Forensic Science International: Digital Investigation 53 (2025) 301929

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA

ANOC: Automated NoSQL database carver

Mahfuzul I. Nissan [a,*], James Wagner [a], Alexander Rasin [b]

[a] University of New Orleans, New Orleans, LA, USA
[b] DePaul University, Chicago, IL, USA

ARTICLE INFO

Keywords:
Database forensics
Digital forensics
NoSQL forensics
Memory forensics
Database carving
Reverse engineering

ABSTRACT

The increased use of NoSQL databases to store and manage data has led to a demand to include them in forensic investigations. Most NoSQL databases use diverse storage formats compared to file carving and relational database forensics. For example, some NoSQL databases manage key-value pairs using B-Trees, while others maintain hash tables or even binary protocols for serialization. Current research on NoSQL carving focuses on single-database solutions, making it impractical to develop individual carvers for every NoSQL system. This necessitates a generalized approach to forensic recovery, enabling the creation of a unified carver that can operate effectively across various NoSQL platforms.

In this research, we introduce Automated NoSQL Carver, ANOC, a novel tool designed to reconstruct database contents from raw database images without relying on the database API or logs. ANOC adapts to the unique storage characteristics of various NoSQL systems, utilizing byte-level reverse engineering to identify and parse data structures. By analyzing storage layouts algorithmically, ANOC identifies and reconstructs key-value pairs, hierarchical storage structures, and associated metadata across multiple NoSQL platforms.

Through extensive experimentation, we demonstrate ANOC's ability to recover data from four representative key-value store NoSQL databases: Berkeley DB, ZODB, etcd, and LMDB. We explore ANOC's limitations in environments where data is corrupted and RAM snapshots. Our findings establish the feasibility of a generalized carver capable of addressing the challenges posed by the diverse and evolving NoSQL ecosystem.

1. Introduction

Addressing the current state of cybersecurity threats requires the continuous development of tools to detect and prevent the advanced, sophisticated threats. Detection tools, specifically, rely on a history of events in audit logs. However, in a compromised system, the accuracy of such logs should also be questioned - logs are subject to tampering or bypassed altogether, e.g., temporarily disabled during malicious activity. At the same time, analyzing persistent storage and memory at the byte-level offers a trusted view of even a compromised system - operating system design inherently abstracts such details from users and applications, making it difficult to impossible to falsify such information. A trusted byte-level analysis can then be used to verify the accuracy of audit logs or potentially build a separate, independent timeline of events, detailing data involved in a breach or how a breach occurred (Nissan et al., 2023; Nissan; Wagner et al., 2023).

Database management systems (DBMSes) serve as the main data repositories for organizations since they support wide-ranging functionality to efficiently manage data together with essential security features. Such native-DBMS security features are limited to authentication, access control, various levels of encryption, data masking, and audit logs. Unfortunately, the challenges of modern cybersecurity cannot be adequately addressed by only covering these native-DBMS security essentials. To compliment current DBMS security functionality, a byte-level, forensic analysis of DBMS storage would support a trusted view of a compromised DBMS server.

Most database forensics research has focused on relational DBMSes (e.g., SQLite, PostgreSQL, Oracle, MySQL). Some research exists for NoSQL DBMSes (e.g., MongoDB, Redis, Berkeley DB), but efforts often focus on an individual DBMSes. This presents a challenge for forensics research to address *all* NoSQL DBMSes, their different versions, and deployments on different operating systems.

This paper proposes ANOC which uses generalized methods to reverse-engineer and carve NoSQL DBMS storage at the byte-level, opening the door for the development of tools to provide trusted view of a compromised system. While this paper considers four representative

* Corresponding author.
E-mail addresses: minissan@uno.edu (M.I. Nissan), jwagner4@uno.edu (J. Wagner), arasin@cdm.depaul.edu (A. Rasin).

https://doi.org/10.1016/j.fsidi.2025.301929

key-value store DBMSes, our motivating philosophy is that a comprehensive reverse engineering tool should reconstruct *everything* from *all* databases. Everything includes the user data; system metadata, which provides context to user data; and unallocated storage, which can contain deleted data or previous copies of current user data. We propose a generalized approach for all NoSQL DBMSes - it is impractical to build a new, specialized tool as new DBMSes are introduced or when DBMSes change their storage architecture. We validate our approach through a series of experiments with four representative NoSQL databases. This is achieved by using synthetic data to learn a DBMS's storage architecture and the describing the metadata; once the storage architecture and metadata are known for a given DBMS, reconstructing the DBMS at the byte level is feasible. Our approach is database-agnostic because we observe the effect of standard user operations performed on the database. As our experiments demonstrate, there are a limited number of strategies to represent data in storage as a response to these operations. For example, tracking a new key-value pair in storage can be done based on sizes of the key/value, based on an offset of where a key/value is stored, or a combination of both. Databases are designed to localize this storage information. The primary contributions of this paper are.

- We propose generalized algorithms to learn and parameterize database storage layouts (Section 4).
- We leverage these parameters to implement generalized database carving (Section 5).
- We evaluate ANOC on four representative key-value store NoSQL databases (Berkeley DB, LMDB, ZODB, etcd) including corrupt storage and RAM snapshots (Section 6).

Beyond the four representative key-value store DBMSes evaluated in this paper, ANOC supports many other DBMSes. Some key-value store NoSQL DBMSes are forks or derivatives of each other. Examples we tested include, Durus and MDBX implement the same storage architecture as ZODB and LMDB, respectively. In addition to key-value store DBMSes ANOC supports the document store DBMSes RavenDB and LiteDB. Exploring the architectures across types of NoSQL DBMSes (e.g., key-value stores, document stores, graph databases) is beyond the scope of this paper.

2. Related work

Databases Forensics. Relational DBMSes store data in structured tables, organizing records into fixed-size pages, indexes, and directories (Hellerstein et al., 2007). These structured storage formats allow forensic tools to recover damaged or deleted records even without logs or system metadata (Wagner et al., 2017).

Forensic approaches are typically database-specific, with tools that are designed to extract records from a particular DBMS (Choi et al., 2021). Database forensic methods leverage structured page layouts, row directories, and metadata catalogs to extract records (Wagner et al., 2015, 2018). Unlike traditional file carving, which is used for recovering standalone files (Poisel and Tjoa, 2013), database forensic tools reconstruct data by interpreting database-specific storage structures (Frühwirt et al., 2012) from a synthetic dataset (Lenard et al., 2020). However, most traditional forensic tools remain database-specific, requiring analysts to develop separate methods for each relational DBMS.

To address this limitation, Wagner et al. proposed *universal reverse engineering* for relational DBMSes by parameterizing database page structures (Wagner et al., 2015, 2016; Wagner and Rasin, 2020). This approach was later implemented in *DBCarver* (Wagner et al., 2017, 2019, 2020a, 2020b), a tool designed to reconstruct relational storage structures at the byte level. *DBCarver* applies parameterization technique to successfully recover data from Oracle, PostgreSQL, IBM DB2, MySQL, Microsoft SQL Server, SQLite, Firebird, and Apache Derby.

