VDBFUZZ: Understanding and Detecting Crash Bugs in Vector Database Management Systems

Shenao Wang[*][†]
shenaowang@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Zhao Liu[*]
r3pwnx@gmail.com
360 AI Security Lab
Beijing, China

Yanjie Zhao[†]
yanjie_zhao@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Quanchen Zou[‡]
zouquanchen@gmail.com
360 AI Security Lab
Beijing, China

Haoyu Wang[†][‡]
haoyuwang@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

Vector Database Management Systems (VDBMSs) have become critical in LLM-integrated applications. However, their inherent complexity, including high-dimensional data structures, diverse indexing strategies, and heterogeneous implementations, makes them prone to reliability issues. Among these, crash bugs caused by boundary condition failures, such as invalid configurations and mismatched data dimensions, are particularly severe. These bugs can result in serious consequences like data loss, corrupted indexes, and cascading failures. To address this gap, we propose VDBFUZZ, the first fuzzing framework specifically designed to detect VDBMS crash bugs through boundary value testing. VDBFUZZ systematically leverages techniques to collect high-quality seeds, generate edge-case inputs, and explore complex API interactions. We evaluated VDBFUZZ on 8 representative VDBMSs, including native systems (e.g., Weaviate, Milvus), libraries (e.g., Faiss, hnswlib), and extended systems (e.g., pgvector, sqlite-vec). VDBFUZZ achieved up to 3x higher code coverage compared to state-of-the-art tools such as RESTler and Schemathesis, uncovering 19 previously unknown bugs, including 13 crash vulnerabilities and 6 runtime exceptions. These results highlight VDBFUZZ's effectiveness in improving the robustness and reliability of VDBMSs.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Denial-of-service attacks*; • **Software and its engineering → Software testing and debugging**.

[*]Both authors contributed equally to this research.
[†]Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.
[‡]Haoyu Wang (haoyuwang@hust.edu.cn) and Quanchen Zou (zouquanchen@gmail.com) are the corresponding authors.

ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773139

ACM Reference Format:
Shenao Wang, Zhao Liu, Yanjie Zhao, Quanchen Zou, and Haoyu Wang. 2026. VDBFUZZ: Understanding and Detecting Crash Bugs in Vector Database Management Systems. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773139

1 INTRODUCTION

Vector Database Management Systems (VDBMSs) have emerged as a crucial technology in the era of Large Language Models (LLMs). Unlike traditional relational or NoSQL databases [22], which primarily handle structured or semi-structured data, VDBMSs are specifically optimized for storing and querying vector embeddings, which are numerical representations of multimodal data such as text, images, and audio [41, 53]. Popular frameworks such as Pinecone [43], Milvus [35], and Weaviate [61] have become indispensable in powering various applications, including search engines [53], recommendation systems [53, 56], and LLM-based workflows [26, 70]. In particular, VDBMSs play a critical role in enabling advanced LLM capabilities, such as Retrieval-Augmented Generation (RAG) [26] and long-term memory [24, 25]. As LLMs are increasingly integrated into real-world applications, VDBMSs have become the backbone for supporting their scalability and effectiveness.

Despite their growing adoption and importance, ensuring the reliability and robustness of VDBMS remains a significant challenge [58, 64]. These systems are inherently complex due to their reliance on high-dimensional data structures, various indexing strategies, and heterogeneous language implementations [35, 46, 61]. The complexity makes them particularly susceptible to reliability issues [58, 64], such as crashes, unexpected errors, or degraded performance. According to a recent empirical study [64], crash bugs account for 15.1% of all observed defects in VDBMS, frequently manifesting as abrupt process terminations, segmentation faults, or unrecoverable runtime exceptions. For example, systems like Qdrant [46] and txtai [39] exhibit high crash rates (up to 40.7% of their total bugs). These failures can lead to severe consequences such as data loss, corrupted vector indexes, or cascading failures in downstream applications, emphasizing the urgent need for systematic methods to detect and mitigate crash bugs in VDBMSs.

However, existing works on VDBMS testing have largely focused on performance evaluation [4, 38, 45, 71], with limited emphasis on systematic reliability testing. Performance testing primarily measures efficiency under limited predefined scenarios, but it fails to uncover deeper issues that require exploring diverse and unpredictable inputs beyond predefined ones. The most relevant work MeTMaP has introduced metamorphic testing for detecting false vector matching problems in LLM augmented generation. Nevertheless, it focuses on algorithm-level inaccuracies in vector similarity computations rather than addressing implementation-specific defects in VDBMSs. While traditional testing approaches, such as unit tests [15, 60] and integration tests [34], are commonly used to ensure the functionality of VDBMSs, they primarily rely on manually written test cases by developers and often fail to uncover deeper reliability issues and edge-case vulnerabilities [17, 59]. Fuzz testing has proven to be an effective approach for uncovering hidden bugs and vulnerabilities in other domains, such as traditional DBMS [19, 23, 53, 67], operating systems [7, 10, 68], and other various applications [8, 11, 65]. Despite these advancements, there is still a significant gap in applying fuzz testing to VDBMSs, particularly for detecting implementation-level reliability issues.

To address these gaps, we conducted a preliminary study to analyze the characteristics of crash bugs and developed VDBFuzz, the first fuzzing framework specifically designed for VDBMSs. Our preliminary study of crash bugs across 15 state-of-the-art VDBMSs revealed that 44.5% of these defects arise from improper handling of boundary inputs (e.g., extreme parameter values, mismatched data dimensions, or malformed structures) and 15.6% of them arise from dependencies on specific API call sequences (e.g., incorrect order, omission, or redundancy of operations). Based on these insights, VDBFuzz systematically explores crash-inducing behaviors through three key stages: 1) *Seed Collection*, which extracts representative API usage patterns from unit tests, example scripts, and documentation to form a structured test corpus; 2) *Template-based Input Mutation*, which generates diverse inputs by applying reusable mutation patterns targeting boundary values, invalid configurations, and edge cases; and 3) *API Sequence Mutation*, which evaluates the robustness of VDBMSs by mutating API call sequences to uncover defects caused by logical inconsistencies or invalid state transitions. By combining these components, VDBFuzz is capable of systematically testing VDBMSs and detecting VDBMS crash bugs.

