil Roussev. 2005. Scalpel: A Frugal, High Performance File Carver.. In *DFRWS*. Citeseer.

[2] James Wagner, Alexander Rasin, Boris Glavic, Karen Heart, Jacob Furst, Lucas Bressan, and Jonathan Grier. 2017. Carving database storage to detect and trace security breaches. *Digital Investigation* 22 (2017), S127–S136.

[3] 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.

[4] James Wagner, Alexander Rasin, Tanu Malik, Karen Hart, Hugo Jehle, and Jonathan Grier. 2017. Database Forensic Analysis with DBCarver.. In *CIDR*.