NoSQL Forensics. NoSQL database forensics research is largely database-specific, with most studies focusing on individual systems rather than providing a generalized approach (Yoon et al., 2016; Chopade and Pachghare, 2019; Qi, 2014). Several forensic tools exist for specific NoSQL DBMSes, leveraging database-specific storage characteristics. Examples include MongoDB's (MongoDB and Inc, 2024) WiredTiger engine (MongoDB, 2023), which has led to forensic tools that analyze its B-tree-based storage, allowing for record reconstruction even after deletion (Yoon and Lee, 2018). Similarly, Redis (Redis and Inc, 2025) forensics has been studied through its in-memory persistence mechanisms, with tools like rdb-tools (Krishnan, 2020) and approaches such as LESS (Sung et al., 2019) designed to analyze and recover historical data from snapshot (RDB) storage formats. While these tools offer valuable insights into specific NoSQL databases, they do not provide a generalized forensic approach applicable across multiple databases.

NoSQL Forensic Challenges. Relational DBMSes store structured data with centralized metadata (Cheung et al., 2005) (e.g., system catalogs like MySQL's INFORMATION_SCHEMA (Oracle Corporation, 2025), PostgreSQL's pg_catalog (PostgreSQL Global Development Group, 2025)), whereas NoSQL systems use diverse storage models that vary in management of metadata and relationships (Candel et al., 2022). Some DBMSes, such as Berkeley DB (Olson et al., 1999), LMDB (Corporation, 2024), and etcd (Authors, 2025), use page-based storage with headers, incorporating some structural similarities to relational databases. However, they lack centralized system catalogs and embed metadata in pages (Berkeley DB, LMDB) or alongside key-value pairs (etcd). This distributed metadata complicates relationship reconstruction, as forensic tools must extract and interpret metadata dynamically rather than relying on predefined structures. Others, such as ZODB (Foundation, 2024), adopt serialized object storage, which deviates entirely from page-based structures and further complicates forensic recovery.

Generalized Approaches. Our approach builds on established file carving techniques, which reconstruct files without relying on system metadata or DBMS software (Garfinkel, 2007; Richard and Roussev, 2005). By drawing from research in relational database reverse engineering that examines byte-level reconstruction (Wagner et al., 2015, 2017), we propose Automated NoSQL Carver (ANOC), a forensic tool that parameterizes and carves NoSQL storage. Unlike relational forensic tools that depend on predefined page structures, ANOC dynamically learns NoSQL storage layouts, identifying key-value pairs, internal metadata, and hierarchical relationships from raw storage.

3. ANOC overview

ANOC reconstructs NoSQL DBMS data from disk images or RAM snapshots. It applies a parameterized carving approach, enabling recovery even when system metadata is missing or the database is corrupted. Sections 4 and 5 detail the parameter extraction and carving processes for Berkeley DB, LMDB, ZODB, and etcd, while Section 6 presents experimental evaluations. Sections 7 & 8 discuss limitations and future research directions.

Fig. 1 is an overview of ANOC. In Step 1, the Parameter Collector analyzes a target database to learn metadata. Synthetic data designed to force metadata changes is automatically loaded into the database (1.A). This known synthetic data allows ANOC to learn the database layout (1. B) and extract key metadata (1.C), which is described with a set of defined parameters (1.D). Initial forensic analysis of NoSQL databases required manual investigation to identify key parameters essential for data reconstruction. Using iterative analysis across multiple DBMSes, we identified a set of fundamental storage characteristics across NoSQL systems. As a result, the process became increasingly automated, requiring minimal manual intervention. To run the parameter collector, the user must start the target DBMS and specify the DBMS file directory.

In Step 2, the Carver uses the parameters to reconstruct system metadata and user data as a read-only operation from an image of a compromised system. The Carver operates as a command-line program, requiring two inputs: the directory containing the storage image files

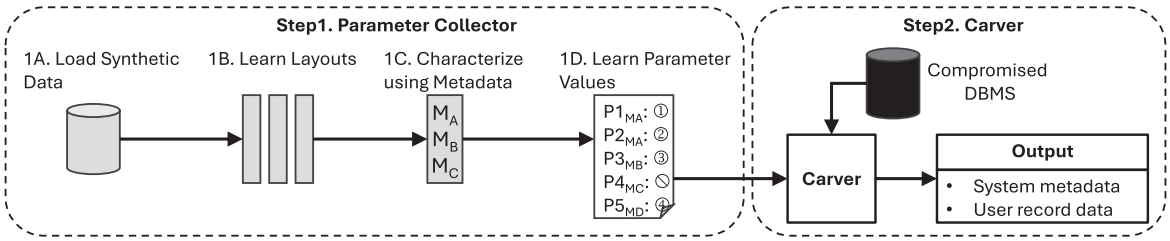

Fig. 1. ANOC architecture.

and an output directory. Once these inputs are provided, the Carver runs automatically, requiring no further user interaction. The extracted data is outputted as JSON files.

4. Parameter collector

The parameter collector examines each DBMS (and different versions) on a trusted system in a controlled environment. This section details how significant parameter values are automatically set; however, over 40 parameters (listed in (Nissan, 2025)) are not covered due to space constraints.

We divide parameter collection into three primary components: page header, record directory, and record data. The page header stores high-level information (e.g., B-Tree level) about the record data. The record directory maintains pointers to the record data. Finally, the record data contains the user data along with additional metadata. The following sections provide more detailed discussions on parameter collection for each one of these components.

4.1. Database storage structure

Many NoSQL systems depend on sequential or algorithmically generated storage layouts that omit record directories found in relational databases. While both NoSQL and relational databases may use structures like B+ trees, their application differs: in relational databases, B+ trees are tightly integrated with structured schemas and indexing, whereas in NoSQL databases, they often store key-value pairs without enforcing a fixed schema. Metadata in NoSQL systems is prone to be non-deterministic, relying on hierarchical pointers or dynamically generated maps such as key–value relationships or object references. These metadata structures are designed to work with database-specific algorithms and require custom interpretations, making traditional carving methods less directly applicable.

Fig. 2 illustrates the diverse storage structures of the four databases examined in this paper. Berkeley DB, LMDB, and etcd utilize pages, whereas ZODB does not. Although page concepts such as page type, (e.g., whether it is a B+ tree leaf node containing records) and page IDs are consistently used across A, B, and C, the metadata and data organization structures vary significantly.

4.2. Page layout

Fig. 3 illustrates the page header layouts for each database. Among these, Berkeley DB, LMDB, and etcd utilize pages, while ZODB does not. A common feature across the paged databases is the 2-byte page type (e.g., leaf node containing records) and the page ID, stored as a 32-bit or 64-bit integer.

Additionally, Berkeley DB uses 2 bytes to store the key count, and it includes a record directory with pointers to records in the page. LMDB also has a record directory but no key count. etcd stores the key count using 2 bytes but does not use a record directory. Finally, ZODB, which does not use pages, contains only metadata embedded directly in the record data stream. Unlike the metadata in Berkeley DB, LMDB, and etcd, the metadata in ZODB is not designed to identify record directories, key counts, or other details typically used for record retrieval. Instead, it is tailored to manage hierarchical object relationships.