To evaluate the effectiveness of VDBFuzz, we conducted comprehensive experiments on 8 representative VDBMSs, including native VDBMSs (e.g., Weaviate, Milvus, Qdrant), VDBMS libraries (e.g., Faiss, HNSWLib, Annoy), and extended VDBMSs (e.g., Pgvector, Sqlite-vec). When compared to state-of-the-art fuzzing tools such as RESTler and Schemathesis, VDBFuzz demonstrated superior performance on native VDBMSs, achieving 3x higher coverage on average. For example, in Weaviate, VDBFuzz covered 15,485 lines of code, compared to RESTler's 4,519 and Schemathesis's 1,811, over the same 120-minute testing period. An ablation study further confirmed the complementary nature of VDBFuzz's mutation strategies, with each component contributing meaningfully to the overall coverage. Overall, VDBFuzz uncovered 19 previously unknown bugs across all tested systems, including 13 crash vulnerabilities and 6 runtime exceptions, with root causes linked to boundary condition failures such as dimension mismatches, invalid configurations, and malformed inputs. These results validate VDBFuzz's effectiveness in systematically detecting crash bugs, significantly improving the robustness and reliability of VDBMSs.

Contributions. To summarize, this paper makes the following key contributions:

- **Investigation of Crash Bugs in VDBMSs.** We conducted a detailed analysis of crash bugs across 15 VDBMSs, revealing that 44.5% of defects arise from improper handling of boundary inputs, while 15.6% stem from invalid API call sequences. These findings informed the design of VDBFuzz to systematically address these critical vulnerabilities.
- **Design and Implementation of VDBFuzz:** We propose VDBFuzz, the first fuzzing framework specifically designed for VDBMSs. VDBFuzz systematically explores crash-inducing behaviors by analyzing real-world API usage, generating diverse edge-case inputs, and testing complex API interactions to ensure robustness and reliability.
- **Real-world Impact:** We evaluated VDBFuzz on 8 representative VDBMSs. VDBFuzz achieved 3x higher code coverage compared to state-of-the-art fuzzing tools, uncovering 19 previously unknown bugs, including 13 crash vulnerabilities and 6 runtime exceptions.

Artifact Availability. We have publicly released the full source code of VDBFuzz at https://github.com/security-pride/VDBFuzz.

2 BACKGROUND AND MOTIVATION

Current VDBMS. VDBMSs are specialized databases designed to handle high-dimensional vector embeddings. A typical VDBMS architecture separates the server and client components to enable scalability and flexibility [41, 58]. The server handles the core functionalities, such as vector storage, index construction, and query processing, while the client provides interfaces and wrappers for interacting with the server. This separation allows VDBMSs to support multiple programming languages and frameworks, enabling developers to integrate them easily into diverse applications. As shown in Table 1, VDBMSs can be categorized into three main types based on their design and functionality.

Table 1: Comparison of Popular VDBMS by Category.

Category	VDBMS	Star	Fork	Server	Client
Library	Faiss [16]	37.3k	4.1k	C++	Python
	Hnswlib [40]	4.9k	731	C++	Python
	Annoy [51]	14k	1.2k	C++	Python
Native	Weaviate [61]	14.7k	1.1k	Go	Py/JS/Java/Go
	Milvus [35]	37.7k	3.4k	Go	Py/JS/Java/Go
	Qdrant [46]	26.2k	1.8k	Rust	Py/JS/Java/Rust/Go
Extended	pgvector [44]	17.7k	896	C	Py/JS/Java/Go
	sqlite-vec [52]	6.2k	233	C	Python

(1) Vector Libraries: These libraries are typically foundational implementations of indexing or search algorithms, designed to provide high-performance vector operations. Examples include Faiss [16],

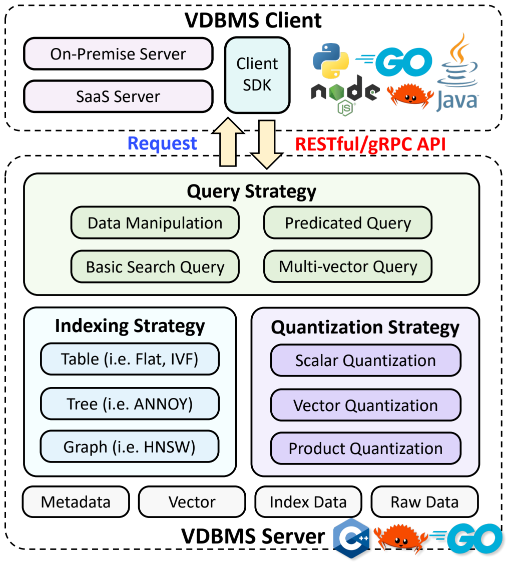


Figure 1: Typical architecture of VDBMSs.

Hnswlib [40], and Annoy [51]. They operate as lightweight libraries embedded directly into applications and are implemented in high-performance languages like C++, with bindings available for other languages such as Python. While vector libraries excel in speed and efficiency, they often lack the complete functionality of a VDBMS, such as user access control and distributed deployment.

(2) Native VDBMS: These are purpose-built systems optimized to handle vector workloads natively, providing a complete suite of functionalities for vector similarity search, including dynamic index updates, metadata filtering, and multi-vector queries [41, 53]. Examples of native VDBMS include Milvus [35], Qdrant [46], and Weaviate [61]. These systems typically expose their functionalities via network APIs, such as REST or gRPC, enabling distributed deployment and large-scale production use cases.

(3) Extended Databases with Vector Support: Extended databases are traditional databases, such as relational or NoSQL systems, that incorporate additional functionality to support vector similarity searches [9, 69]. These systems typically rely on plugins, extensions, or external libraries to perform vector-related tasks. Examples include PostgreSQL [44] with the pgvector extension [42], Sqlite [52] with vector similarity search [3], and MongoDB with integrated vector search capabilities [36]. These systems allow organizations to leverage existing database infrastructure while adding vector similarity search functionality [66].

VDBMS Workflow. The architecture of a VDBMS is designed to efficiently handle the entire lifecycle of vector data, from storage to indexing and query execution. A typical workflow within a VDBMS involves three key steps, as illustrated in Figure 1. First, vector data, typically represented as dense fixed-length floating-point arrays, is stored alongside metadata such as labels and timestamps to enable hybrid queries. Next, the system constructs indexes to accelerate similarity searches [63]. Popular indexing techniques include table-based indexes (e.g. LSH [6, 31] and IVF [18]), tree-based approaches (e.g., ANNOY [51]), graph-based methods (e.g., HNSW [32]), and quantization techniques (e.g., PQ [18, 28]). Finally, during query execution, the system retrieves approximate nearest neighbors from the index and applies metadata filters to refine results. VDBMSs support various distance metrics, such as Euclidean distance and cosine similarity, and some systems, like Milvus and Qdrant, allow multi-vector queries to aggregate results dynamically.