Fig. 4 illustrates two example page information layouts. In Berkeley DB, the page ID, page type, and key count are at offsets 8, 24, and 20, respectively. In etcd, the page ID, page type, and key count are at offsets 0, 8, and 10, respectively. Berkeley DB also provides metadata for the previous page ID and the next page ID at offsets 12 and 16, respectively. In terms of storage, Berkeley DB and etcd represent the page ID with 4 and 8 bytes, respectively. Both examples allocate 2 bytes for the key count.

The parameter collector automatically determines the offset and byte usage for each metadata item in Fig. 4. Thus, each metadata has a) an offset parameter and b) a size parameter. Table 1 lists example parameters we defined and the values automatically returned by ANOC. Page ID parameters are determined by comparing constant offsets across pages for a sequential number. The page size is identified by analyzing the positions of page IDs and the order of records in a page. Databases without pages use a contiguous layout of metadata and user data.

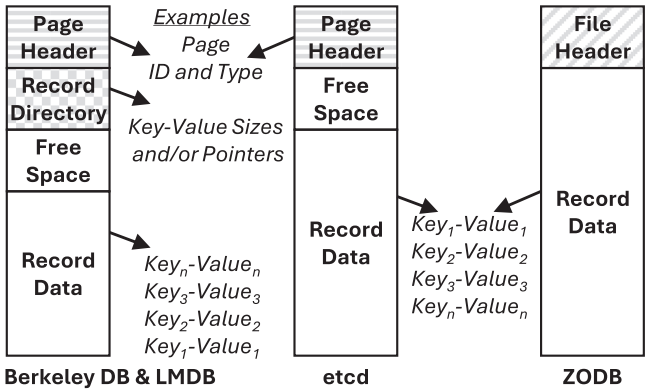

Fig. 2. High-Level Overview of Database Storage for Berkeley DB, LMDB, etcd, & ZODB.

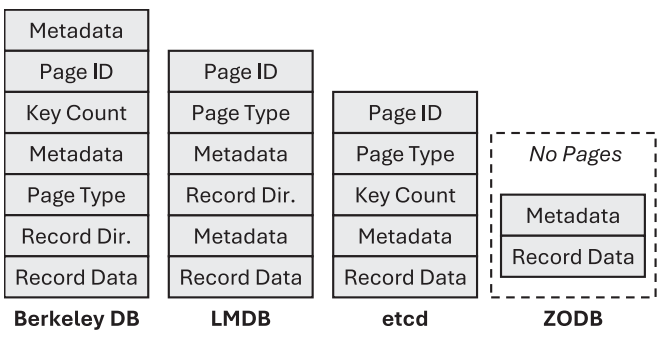

Fig. 3. Storage Structures for Berkeley DB, LMDB, etcd, & ZODB.

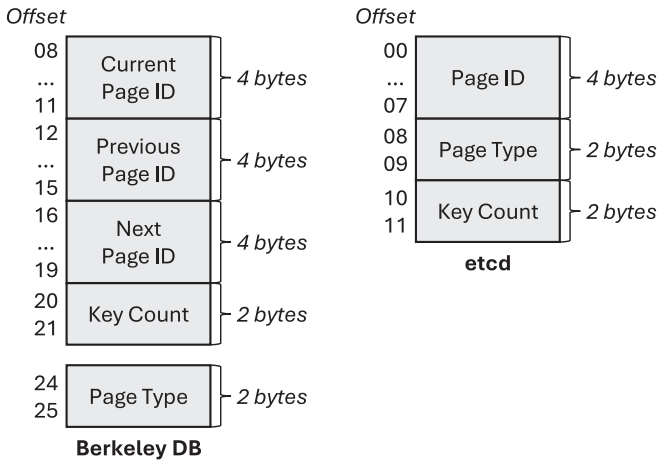

Fig. 4. Page header layouts for Berkeley DB & etcd.

Table 1
Example page parameters used to reconstruct Fig. 2.

Parameter	Berkeley	LMDB	etcd	ZODB
General ID	(1, 5)	(2, 0)	(2, 0)	(46, 53)
General ID Offset	24	10	08	00
Page Size	4 KB	4 KB	4 KB	N/A
Unique Page ID Offset	08	00	00	N/A
Previous Page ID Offset	12	N/A	N/A	N/A
Next Page ID Offset	16	N/A	N/A	N/A
Page ID Size	4 bytes	8 bytes	8 bytes	N/A
Key Count Offset	20	N/A	10	N/A
Key Count Size	2 bytes	N/A	2 bytes	N/A

4.3. Record directory

The record directory stores pointers that reference records (or each key and each value) in a page. Berkeley DB and LMDB use a record directory, while etcd and ZODB do not. Both Berkeley and LMDB store the record directory after the page ID and key count.

Fig. 5 shows examples of record directories for Berkeley DB and LMDB. Both examples store 2-byte little-endian addresses, and the first directory address is consistent across all pages for a single DBMS. In Berkeley DB, the first key and first value addresses are stored at offsets 26 and 28, respectively. Subsequent key addresses are calculated using formula $\underline{26} + (n \times 4) - 4$, while subsequent value addresses follow $28 + (n \times 4) - 4$. In contrast, LMDB stores only value addresses, with the first value address at offset 16 and subsequent addresses calculated as $16 + (n \times 2) - 2$. In both examples, addresses are appended from Top-to-Bottom in the directory.

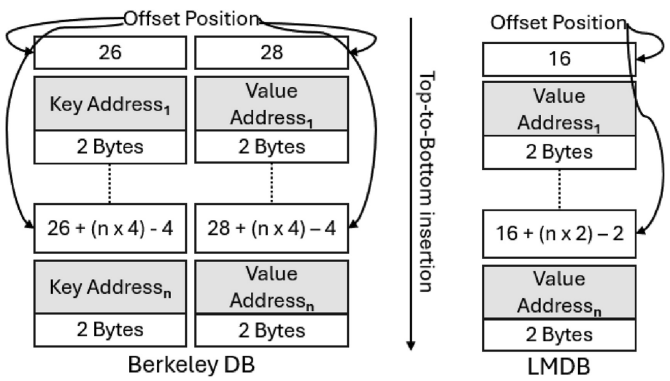

Fig. 5. Record directories for Berkeley DB & LMDB.

The record directory parameters, such as the offset and address size, were automated through pointer chasing (Anders et al., 2016). Table 2 lists a sample set of parameters we defined and the values automatically returned by ANOC for Fig. 5.

4.4. Record data

Fig. 6 illustrates four example record layout implementations in Berkeley DB, etcd, LMDB, and ZODB. Berkeley DB record layout shows the sizes for the key and value, which is what the record directory points to. A constant delimiter is present 1) between the key size and key and 2) between the value size and value. Records in this layout are inserted using a Bottom-to-Top approach. etcd layout does not use a directory for storing offsets. Instead, records are appended Top-to-Bottom, with no metadata separating the key size and key, or the value size and value. LMDB uses a record directory that points to the value sizes but does not have addresses to the keys or key sizes. In this layout, records are organized sequentially as value size, metadata, key size, metadata, followed by the key and value, with no delimiter or metadata between the key and value. It also appends records in a Bottom-to-Top order. ZODB stores records sequentially as key size, metadata, then the key, metadata, value size, metadata, and finally the value. This layout appends records Top-to-Bottom.