VDBMS Bugs. While VDBMSs provide efficient solutions for managing high-dimensional vector embeddings, they are not immune to implementation flaws, which can lead to severe system vulnerabilities. VDBMS bugs refer to defects in the implementation of vector storage, indexing, or query execution processes that may cause system crashes or incorrect behaviors. For instance, a notable example is a bug in the Faiss similarity search library's `IndexIVFPQFastScan` implementation[1]. As shown in Listing 1, when the `nlist` parameter (number of Voronoi cells) is not byte-aligned, attempting to reconstruct vectors via `reconstruct` or `reconstruct_batch` triggers out-of-bounds memory access due to invalid decoding of internal codes.

```
index = faiss.IndexIVFPQFastScan(
    faiss.IndexFlatL2(dim),
    dim,
    nlist=100,
    dim // 2,
    4,
    faiss.METRIC_L2
)
index.train(vec)
```

Listing 1: A crash bug in Faiss `IndexIVFPQFastScan` caused by non-byte-aligned `nlist` values.

3 A PRELIMINARY STUDY

To gain a deeper understanding of crash bugs in VDBMSs, we conducted a preliminary study to investigate the root cause of these defects. This study aims to identify the functions where crash bugs originate, the characteristics of crash-inducing inputs, and any dependencies on specific function call sequences, thereby providing actionable insights for the design of VDBFuzz.

3.1 Study Overview

To systematically investigate the root causes of crash bugs in VDBMS, we design a methodology that includes data collection, filtering, and classification of crash-related defects.

Data Collection and Scope. We based our analysis on the dataset from Xie et al. [64], which includes 1,671 bug-fix pull requests (PRs) from 15 widely used open-source VDBMSs, including Faiss [16], Milvus [35], and Qdrant [46]. Using this dataset, we extracted and analyzed 247 crash-related PRs, which account for 15.1% of all recorded defects. To ensure the relevance of our study, we manually reviewed these PRs to confirm their association with crash symptoms, such as process termination, segmentation faults, or unrecoverable runtime exceptions. To ensure the quality of our dataset, we applied the following filtering criteria: 1) The PR must document a root cause and include sufficient details, such as the triggering input or the affected function; 2) The PR must address an

[1]https://github.com/facebookresearch/faiss/issues/4089

Table 2: The number of collected VDBMS crash bugs.

VDBMS	Ori.	Filt.	Storage	Index	Query	Final
Chroma [12]	15	9	3	2	1	6
DeepLake [1]	18	11	1	–	6	7
Faiss [16]	1	-	–	1	–	1
hnswlib [40]	2	-	–	1	1	2
LanceDB [29]	13	7	2	1	3	6
Marqo [33]	8	2	1	–	5	6
Milvus [35]	62	17	9	3	33	45
pgvecto [42]	15	5	1	2	7	10
Qdrant [46]	11	4	5	1	1	7
txtai [39]	2	1	–	–	1	1
Usearch [54]	8	4	–	1	3	4
Vespa [55]	23	12	–	–	11	11
Voyager [2]	2	-	–	–	2	2
Weaviate [61]	26	6	10	2	8	20
Total	**206**	**78**	**32**	**14**	**82**	**128**

actual crash bug, confirmed by its resolution and subsequent merging into the main branch of the repository; 3) PRs with ambiguous descriptions or bugs unrelated to core VDBMS functionalities (e.g., storage, indexing, or query execution) were excluded. After filtering out ambiguous or irrelevant cases, we obtained a refined dataset of 128 confirmed crash bugs. Our analysis focused on three key aspects: the affected functions, the characteristics of their inputs, and their dependency on specific function call sequences.

Threats to Validity. Similar to other empirical studies, our results have the following limitations, which should be taken into account. 1) *Potential Bias.* Our analysis relies on the dataset from Xie et al. [64], which is composed of bug-fix pull requests from 15 open-source VDBMSs. This dataset may not fully represent all crash bugs encountered in other proprietary or less widely adopted VDBMSs. Additionally, the quality and completeness of the PR descriptions vary across projects, potentially impacting the accuracy of our classification. 2) *Filtering Criteria.* The filtering criteria we applied to identify crash-related bugs were designed to focus on core functionalities (storage, indexing, and query execution). However, this may have excluded other relevant crash bugs, such as those arising from deployment, configuration, or build processes, which might also impact system reliability in real-world scenarios. 3) *Manual Review and Labeling.* Although we manually reviewed and labeled the crash bugs, human error and subjectivity may have influenced the classification. Despite these limitations, the preliminary study provides a foundational understanding of crash bugs in VDBMSs and highlights areas for further research and tool development.

3.2 General Findings

Our analysis of 128 crash bugs across 15 VDBMSs reveals three complementary perspectives on their root causes: (1) *where* the crashes occur (i.e., affected functional modules), (2) *how* improper inputs trigger crashes, and (3) *when* specific operation sequences lead to failures. To make these relations explicit, Table 3 summarizes how the 128 crashes are partitioned by primary root cause category. In particular, 56 crashes are mainly triggered by improper inputs, and are further analyzed in Table 4; 20 crashes are mainly caused by operation-sequence dependencies, and are detailed in Table 5. The remaining crashes are attributed to other causes, such as data races, high-concurrency issues, dependency problems, and internal system errors that are not directly exposed at the user-input level.

Table 3: Overview of root causes for the 128 analyzed crash bugs.

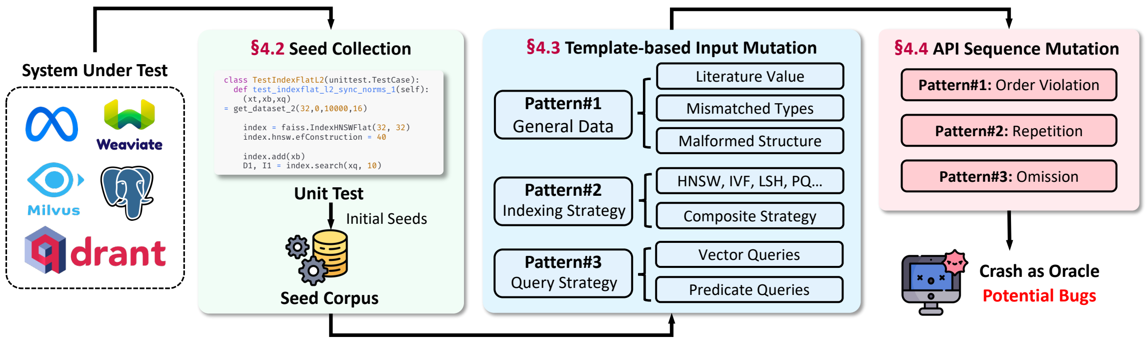
Root Cause Category	# Bugs	Percentage (%)
Input-related Crashes	56	43.8
Operation-sequence Dependencies	20	15.6
Data Races	11	8.6
High-concurrency Issues	6	4.7
Dependency Issues	4	3.1
Internal System Errors	31	24.2
Total	**128**	-

Table 4: Crash-inducing input categories with examples.

Input	Examples	Frequency (%)
Null/Zero Values	Zero-length vectors	21 (37.5%)
Out-of-Bounds Values	Negative indices	7 (12.5%)
Type Mismatches	Mismatched data types	8 (14.3%)
Size Constraints	Vector size beyond limits	3 (5.4%)
Index/Query Constraints	Invalid search parameters	17 (30.4%)
Total	–	**56 (100.0%)**

Functions Affected by Crash Bugs. Our analysis shows that crash bugs are primarily concentrated in three core functions: storage, indexing, and query execution. Query execution functions are the most affected, accounting for 64.1% of the total crash bugs. These often involve complex logic for handling vector search queries, ranking, and filtering, making them particularly prone to defects. Storage-related functions, responsible for persisting and retrieving vector data, account for 25% of the crashes. Indexing-related functions, which manage the creation and maintenance of vector indices, contribute to 10.9% of the crash bugs.

Crash-Inducing Inputs. As shown in Table 4, crash-inducing inputs can be classified into five primary categories: null or zero values, excessively large or negative values, type mismatches, violations of size constraints, and configuration constraints. Null or zero values caused the most crashes (21 out of 56, 37.5%), often leading to invalid states or unexpected behaviors in query execution or storage operations. Out-of-bounds values, such as excessively large or negative inputs (e.g., passing a vector of unsupported dimensions or negative indices), accounted for 12.5% of crashes, frequently resulting in memory-related issues like buffer overflows or segmentation faults. Type mismatches, where the input type did not match the expected type (e.g., providing a string where a numeric vector ID was required), contributed to 14.3% of crashes, exposing weaknesses in type validation mechanisms. Violations of size constraints, such as inputs exceeding the maximum allowed dimensions for vectors or batch sizes, led to 5.4% of crashes. Configuration constraints, specific to indexing or query API parameters (e.g., incorrectly configuring the dimensionality of an index or the number of nearest neighbors for a search query), accounted for 30.4% of crashes.



```
class TestIndexFlatL2(unittest.TestCase):
  def test_indexflat_l2_sync_norms_1(self):
    (xt,xb,xq)
= get_dataset_2(32,0,10000,16)

    index = faiss.IndexHNSWFlat(32, 32)
    index.hnsw.efConstruction = 40

    index.add(xb)
    D1, I1 = index.search(xq, 10)
```

Figure 2: The overall workflow of VDBFuzz.

Table 5: Crash bugs caused by operation dependencies.

Dependency Issue	Frequency	Percentage (%)
Missing Critical Operations	6	30.0
Incorrect Execution Order	7	35.0
Redundant Operations	5	25.0
Concurrent Operations	2	10.0
Total	**20**	**100.0**

Operation Sequence Dependencies. Our analysis identified 20 crash bugs caused by operation dependencies, as summarized in Table 5. The most frequent issue is incorrect execution order, which accounts for 35% of the cases. This issue arises when operations are performed in an improper sequence, such as querying uninitialized resources or deleting active resources, leading to invalid states or crashes. The second most common issue is missing critical operations, observed in 30% of the cases. These bugs occur when essential steps, such as resource initialization or cleanup, are skipped, causing the system to transition into an incomplete or inconsistent state. Redundant operations, responsible for 25% of the crashes, result from repeated execution of the same operation, often leading to resource conflicts, system instability, or excessive resource consumption. Finally, concurrent operations, observed in 10% of the cases, highlight the challenges of managing shared resources in a multi-threaded environment. These issues often manifest as race conditions or deadlocks due to inadequate synchronization.
Insights for VDBFuzz Design. Taken together, these findings directly motivate the design of VDBFuzz. First, the concentration of crashes in storage, indexing, and especially query execution functions suggests that a practical testing tool should prioritize these core APIs when generating test workloads. Second, the prevalence of input-related crashes indicates that systematically mutating user-provided parameters along common failure modes is essential to reveal crash issues. Third, the presence of operation-sequence–dependent crashes highlights the need to explore diverse API invocation orders and state transitions, rather than treating each API call in isolation.

4 WORKFLOW OF VDBFUZZ

The workflow of VDBFuzz is illustrated in Figure 2, which consists of three main stages: 1) *Seed Collection*, 2) *Template-based Input Mutation*, and 3) *API Sequence Mutation*. These stages are designed to systematically explore crash-inducing behaviors in VDBMSs.

4.1 Preliminary and Definition

To better describe the thoughts behind VDBFuzz, we provide the core definitions of concepts related to API constraints, input mutation, and cross-VDBMS test case transfer.
VDBMS API Constraints. Let $\mathcal{D}$ denote the set of all possible VDBMSs, and let $\mathcal{A}$ represent the set of all API functions exposed by VDBMSs. For a specific VDBMS $D \in \mathcal{D}$, the subset of APIs supported by D is $\mathcal{A}_D \subseteq \mathcal{A}$. Each API function $a \in \mathcal{A}_D$ accepts input parameters $\mathcal{P}_a$ and defines an input space $\mathcal{I}_a$. The valid input space $\mathcal{I}_a^{\text{valid}}$ is determined by a set of constraints C_a, where each constraint $c \in C_a$ is a predicate:

$$c : \mathcal{I}_a \to \{\text{true}, \text{false}\}.$$

The valid input space is then:

$$\mathcal{I}_a^{\text{valid}} = \{\mathbf{x} \in \mathcal{I}_a \mid \forall c \in C_a, c(\mathbf{x}) = \text{true}\}.$$

In addition to exploring the valid input space, VDBFuzz also tests invalid or boundary inputs in $\mathcal{I}_a^{\text{variant}} = \mathcal{I}_a \setminus \mathcal{I}_a^{\text{valid}}$ to identify crash-inducing behaviors.
Input Mutation. An input mutation is a transformation μ applied to an input $\mathbf{x} \in \mathcal{I}_a$, producing a new input $\mathbf{x}' \in \mathcal{I}_a$. Formally:

$$\mu : \mathcal{I}_a \to \mathcal{I}_a, \quad \mathbf{x}' = \mu(\mathbf{x}).$$

The goal of mutation is to generate diverse inputs, including valid inputs in $\mathcal{I}_a^{\text{valid}}$ and boundary or invalid inputs in $\mathcal{I}_a^{\text{variant}}$.
API Sequence Mutation. API sequence mutation involves modifying the order or structure of a sequence of API calls to uncover crash-inducing behaviors. Let $\mathcal{S} = [s_1, s_2, \ldots, s_k]$ represent an ordered sequence of API calls, where each s_i is a tuple $(a, \mathbf{x})$ consisting of an API $a \in \mathcal{A}_D$ and its input $\mathbf{x}$. A sequence mutation ρ generates a new sequence $\mathcal{S}'$ as follows:

$$\rho : \mathcal{S} \to \mathcal{S}', \quad \mathcal{S}' = \rho(\mathcal{S}).$$

Such mutations include operations such as reordering API calls, omitting critical operations, or introducing redundant calls.