Table 3 lists record data parameters collected by ANOC. All four databases in Fig. 6 store the key and value sizes as 8-bit or 16-bit integers for strings between 1 and 2^8 bytes or 2^8–2^{16} bytes, respectively. For example, in Table 3, for Berkeley DB, the parameter collector saves the Key Delimiter (i.e., the constant byte sequence between the key size and the key) as **(0, 1)**. From the delimiter, the Key Size and the Key are at offsets −1 and 1, respectively.

5. Carver

The Carver operates as read-only on the user-specified files, disk images, or RAM snapshots. The parameter file(s) provides the details needed to reconstruct the database. The reconstructed output is returned as a JSON file containing database metadata and user data.

Table 4 summarizes the functionality currently supported by the Carver. The table outlines core features, i.e., page detection, page parsing, record directory, and string decoding, specifying whether each feature is available (✓) or not applicable (N/A) for each database. Future work (Section 8) considers additional features.

The Carver first reads the parameters listed in Table 1. When it detects a general ID marker associated with a page, it reconstructs only the metadata and data as it is described by the parameters. This includes the page size, page ID, record directory, record count, and record data. If certain components are missing or not applicable to a specific database, they are not included in the JSON output. However, because general ID markers are typically short sequences of bytes, the Carver may identify false matches. To eliminate false matches, it applies a series of assertions to validate the reconstructed components. Assertions include: page header IDs must be greater than 0, the record directory (when present) contains at least one address, the record data includes at least one record, a record directory address is within the bounds of a page, and the key count matches the number of records in a page.

Table 2
Record directory parameters used to reconstruct Fig. 5.

Parameter	Description	Berkeley	LMDB
Key Offset	1st key address offset	26	N/A
Value Offset	1st value address offset	28	16
Little Endian	Addresses are stored using little-endian.	True	True
Top-to- Bottom	Addresses appended in ascending order.	True	True
Address Size	# bytes allocated for each address.	2 bytes	2 bytes

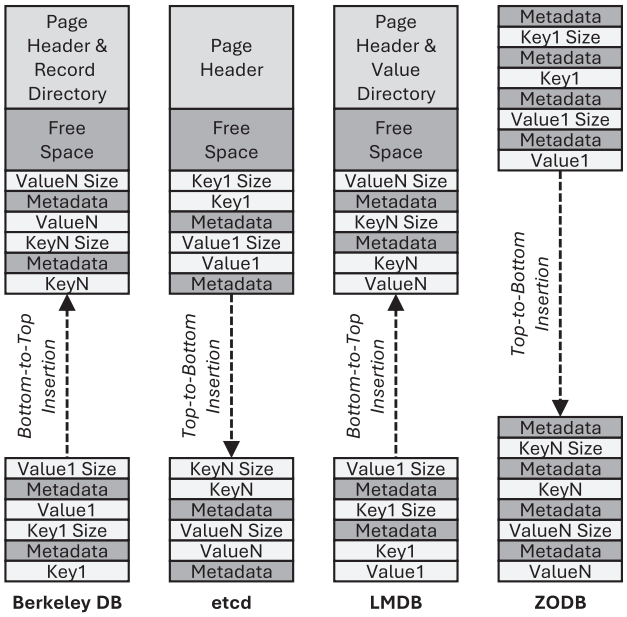

Fig. 6. Record layouts for Berkeley DB, etcd, LMDB, & ZODB.

Table 3
Record data parameters used to reconstruct Fig. 6.

Parameter		Berkeley	etcd	LMDB	ZODB
Key	Size	True	True	True	True
	Size Offset	−1	N/A	1	1
	Delimiter	(0, 1)	N/A	(0, 0, 0)	(0, 0, 0)
	Offset	1	N/A	2	−1
Value	Size	True	True	True	True
	Size Offset	−1	N/A	1	1
	Delimiter	(0, 1)	N/A	(0, 0, 0)	(0, 0, 0)
	Offset	1	N/A	2	−1

Table 4
Carving functionality breakdown across databases.

Function	Berkeley	ZODB	etcd	LMDB
Pages	✓	N/A	✓	✓
Record Dir.	✓	N/A	N/A	✓
Strings	✓	✓	✓	✓

5.1. Page header

The Carver uses the parameter file values from Table 1 to reconstruct the page header metadata from Fig. 4. In Berkeley DB, the Carver moves to offset 24 to reads 2 bytes, identifying whether the page was a leaf node. It then moves to offsets 9, 12, and 16 to read 4 bytes each and reconstruct the unique page ID, the previous page ID, and the next page ID, respectively. At offset 20, the Carver reads 2 bytes to determine the record count as a 16-bit integer. In etcd, the Carver follows a similar process but noted that this example does not include previous or next page IDs. To reconstruct the key count, the Carver moves to offset 10, reads 2 bytes as a 16-bit integer. All values were interpreted using little-endian format.

5.2. Record directory

Table 2 parameter values were used to reconstruct the record directory for the two examples in Fig. 5. Table 5 outlines how the record directory addresses were reconstructed. The Carver begins by using the Key Offset or Value Offset parameter to move to the first address in the record directory. Each address is derived using the formulas $\text{KeyAddress}_n = K_n + (n \times 4) - 4$ and $\text{ValueAddress}_n = V_n + (n \times 4) - 4$. If only key values were present, the formula $\text{KeyAddress}_n = K_n + (n \times 2) - 2$ was used, where K_n and V_n represent decoding constants provided as parameters. Once the bytes for each address are collected, they are interpreted as 16-bit integers. After the first address is reconstructed, the Carver calculates subsequent addresses based on the Address Size and Top-to-Bottom Insertion parameters.

Table 5
Record directory address reconstructed from Fig. 5.

Address	Fig. 5 **Value**		
	Berkeley DB		**LMDB**
	Key	**Value**	**Value**
Address_1	26 27	28 29	16 17
Address_2	30 31	32 33	18 19
Address_3	34 35	36 37	20 21
Address_n	526 527	528 529	240 241

5.3. Record data

Table 3 parameter values were used to reconstruct the record data in the four examples from Fig. 6. The delimiters are used to locate key/value lengths, which are then used to extract the keys/values. If there is a record directory (e.g., Berkeley DB and LMDB in Fig. 6), the Carver moves to a record using this respective address.

For example, in Berkeley DB, the database contains a record directory with addresses for the keys and values. The Carver uses these addresses to move to each record. Then, the key (or value) size offsets from the delimiter parameter, **(0, 1)**, allow the Carver to extract the key (or value) length. Finally, the Carver uses the key (or value) offsets from the delimiter to extract the user data. If the database does not use a record directory (e.g., etcd and ZODB in Fig. 6), the Carver reads the data sequentially.

6. Experiments

ANOC has been tested on Berkeley DB, ZODB, etcd, and LMDB—on both Windows and Linux operating systems. Table 6 summarizes the tested DBMS version, operating system, and their respective page sizes. To test ANOC on different systems and configurations, experiments on Linux OS (Ubuntu 20.04 LTS) were conducted using VirtualBox with an AMD 7950X3D @ 4.2 GHz processor. The VM was configured with 8 GB of RAM and 4 processors. For Windows OS (Windows 11), experiments were performed on a system with an Intel Core i5-1135G7 @ 2.40 GHz, 8 processors, and 16 GB of RAM.