4.2 Seed Collection

The first step in VDBFuzz is the collection of initial seeds, which serve as the foundation for input mutation and sequence mutation.

Table 6: Summary of indexing strategies and their mutation rules.

Index	Key Parameters	Constraints	Mutation Examples
HNSW [40]	M, ef_{cons}, ef	$M, ef_{\text{cons}}, ef \in \mathbb{Z}^+$, $ef_{\text{cons}} \geq M$	$M < 0$, $ef_{\text{cons}} < M$, ef set to extreme values
IVF [18]	$n_{\text{lists}}, n_{\text{probe}}$	$n_{\text{lists}}, n_{\text{probe}} \in \mathbb{Z}^+$, $n_{\text{probe}} \leq n_{\text{lists}}$	$n_{\text{probe}} > n_{\text{lists}}$, $n_{\text{lists}} = -1$
PQ [28]	$n_{\text{subvectors}}, n_{\text{bits}}$	$n_{\text{subvectors}}, n_{\text{bits}} \in \mathbb{Z}^+$, dim mod $n_{\text{subvectors}} = 0$	$n_{\text{subvectors}} = -1$, $n_{\text{bits}} = 0$, dim mod $n_{\text{subvectors}} \neq 0$
LSH [31]	L, k, r	$L, k \in \mathbb{Z}^+$, $r \in \mathbb{R}^+$, $r > 0$	$L = 0$, $r = -1$, extreme values for L, k, r
Flat [18]	metric	metric $\in$ {L2, Inner Product, Cosine}	Mutate metric to unsupported values (e.g., `manhattan`)
ANNOY [51]	n_{trees}, k	$n_{\text{trees}} \in \mathbb{Z}^+$, $1 \leq k \leq \lvert D \rvert$	$n_{\text{trees}} = -1$, $k > \lvert D \rvert$, or extreme k that degrade performance

These seeds are derived from the Python client libraries of the target VDBMSs, leveraging their built-in unit tests and example scripts [15, 60]. The purpose of this step is to capture representative and valid API usages, ensuring that the seeds reflect real-world interactions with the system under test (SUT). To construct the initial seed corpus, VDBFuzz first analyzes the unit test suites provided by the VDBMS. Each test case is parsed to extract API calls, input parameters, and their dependencies. For example, a unit test for vector indexing might include API calls for creating an index, adding vectors, and performing queries. These calls are recorded in sequence, preserving their logical relationships. Additionally, example scripts and usage documentation are processed in a similar manner to supplement the seed corpus with diverse API patterns not covered by the unit tests. The resulting seed corpus is a structured collection of valid API calls, each represented as a tuple $(a, \mathbf{x})$, where a is the API function and $\mathbf{x}$ is the input parameter set.

4.3 Pattern-based Input Mutation

To systematically uncover boundary inputs to trigger bugs in VDBMSs, we formalize input generation and mutation strategies into reusable patterns. These patterns are specifically designed to construct boundary inputs that expose vulnerabilities in the handling of edge cases, invalid configurations, and unexpected parameter interactions. Below, we describe the patterns for general data mutation, index strategy mutation, and query strategy mutation.

Pattern 1: General Data Mutation. General data mutation systematically generates inputs in $\mathcal{I}_a^{\text{variant}}$, targeting common failure modes such as type errors, boundary violations, and structural inconsistencies. These mutations aim to test the robustness of the database's parsers, execution engines, and indexing mechanisms when handling edge cases. Based on our analysis, approximately 12.3% of observed bugs result from extreme numerical values, 38.6% from zero or null values, and 14.0% from malformed or crafted data structures. To address these failure modes, we define the following rules for generating boundary literal values, mismatched types, malformed structures, and mismatched dimensions.

Sub-pattern 1.1: Boundary Literal Value. This sub-pattern focuses on constructing boundary literal values. These include numerical limits such as minimum, maximum, and zero, as well as structural emptiness, such as `NULL`, empty strings, empty lists, and empty dictionaries. For example, numerical boundaries like `-99999` (minimum) and `99999` (maximum) can be used to test the behavior under extreme conditions. Similarly, structural emptiness, such as an empty list (`[]`) or an empty dictionary (`{}`), helps evaluate how the system handles uninitialized or missing data. Formally, boundary values can be defined as:

$$\mathcal{I}_a^{\text{boundary}} \rightarrow \{\pm 0.999...99, \pm 999...99, 0, \texttt{NULL}, `', [], \{\}\}$$

Sub-pattern 1.2: Mismatched Types. This sub-pattern targets inputs in $\mathcal{I}_a^{\text{variant}}$ where the type of a value does not match the expected type. Examples include providing a string where an integer is expected, or passing a list (`[1, 2, 3]`) where a floating-point value is required. Such mutations can expose flaws in type-checking mechanisms or unexpected type coercion behaviors. Formally, invalid types are defined as:

$$\mathcal{I}_a^{\text{type}} = \{\mathbf{x} \in \mathcal{I}_a \mid \mathbf{x} \notin \text{ExpectedType}\}.$$

Sub-pattern 1.3: Malformed Structure. This sub-pattern generates inputs with invalid structures to test the resilience of parsers and execution engines. Such inputs include corrupted JSON objects, incomplete filter definitions, or misaligned data structures. For example, a malformed JSON object like `{"key":}` (missing value) or an incomplete filter like `filter = {"field":}` (missing condition) can reveal vulnerabilities in input validation. Formally, malformed structures are defined as:

$$\mathcal{I}_a^{\text{malformed}} = \{\mathbf{x} \in \mathcal{I}_a \mid M_{\text{json}}(\mathbf{x}) \vee M_{\text{filter}}(\mathbf{x}) \vee M_{\text{schema}}(\mathbf{x})\},$$

Pattern 2: Indexing Strategy Mutation. Indexing strategy mutation focuses on systematically exploring invalid or boundary configurations of key parameters used for creating vector indexes. Incorrect or extreme configurations, such as those in HNSW, IVF, PQ, and LSH indexing methods, can lead to indexing failures, degraded query performance, inaccurate results, or even system crashes. This pattern evaluates how well the VDBMS handles misconfigurations, extreme values, and invalid combinations. Based on the abstraction of indexing methods, we define two overarching mutation patterns.

Sub-pattern 2.1: Single Indexing Strategy Mutation. This pattern focuses on parameters of individual indexing methods (e.g., HNSW, IVF, PQ, and LSH). Each method defines a unique set of key parameters, such as M, ef_{cons}, and ef in HNSW, or n_{lists} and n_{probe} in IVF. These parameters have specific constraints (e.g., positivity, ranges, interdependencies), violations of which can lead to indexing errors or performance degradation. The general mutation space for single indexing strategies is defined as:

$$\mathcal{I}_{\text{index}}^{\text{variant}} = \{p_i \mid p_i \notin \text{ValidRange}(p_i) \vee \text{Violation}(p_1, p_2, \ldots, p_n)\},$$

where p_i represents a parameter, and constraints are method-specific (e.g., $ef_{\text{cons}} \geq M$ in HNSW, $n_{\text{probe}} \leq n_{\text{lists}}$ in IVF). Table 6 summarizes the parameters, constraints, and mutation strategies for the most commonly used indexing methods.

Table 7: Constraints and mutations for vector queries.

Parameter	Constraints and Mutations
k (number of results)	**Valid:** $k \in \mathbb{Z}^+$, $1 \le k \le \lvert D\rvert$ **Mutations:** • $k = 0$ (empty results) • $k < 0$ (e.g., $k = -1$) • $k > \lvert D\rvert$ (exceeds dataset size) • Non-integer types (e.g., $k = 3.5$, $k =$ "ten")
$\vec{q}$ (query vector)	**Valid:** $\vec{q} \in \mathbb{R}^d$, $d > 0$ **Mutations:** • Incorrect dimensionality ($d' \neq d$) • Malformed values (e.g., `NaN`, `Inf`) • Extreme values (e.g., excessively large)
$\mathcal{M}$ (distance metric)	**Valid:** $\mathcal{M} \in \mathcal{M}_{\text{valid}}$ **Mutations:** • Unsupported metrics (e.g., `cosine` for `L2`)
o (offset for pagination)	**Valid:** $o \in \mathbb{Z}$, $o \ge 0$ **Mutations:** • Negative values (e.g., $o = -1$) • Non-integer values (e.g., $o = 1.5$) • Excessively large offsets (e.g., $o > \lvert D\rvert$)

Pattern 2.2: Composite Indexing Strategy Mutation. This pattern targets combinations of multiple indexing methods, such as IVF-PQ or HNSW-PQ, where parameters from different methods must satisfy simultaneous constraints. The mutation space for composite strategies is defined as:

$$\mathcal{I}^{\text{variant}}_{\text{composite}} = \cup_{i=1}^{n} \mathcal{I}^{\text{variant}}_{i},$$

where $\mathcal{I}^{\text{variant}}_{i}$ represents the invalid parameter space of the i-th component indexing method. For example, in IVF-PQ, $n_{\text{subvectors}}$ can be set to a non-divisible value while $n_{\text{probe}} > n_{\text{lists}}$, exposing flaws in parameter validation across strategies.

Pattern 3: Query Parameter Mutation. Query parameter mutation focuses on exploring the robustness of VDBMS query functionalities by systematically testing various query parameters. Querying is the core operation in vector databases, and the behavior of queries is shaped by parameters such as the number of results to return, query vectors, filtering conditions, distance metrics, and pagination settings. Misconfigured parameters, overly complex filters, or invalid combinations can lead to logic errors, degraded performance, or even system failures. This pattern is divided into two sub-patterns: *vector queries* and *predicate queries.*

Sub-pattern 3.1: Vector Queries. Vector queries are defined by parameters such as k (the number of results to return), the query vector $\vec{q}$, the distance metric $\mathcal{M}$, and the pagination offset o. These parameters must satisfy specific constraints, as summarized in Table 7. Mutation strategies explore invalid configurations, such as setting k to values outside its valid range, introducing malformed query vectors, using unsupported distance metrics, or applying invalid pagination offsets. Formally, the invalid parameter space for vector queries is defined as: Formally, the invalid parameter space for vector queries can be defined as:

$$\mathcal{I}^{\text{variant}}_{\text{vector}} = \{(k, \vec{q}, \mathcal{M}, o) \mid \begin{cases} k \notin \mathbb{Z}^+ \text{ or } k > |D|, \\ \vec{q} \notin \mathbb{R}^d, \\ \mathcal{M} \notin \mathcal{M}_{\text{valid}}, \\ o \notin \mathbb{Z} \text{ or } o < 0 \end{cases} \}.$$

Sub-pattern 3.2: Predicate Queries. Predicate queries extend vector queries by introducing filtering conditions F, which constrain results based on metadata attributes or logical expressions. These filters enable more complex query scenarios, such as combining similarity search with metadata-based constraints.

Filters F can fail due to syntax errors (e.g., mismatched parentheses, invalid operators) or semantic errors (e.g., referencing nonexistent fields, unsupported operations, or logically contradictory conditions). For example, a filter might reference a non-existent metadata field, apply a range query (`gte`/`lte`) on a string field, or include logically contradictory constraints such as `field > 10 AND field < 5`. Geospatial filters may also fail due to malformed coordinates or nonsensical ranges. Additionally, overly complex filters with deeply nested `AND`/`OR` combinations can degrade query performance, especially when combined with similarity search. Formally, the invalid space for predicate queries is defined as:

$$\mathcal{I}^{\text{variant}}_{\text{predicate}} = \{(k, \vec{q}, \mathcal{M}, o, F) \mid \begin{cases} (k, \vec{q}, \mathcal{M}, o) \in \mathcal{I}^{\text{variant}}_{\text{vector}}, \\ F \in \mathcal{F}_{\text{invalid}} \end{cases} \}.$$

Here, $\mathcal{F}_{\text{invalid}}$ represents the space of invalid filters, encompassing syntax errors, semantic errors, and logical contradictions.