Dataset. We used the SSBM (Neil et al., 2009) benchmark to generate data at different scales for experiments. Since this paper focuses on key-value pair databases, we selected two representative columns from each table, as shown in Table 7. To ensure consistency across different databases, all data, including integers and dates, were stored as strings. This approach was adopted because, unlike relational databases, NoSQL databases typically store data at the byte level and often lack native

Table 6
DBMS version, testing operating system, & page size.

DBMS version	Testing OS	Page Size (KB)
Berkeley 18.1.40	Linux	4
Berkeley 18.1.40	Windows	4
ZODB 6.0	Linux	N/A
ZODB 6.0	Windows	N/A
etcd 3.2.26	Linux	4
etcd 3.2.26	Windows	4
LMDB 0.9.24	Linux	4
LMDB 0.9.24	Windows	4

Table 7
Dataset.

Table	Key	Value
Customer	name	nation
Supplier	name	city
Part	partkey	name
Date	datekey	date
Lineorder	orderkey	shipmode

support for data types.

Columns such as `Date.datekey`, `Part.partkey`, and `Lineorder.orderkey`, originally integers, were converted to strings with left padding to maintain consistent formatting. Since NoSQL DBMSes do not use the concept of tables, we prefixed the table name to the keys to distinguish data. Keys were constructed as `"Date#" + "datekey"`, `"Lineorder#" + "orderkey"`, and `"Part#" + "partkey"`, resulting in `"Date#00000001"`, `"Part#00000001"`, and `"Lineorder#00000001"`. Keys such as `Customer.name` and `Supplier.name` already include information indicating their source tables.

6.1. Accuracy

This experiment tests ANOC's accuracy across operating systems for Berkeley DB, ZODB, etcd, and LMDB. 15 million records were inserted into each DBMS. Table 8 summarizes the results.

We observed 269,269 repeated records in Berkeley DB and 269,099 repeated records in LMDB. These repetitions occur because the file layout in these databases incorporates additional metadata and redundant records to facilitate efficient access within their storage model.

For Berkeley DB, ANOC reconstructed data by using the record directory, which explicitly points to individual records in a page ensuring no duplication in reconstruction. In contrast, for LMDB, ANOC reconstructed 134,959 repeated records. Although ANOC used the record directory to reconstruct records, repeated records were included because the record directory in LMDB sometimes references entries from previous pages. The discrepancy between the number of repeated records in the LMDB database file (269,099) and those reconstructed by the Carver (134,959) is due to LMDB's use of a B-tree index. Some records are found in the B-tree intermediate nodes.

ZODB organizes data sequentially in files without using a record directory. Consequently, there were no repeated records in its database files. The Carver successfully reconstructed all 15 million records from ZODB without duplication, achieving 100 % accuracy.

For etcd, although 15 million records were inserted, the Carver extracted 14,855,388 unique records, achieving 99 % accuracy. The slight discrepancy stems from etcd's storage optimization, which dynamically reconstructs elements like index mappings and transaction logs at runtime rather than persisting them explicitly. Unlike traditional databases, etcd maintains a lean storage model where some key–value relationships exist transiently in memory and materialize on disk only when necessary. As a result, certain records lack a resolvable on-disk footprint, making full recovery inherently challenging. Additionally, key reassignments during compaction and defragmentation further contribute to minor losses in the extraction process.

Table 8
Accuracy.

Database	Records Inserted (Unique)	Records in DB File	Carved Record	Accuracy
Berkeley	15M	15269269 (with repetition)	15M (Unique)	100 %
ZODB	15M	15M	15M (Unique)	100 %
etcd	15M	15M	14855388 (Unique)	99 %
LMDB	15M	15269099 (with repetition)	15134959 (with repetition)	100 %

6.2. Performance

This experiment measures the runtime performance of the ANOC Carver and its scalability for Berkeley DB, ZODB, etcd, and LMDB, using datasets ranging from 1 million to 15 million records. We attempted to extend the experiment beyond 15 million records; however, ZODB's use of the Pickle serialization format and its single-file architecture limited our ability to insert additional records. Pickle's sequential file handling and lack of concurrent writes caused a bulk loading error that our data size (~20M records) was beyond Pickle's single file capabilities, while ZODB's single-file architecture inherently restricts the management and scalability of larger datasets. The performance metrics were assessed by measuring the carving speed (MB/s) and total runtime for each dataset.

Fig. 7 compares the carving performance of BerkeleyDB, ZODB, etcd, and LMDB, while Tables 9–12 summarize the individual throughput performance metrics. BerkeleyDB had the fastest carving speeds, followed by LMDB, ZODB and etcd. For example, our dataset of 5 million records was carved at a speed of 30.94 MB/s in 6.66 s for Berkeley DB and at 29.15 MB/s in 6.32 s for LMDB. Additionally, the largest dataset of 15 million records achieved a carving speed of 37.57 MB/s with a runtime of 21.46 s for Berkeley DB and 28.29 MB/s with a runtime of 19.47 s for LMDB.

This efficiency stems from the record directory storage architectures of Berkeley DB and LMDB. In Berkeley DB, each page includes a record directory in the page header that provides direct pointers to record locations, enabling the Carver to locate and extract records more quickly. Although LMDB also uses a record directory, it only contains the offset positions of value lengths. As a result, the Carver must calculate the key positions based on these offsets, introducing additional computational overhead and leading to slightly slower performance compared to Berkeley DB.

In contrast, ZODB and etcd demonstrated slower carving speeds and longer runtimes compared to Berkeley DB and LMDB. For the dataset containing 5M records, ZODB achieved a carving speed of 18.91 MB/s with a total runtime of 11.60 s, whereas etcd reached a speed of 18.16 MB/s with a runtime of 21.82 s. As for the largest dataset containing 15M records, ZODB carved at 18.61 MB/s, taking 35.88 s, while etcd carved at 18.89 MB/s, taking 63.28 s.

This slower performance is attributed to the absence of a record directory in ZODB and etcd (page header). Unlike Berkeley DB and LMDB, which store pointers or offsets to records within a page, ZODB and etcd require the Carver to traverse and process each record individually. This architectural limitation increases computational overhead, resulting in slower carving speeds and longer processing times compared to Berkeley DB and LMDB.

6.3. Corruption

The objective of this experiment is to evaluate the ability of the Carver to reconstruct database content from corrupted files. Using the approach outlined in Section 6, we loaded customer table with sizes of 5M, 10M, and 15M into Berkeley DB and ZODB databases, and then created a persistent storage image. Corruption with varying levels of damage (1 %, 2 %, 5 %, and 10 %) was simulated by overwriting random 1 KB segments of the file, where each level was applied independently. Finally, the damaged images were processed using ANOC. The results are summarized in Tables 13 and 14, which highlight the percentage of records successfully reconstructed at each level.