4.4 API Sequence Mutation.

API sequence mutation systematically evaluates the robustness and correctness of VDBMSs by mutating sequences of API calls. Let $\Sigma = \{s_1, s_2, \ldots, s_n\}$ denote the set of all valid API operations, where each s_i represents an individual API call. A valid API sequence is defined as an ordered list of calls:

$$\mathcal{S}_{\text{valid}} = \langle s_1, s_2, \ldots, s_k \rangle,$$

where each s_i must satisfy its precondition $P(s_i)$, and the sequence collectively ensures valid state transitions. For example, a normal API sequence might comprise operations such as `connect`, `create_collection`, `add_object`, `search`, and `close`, executed in the correct order. Deviations from this order or sequence can lead to various invalid states. Mutation strategies systematically violate one or more constraints of $\mathcal{S}_{\text{valid}}$. These strategies can be categorized into three primary types: *Order Violation*, *Repetition*, and *Omission*, as summarized in Table 8.

Pattern 1: Order Violation. Order violation involves reordering API operations in a way that disrupts the sequence's logical flow and violates the precondition constraints of one or more operations. For instance, calling `search` before `add_object` would result in a state where the search operation is executed on an empty or uninitialized state, thereby violating the expected preconditions.

Pattern 2: Repetition. Repetition of operations focuses on testing robustness when specific API calls are invoked multiple times, potentially exceeding their intended usage. For example, repeatedly invoking `connect` without corresponding `close` calls can lead to resource exhaustion, such as the depletion of available connections.

Table 8: API sequence mutation strategies.

Mutation Strategy	Mutation Description
Order Violation	Reorder operations in the sequence $\mathcal{S}$ such that at least one s_i violates its precondition $P(s_i)$. **Example:** `search` before `add_object`
Repetition of Operations	Repeat an operation $s_r \in \mathcal{S}$. **Example:** `connect` → `connect`
Omission of Operations	Remove one or more critical operations from $\mathcal{S}$. **Example:** Skipping `add_object`

Pattern 3: Omission. Omission of operations involves skipping critical API calls required to maintain valid state transitions. For example, omitting the `add_object` call that expects objects to be added to a collection before a `search` operation can leave the system in an incomplete state, leading to failed expectations or errors.

The three mutation strategies collectively aim to create scenarios that stress the system's ability to maintain correctness under adverse conditions. Additionally, these strategies are invaluable for testing concurrency and distributed VDBMSs, where interactions between multiple API calls can lead to race conditions, deadlocks, or inconsistencies in shared resources.

5 EVALUATION

5.1 Implementation

The implementation of VDBFuzz is designed to uncover boundary and edge-case issues across different types of VDBMSs, including native VDBMSs (e.g., Weaviate [61], Milvus [35], and Qdrant [46]), VDBMS libraries (e.g., Faiss [16], HNSWLib [40], and Annoy [51]), and extended VDBMSs (e.g., pgvector [42] and sqlite-vec [3]). Native VDBMSs typically expose their core functionalities via HTTP or gRPC APIs, while VDBMS libraries and extended VDBMSs rely on Python-based client interfaces for interaction. To ensure comprehensive testing across these diverse implementations, we tailored VDBFuzz to the specific characteristics of each category.

For native VDBMS like Weaviate, Milvus, and Qdrant, we leverage the Python client libraries to construct a fuzz harness for generating and executing API sequences, while relying on hooks at the HTTP and gRPC communication to perform seed mutation. By intercepting the communication between the Python client and the database server, we apply mutation strategies directly to the requests being sent to the core implementation. For VDBMS libraries such as Faiss, HNSWLib, and Annoy, which do not expose API endpoints, and for extended VDBMS systems like PostgreSQL, we implemented our fuzzing strategies directly at the Python layer. This involved creating mutation strategies to test various input parameters and configurations during indexing and querying.

Specifically, we have implemented a prototype of VDBFuzz using over 4K lines of code (LoC) with Python, excluding any third-party libraries or open-source tools. The HTTP and gRPC interceptors, comprising around 1,300 lines of code, are responsible for capturing and mutating communication payloads. The core fuzzer logic, including modules for API sequence mutation, data mutation, and template generation, takes up approximately 3,300 lines and is designed to generate diverse and complex test cases. The template generation module ensures that inputs adhere to the structural requirements of the target APIs while systematically introducing invalid or boundary values. Additionally, auxiliary modules handle tasks such as parsing test outputs, managing mutation strategies, and integrating with the fuzz harness.

5.2 Evaluation Setup

Running Environment. The evaluation of VDBFuzz was conducted on a high-performance server running Ubuntu Linux 22.04, equipped with two AMD EPYC Milan 7713 CPUs (2.0 GHz, 64 cores, 128 threads each), 512 GB of RAM, and four 7.68 TB NVMe SSDs. To ensure comprehensive coverage analysis across the diverse implementations of VDBMSs written in C++, Rust, and Go, we employed language-specific workflows. For Rust-based systems (Qdrant), the code was compiled with coverage instrumentation enabled (`-Cinstrument-coverage`), and profiling data was collected as `.profraw` files, which were merged using llvm-profdata and analyzed with grcov [37] to generate branch-level HTML reports. For Go-based implementations (Weaviate and Milvus), we utilized goc [47], a coverage tool specifically designed for the Go language, to instrument the code and generate detailed coverage reports during test execution.

Research Questions. To evaluate the performance of VDBFuzz, we aim to answer the following research questions (RQs):

- **RQ1: Comparison with Existing Tools.** How does VDBFuzz compare to state-of-the-art fuzzers and testing tools in terms of code coverage across different VDBMSs?
- **RQ2: Ablation Study.** How does each mutation strategy individually contribute to the overall performance of VDBFuzz?
- **RQ3: Real-World Bugs Discovered.** What types of real-world boundary and edge-case vulnerabilities were discovered by VDBFuzz in tested VDBMSs?

5.3 Comparison with Existing Tools (RQ1)

To evaluate the effectiveness of VDBFuzz compared to existing state-of-the-art API fuzzing tools, we conducted a series of experiments using the three native VDBMSs in our evaluation: Weaviate, Qdrant, and Milvus. The tools selected for comparison were RESTler [13] and Schemathesis [14], as both are widely adopted for OpenAPI testing and have demonstrated strong results in identifying API vulnerabilities and achieving high code coverage.

Baseline Setup. For each VDBMS, we used its publicly available OpenAPI schema to guide the fuzzing process. Each tool was run for two hours (120 minutes) on each VDBMS to ensure fair resource allocation and allow sufficient time to explore the API space.