6.3.1. Berkeley DB

For Berkeley DB, the reconstruction percentage decreased as file damage increased, as summarized in Table 13. At 0 % corruption, the reconstruction rate achieved was 100 % across all dataset sizes, with 5M, 10M, and 15M records fully recovered. However, as damage levels

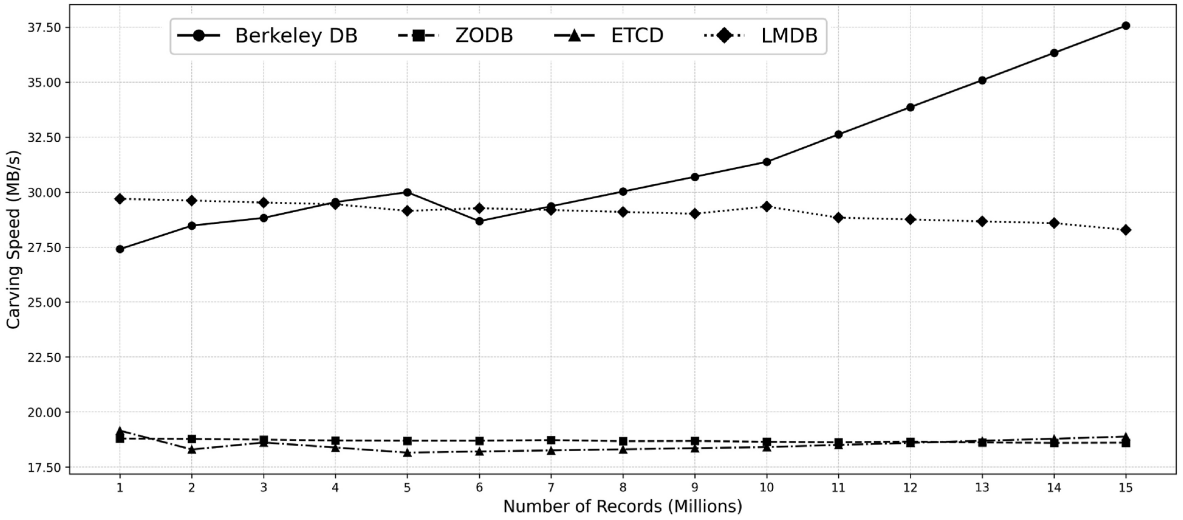

Fig. 7. Carving performance comparison.

Table 9
ANOC Carver speeds on BerkeleyDB.

No. of Records	DB File (MB)	Pages	Time (MB/s)	Total Time (s)
5M	184	44643	30.94	6.66
10M	369	89286	31.38	13.14
15M	553	133929	37.57	21.46

Table 10
ANOC Carver speeds on LMDB.

No. of Records	DB File (MB)	Pages	Time (MB/s)	Total **Time** (s)
5M	184	44650	29.15	6.32
10M	368	89294	29.35	12.55
15M	552	133938	28.29	19.47

increased, the recovery rates showed reductions.

For the 5M dataset, 96.5 % of records were reconstructed at 1 % corruption, 91.3 % at 5 %, and 85.1 % at 10 % corruption. Similarly, the 10M dataset showed 94.3 % recovery at 1 % corruption and 76 % at 10 % corruption. The 15M dataset had a sharper decline, with recovery dropping to 91.8 % at 1 % corruption and 71.6 % at 10 % corruption.

The reduction in record carving rates is due to Berkeley DB's hierarchical storage model, which relies on metadata structures such as headers and row directories to organize and access data. When these structures are damaged, our Carver is unable to recover the affected pages.

Our experiments showed that each Berkeley DB page stored ~126 records. Corruption in the middle of a page can disrupt parsing by damaging the key-value structure, making it impossible to extract the affected record if its key, value, or delimiters are lost. However, recovery may still be possible even when the record directory is damaged. Similar to our approach with etcd and ZODB, valid key-value structures can be identified sequentially beyond the corrupted section. However, this method has not yet been implemented by ANOC for Berkeley DB.

6.3.2. ZODB

In contrast, ZODB exhibited a linear decline in reconstruction rates. At 0 % corruption, all records were fully reconstructed for datasets of sizes 5M, 10M, and 15M. With 1 % corruption, reconstruction rates slightly decreased to 99 % across all dataset sizes. At 10 % corruption, recovery rates were still relatively high, with 90 % for the 10M dataset and 89.8 % for the 15M dataset.

This linear decline in recovery rates is attributed to ZODB's sequential storage model, which organizes key-value pairs without a row directory. Thus, corruption in one part of storage has a localized effect, leaving other areas unaffected and recoverable. ANOC scans through to the end of the file, as ZODB does not store an explicit termination marker where key-value sequences finally end. If corruption disrupts the expected key-value layout, ANOC continues scanning for the next valid structure, allowing recovery to resume beyond the corrupted section.

Table 11
ANOC Carver speeds on etcd.

No. of Records	DB File (MB)	Pages	Time (MB/s)	Total **Time** (s)
5M	416	101327	18.16	21.82
10M	835	203480	18.41	43.24
15M	1331	305659	18.89	63.28

6.4. Memory snapshots

The experiments in this section are designed to demonstrate ANOC's

Table 12
ANOC Carver speeds on ZODB.

No. of Records	DB File (MB)	Time (MB/s)	Total **Time** (s)
5M	230	18.91	11.60
10M	460	18.80	23.33
15M	690	18.61	35.88

Table 13
Data reconstructed from a corrupted file of Berkeley DB.

Records Inserted	File Percent Damage				
	0 %	**1 %**	**2 %**	**5 %**	**10 %**
5M	5M (100 %)	4.8M (96.5 %)	4.6M (93.8 %)	4.5M (91.3 %)	4.2M (85.1 %)
10M	10M (100 %)	9.4M (94.3 %)	9.1M (91.2 %)	8M (80.1 %)	7.6M (76.0 %)
15M	15M (100 %)	13.7M (91.8 %)	12.8M (85.1 %)	12M (80.3 %)	10.7M (71.6 %)

Table 14
Data reconstructed from a corrupted file of ZODB.

Records Inserted	File Percent Damage				
	0 %	**1 %**	**2 %**	**5 %**	**10 %**
5M	5M (100 %)	4.95M (99.0 %)	4.9M (98.0 %)	47.5M (95.0 %)	4.5M (90.0 %)
10M	10M (100 %)	9.9M (99.0 %)	9.8M (98.0 %)	9.5M (95.0 %)	9.0M (90.0 %)
15M	15M (100 %)	14.8M (98.9 %)	14.6M (97.9 %)	14.2M (94.9 %)	13.4M (89.8 %)

ability to carve RAM snapshots. We do this with designed query workloads that demonstrate the cache eviction policies for Berkeley DB and LMDB.

6.4.1. Berkeley DB

6.4.1.1. Procedure. We set the Berkeley DB cache size to 10 MB and queried all five tables in the following sequence: *customer*, *supplier*, *part*, *date*, and *lineorder*. Each query selected 200,000 key-value pairs from the respective table.

To distinguish between LRU and FIFO eviction policies, we re-accessed 10,000 key-value pairs from the previously queried table before executing the next table. For instance, before executing the query on the *supplier* table, we re-accessed 10,000 keys from the *customer* table to make those pages the most recently accessed. This procedure ensured that the retention of re-accessed pages could be observed, which is characteristic of an LRU policy, as opposed to FIFO, which would evict the oldest read pages regardless of recent access.

Each query was designed to fill ~6.6 MB of the 10 MB cache, ensuring conditions that trigger page evictions. After executing each query, a process snapshot was captured, resulting in a total of five snapshots. ANOC then reconstructed the database contents from each snapshot to analyze which data was retained or evicted.

6.4.1.2. Results. ANOC reconstructed the database pages from the snapshots at an average carving speed of 38 MB/s, and Fig. 8 illustrates that Berkeley DB uses a Least Recently Used (LRU) eviction policy to manage its cache.

Across the five snapshots, some pages from the re-accessed 10,000 key-value pairs of the previously queried table were consistently retained in the cache, though not all were present due to space constraints and the eviction of older pages. In Snapshot 1, only the *customer* table's pages were present in the cache. In Snapshot 2, both *supplier* and *customer* pages were observed, with a portion of the re-accessed *customer* pages retained alongside the newly queried *supplier* pages. By Snapshot 3, no *customer* pages remained, while some re-accessed *supplier* pages were retained alongside the *part* table pages.

In Snapshot 4, most *part* pages were replaced by *date* pages, although some re-accessed *part* pages were still present. Finally, in Snapshot 5, the cache predominantly contained *lineorder* table pages, with a subset of re-accessed *date* pages retained. These patterns demonstrate that the most recently accessed pages are prioritized for retention, aligning with LRU behavior. This behavior is not consistent with FIFO (First In, First Out), which would have evicted the re-accessed pages regardless of their recent access, as they were inserted earlier. The presence of re-accessed pages across snapshots confirms that Berkeley DB uses LRU rather than FIFO for cache management.

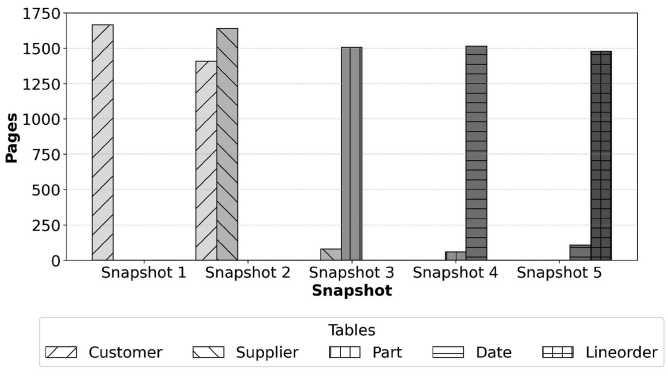

Fig. 8. Berkeley DB recovered cache pages.

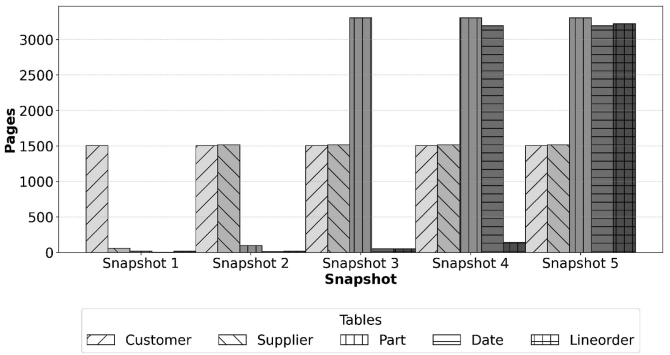

Fig. 9. LMDB recovered cache pages.

Additionally, we observed that although the cache size was set to 10 MB, Berkeley DB utilized approximately 12.5 MB of memory during query execution. This additional memory usage is attributed to overheads associated with managing B-tree structures, including internal metadata, alignment inefficiencies during page allocation, and temporary buffers used for cursor operations.

6.4.2. Lightning memory-mapped database (LMDB)

6.4.2.1. Procedure. We set the data file size for LMDB to 50 MB, corresponding to 200,000 records from each table: *Customer*, *Supplier*, *Part*, *Date*, and *Lineorder*. This workload was chosen to align with the configuration of the Berkeley DB experiment, ensuring a fair comparison of caching mechanisms between the two systems. Unlike Berkeley DB, which implements its own caching mechanism, LMDB relies on the OS to manage caching and page eviction. To maintain consistency, the query workload for LMDB was also designed to match the Berkeley DB experiment.

6.4.2.2. Results. After executing the query to select 200,000 records, we observed that all pages of the queried table were present in the memory snapshot. ANOC reconstructed database pages at 30 MB/s. Fig. 9 summarizes the results. In Snapshot 1, all 1505 pages from the *customer* table were cached, along with a few pages from other tables. In Snapshot 2, all 1516 pages from the *supplier* table were cached, with no eviction of the *customer* pages. By Snapshot 3, the cache included all 3305 pages from the *part* table, along with pages from previously queried tables. In Snapshot 4, all 3191 pages from the *date* table were loaded, while pages from earlier tables were also retained. Finally, in Snapshot 5, the OS loaded all 3218 pages from the *lineorder* table and the entire database file into memory, retaining all pages from every table.

The higher number of pages in *part*, *date*, and *lineorder* tables, compared to *customer* and *supplier*, is attributed to LMDB's B-tree structure and management of key-value pairs. More frequent B-tree reorganization and creation of intermediate nodes results in higher page counts.

We also observed that the OS retained pages in memory even after database shutdown. This reflects the OS's caching mechanism, where recently accessed pages are kept in memory to optimize future access. In a separate experiment with a larger dataset (configured 2 GB file size for 15 million records), the OS frequently evicted the least recently used

pages due to increased memory demand. This behavior aligns with standard OS memory management strategies, where page retention depends on memory availability and access patterns. When memory demand increases, the OS prioritizes active processes and data by evicting less frequently accessed pages.

7. Limitations

The current implementation of ANOC has three known limitations: databases that use compression, encryption, or Log-Structured Merge-trees (LSM).

7.1. Compression & encryption

Some NoSQL databases, such as MongoDB (WiredTiger), Badger, and RocksDB, compress the keys and values by default. The current version of the Parameter Collector (Section 4) relies on searching storage for the known plaintext synthetic data. Therefore, the current version of ANOC does not support such databases. To add support for these databases, the compression algorithms used by each database would need to be provided to ANOC.

In addition to compression, the Carver (Section 5) does not reconstruct encrypted data. If individual keys or values are encrypted, ANOC could carve and return the ciphertext using the metadata. However, ANOC would not be able to return any information if the entire file is encrypted.

7.2. LSM-based architectures

Databases, such as LevelDB and RocksDB, use LSM-tree structures, which write data into immutable, sequential SSTables. LSM-trees do not have stable page boundaries and use compaction to rearrange data, which breaks ANOC's page-based carving model. The frequent layout changes, fragmentation, and multi-level storage hierarchy obscure the necessary patterns and metadata needed for carving. Supporting these structures would require implementing LSM-specific libraries to decode SSTables and interpret compaction logic.

8. Future work

Future work will enhance ANOC's capability in several key areas. Our focus will be incorporating unallocated storage areas to detect residual information and reconstruct deleted records. This enhancement will allow investigators to recover deleted data and validate modifications by comparing reconstructed records with their original versions. We will refine the parameter collection process to support various indexing mechanisms and serialization formats, expanding compatibility with a broader range of NoSQL databases. Additionally, our work will utilize the output from the Carver for security auditing. By analyzing in-memory data, we will identify unauthorized modifications, detect inconsistencies between memory and disk storage, and uncover potential tampering attempts. Furthermore, while ANOC currently processes string-based data, we will extend its capabilities to recognize and extract integers, timestamps, and other data types, making it more adaptable across different data formats.

9. Conclusion

We presented Automated NoSQL Carver (ANOC) to address the forensic challenges posed by diverse NoSQL storage architectures. ANOC reconstructs database contents from raw storage using byte-level analysis, without relying on system metadata or database APIs. It identifies key-value structures, hierarchical relationships, and metadata across Berkeley DB, ZODB, etcd, and LMDB. Our experiments demonstrate ANOC's effectiveness in recovering structured data from corrupted storage and RAM snapshots, validating its applicability in forensic investigations.

As NoSQL databases continue to evolve, ANOC provides a generalized framework for forensic analysis, offering a foundation for future enhancements in database recovery, integrity verification, and broader NoSQL support.

Acknowledgments

This work was partially funded by the Louisiana Board of Regents Grant LEQSF(2022-25)-RD-A-30 and by US National Science Foundation Grant IIP-2016548.

References

Anders, M.A., Kaul, H., Chen, G.K., 2016. Pointer Chasing across Distributed Memory. https://patentimages.storage.googleapis.com/38/e2/20/2280c48467bffe/US20160179670A1.pdf.
Authors, E., 2025. Etcd: Distributed Reliable Key-Value Store, 2025-01-29. https://github.com/etcd-io/etcd.
Candel, C.J.F., Ruiz, D.S., García-Molina, J.J., 2022. A unified metamodel for nosql and relational databases. Inf. Syst. 104, 101898.
Cheung, S.Y., Lu, J.J., Wyss, C.M., 2005. Metadata management and relational databases. In: Proceedings of the 43rd Annual Southeast Regional Conference, vol. 1, pp. 227–232.
Choi, H., Lee, S., Jeong, D., 2021. Forensic recovery of sql server database: Practical approach. IEEE Access 9, 14564–14575.
Chopade, R., Pachghare, V.K., 2019. Ten years of critical review on database forensics research. Digit. Invest. 29, 180–197.
Corporation, S., 2024. Lightning Memory-Mapped Database (Lmdb), 2025-01-29. https://github.com/LMDB.
Foundation, Z., 2024. Zodb, 2025-01-29. https://zodb.org/en/latest.
Frühwirt, P., Kieseberg, P., Schrittwieser, S., Huber, M., Weippl, E., 2012. Innodb database forensics: reconstructing data manipulation queries from redo logs. In: 2012 Seventh International Conference on Availability, Reliability and Security. IEEE, pp. 625–633.
Garfinkel, S.L., 2007. Carving contiguous and fragmented files with fast object validation. Digit. Invest. 4, 2–12.
Hellerstein, J.M., Stonebraker, M., Hamilton, J., et al., 2007. Architecture of a database system. Foundat. Trends Databases 1 (2), 141–259.
Krishnan, S., 2020. Rdbtools. https://github.com/sripathikrishnan/redis-rdb-tools, 2025-01-29.
Lenard, B., Wagner, J., Rasin, A., Grier, J., 2020. Sysgen: system state corpus generator. In: Proceedings of the 15th International Conference on Availability, Reliability and Security, pp. 1–6.
MongoDB, I., 2023. Wiredtiger, 2025-01-29. https://source.wiredtiger.com/11.0.0/arch-btree.html.
MongoDB, Inc., 2024. Mongodb, 2025-01-29. https://www.mongodb.com.
Neil, P.O., et al., 2009. The star schema benchmark and augmented fact table indexing. In: Performance Evaluation and Benchmarking.
M. I. Nissan, Analysis of Forensic Artifacts in Database Memory Using Support Vector Machine.
Nissan, M.I., 2025. ANOC Parameters. Zenodo. https://doi.org/10.5281/zenodo.15383693.
Nissan, M.I., Wagner, J., Aktar, S., 2023. Database memory forensics: a machine learning approach to reverse-engineer query activity. Forensic Sci. Int.: Digit. Invest. 44, 301503.
Olson, M.A., Bostic, K., Seltzer, M.I., Berkeley, D.B., 1999. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. https://dl.acm.org/doi/10.5555/1268708.1268751.
Oracle Corporation, 2025. Mysql 8.4 Reference Manual: Information_schema Tables, 2025-01-29. https://dev.mysql.com/doc/refman/8.4/en/information-schema.html.
Poisel, R., Tjoa, S., 2013. A comprehensive literature review of file carving. In: 2013 International Conference on Availability, Reliability and Security. IEEE, pp. 475–484.
PostgreSQL Global Development Group, 2025. Postgresql Documentation: Schemas, 2025-01-29. https://www.postgresql.org/docs/current/ddl-schemas.html.
Qi, M., 2014. Digital forensics and nosql databases. In: 2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD). IEEE, pp. 734–739.
Redis, Inc., 2025. Redis, 2025-01-29. https://redis.io/.
Richard III, G.G., Roussev, V., 2005. Scalpel: a frugal, high performance file carver. In: DFRWS. Citeseer.
Sung, H., Jin, M., Shin, M., Roh, H., Choi, W., Park, S., 2019. Less: Logging exploiting snapshot. In: 2019 IEEE International Conference on Big Data and Smart Computing (BigComp). IEEE, pp. 1–4.
Wagner, J., Rasin, A., 2020. A framework to reverse engineer database memory by abstracting memory areas. In: International Conference on Database and Expert Systems Applications. Springer, pp. 304–319.
Wagner, J., Rasin, A., Grier, J., 2015. Database forensic analysis through internal structure carving. Digit. Invest. 14, S106–S115.
Wagner, J., Rasin, A., Grier, J., 2016. Database image content explorer: carving data that does not officially exist. Digit. Invest. 18, S97–S107.

Wagner, J., et al., 2017. Database forensic analysis with dbcarver. In: Conference on Innovative Data Systems Research.
Wagner, J., Rasin, A., Heart, K., Malik, T., Furst, J., Grier, J., 2018. Detecting database file tampering through page carving. In: 21st International Conference on Extending Database Technology.
Wagner, J., Rasin, A., Heart, K., Jacob, R., Grier, J., 2019. Db3f & df-toolkit: the database forensic file format and the database forensic toolkit. Digit. Invest. 29, S42–S50.
Wagner, J., Rasin, A., Heart, K., Malik, T., Grier, J., 2020a. Df-toolkit: interacting with low-level database storage. Proc. VLDB Endowment 13 (12), 2845–2848.
Wagner, J., Rasin, A., Malik, T., Grier, J., Forensics, G., 2020b. Odsa: Open database storage access. In: 23rd International Conference on Extending Database Technology.
Wagner, J., Nissan, M.I., Rasin, A., 2023. Database memory forensics: identifying cache patterns for log verification. Forensic Sci. Int.: Digit. Invest. 45, 301567.
Yoon, J., Lee, S., 2018. A method and tool to recover data deleted from a mongodb. Digit. Invest. 24, 106–120.
Yoon, J., Jeong, D., Kang, C.-h., Lee, S., 2016. Forensic investigation framework for the document store nosql dbms: mongodb as a case study. Digit. Invest. 17, 53–65.