- **RESTler**: RESTler was configured in its default fuzzing mode, which performs stateful API fuzzing by generating API sequences based on dependencies inferred from the OpenAPI schema. It prioritizes breadth-first exploration of the API space.
- **Schemathesis**: Schemathesis was executed in its default property-based testing mode, which generates test cases by mutating inputs to API endpoints based on the OpenAPI schema. It focuses on identifying schema violations and edge cases.

Table 9: VDBFuzz's improvement in coverage compared to baseline fuzzers across Weaviate, Qdrant, and Milvus.

VDBMS	Baseline	Cov.	VDBFuzz	
Weaviate	RESTler	4,519 (7.0%)	15,485 (24.0%)	**+243.0%**
	Schemathesis	1,811 (2.8%)		**+755.1%**
Qdrant	RESTler	8,402 (9.1%)	37,403 (40.3%)	**+345.3%**
	Schemathesis	7,050 (7.6%)		**+430.6%**
Milvus	RESTler	6,331 (11.0%)	22,109 (38.5%)	**+249.2%**
	Schemathesis	4,849 (8.4%)		**+355.8%**

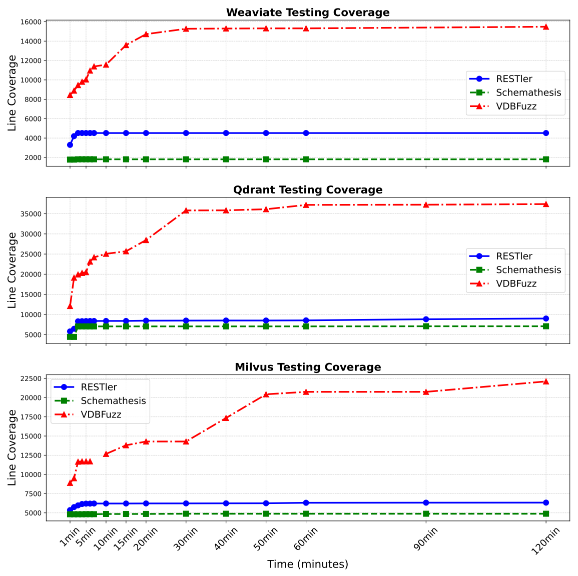


Figure 3: Code coverage comparison of RESTler, Schemathesis, and VDBFuzz on Weaviate, Qdrant, and Milvus.

- **VDBFuzz**: VDBFuzz was run in its full mutation mode, which combines API sequence mutation, input data mutation, and template-based indexing and query mutation strategies.

To measure the effectiveness of each tool, we recorded the total line coverage achieved over time. We report the mean line coverage over five runs of VDBFuzz and these baselines.

Comparison Results. Table 9 highlights the overall testing capabilities of VDBFuzz compared to baseline fuzzing tools RESTler and Schemathesis across Weaviate, Qdrant, and Milvus. The results show that VDBFuzz consistently outperformed the baseline tools by a substantial margin in both testing efficiency and effectiveness. On average, VDBFuzz achieved more than 3x the total line coverage of RESTler and over 4x the coverage of Schemathesis.

Specifically, we visualized the testing progress over time in Figure 3 to provide a detailed comparison of line coverage growth across the three VDBMSs. Within the first 10 minutes, VDBFuzz achieved significantly higher line coverage compared to the baseline tools, demonstrating its superior efficiency in exploring API spaces. For instance, on Weaviate, VDBFuzz reached 11,571 lines of code coverage within 10 minutes, compared to 4,519 lines for RESTler and 1,811 lines for Schemathesis. Over the entire 120-minute testing period, VDBFuzz maintained its advantage, achieving total coverage of 15,485 lines on Weaviate, 37,403 lines on Qdrant, and 22,109 lines on Milvus. These results represent approximately 3.4x to 4.2x higher coverage than RESTler and 4.5x to 8.5x higher coverage than Schemathesis across the three VDBMSs. Notably, RESTler and Schemathesis exhibited limited growth in coverage after the early stages of testing, with RESTler plateauing after just a few minutes and Schemathesis reaching its peak coverage at a significantly lower level, highlighting their limitations when applied to VDBMS fuzzing. In contrast, VDBFuzz demonstrated consistent improvement throughout the testing period, uncovering increasingly complex API interactions and implementation logic.

Table 10: Ablation study on mutation strategies across Weaviate, Qdrant, and Milvus.

Strategy	Weaviate		Qdrant		Milvus	
VDBFuzz	**15,485**		**37,403**		**22,109**	
VDBFuzz-seed-only	14,842	**-643**	36,342	**-1,061**	20,140	**-1,969**
VDBFuzz-w/o index	15,469	**-16**	36,493	**-910**	20,355	**-1,754**
VDBFuzz-w/o query	15,331	**-154**	36,550	**-853**	21,709	**-400**

5.4 Ablation Study (RQ2)

To evaluate the contribution of different mutation strategies in VDBFuzz, we conducted an ablation study by systematically disabling specific components and measuring their impact on line coverage across Weaviate, Qdrant, and Milvus. Table 10 summarizes the results, showing the total line coverage achieved by the full version of VDBFuzz compared to its ablated variants. The results indicate that the full version of VDBFuzz achieved the highest line coverage across all three VDBMSs, with 15,485 lines for Weaviate, 37,403 lines for Qdrant, and 22,109 lines for Milvus. Disabling specific components resulted in a consistent reduction in coverage, demonstrating the complementary nature of the mutation strategies. Below, we analyze the impact of each strategy:

- **Seed-only.** In this variant, VDBFuzz relied solely on the initial seeds. This resulted in the largest reduction in coverage, with decreases of 643 lines on Weaviate, 1,061 lines on Qdrant, and 1,969 lines on Milvus. These results highlight the importance of edge case mutations in uncovering deeper implementation logic that seed-only approaches fail to explore.
- **Index-specific Mutation.** When index-specific mutation was disabled, the coverage dropped by 16 lines on Weaviate, 910 lines on Qdrant, and 1,754 lines on Milvus. This strategy had a particularly significant impact on Milvus and Qdrant.
- **Query-specific Mutation.** Disabling the query-specific mutation strategy slightly reduced coverage, with decreases of 154 lines on Weaviate, 853 lines on Qdrant, and 400 lines on Milvus. While the impact of this strategy was smaller compared to index-specific mutation, it played a crucial role in improving the exploration of query-related API behaviors, especially on Weaviate, where the reduction was more pronounced.

Table 11: VDBFuzz has discovered 19 undisclosed crash bugs and exceptions.

VDBMS	Crash	Exception	Types
Weaviate	2	2	OOBR, Type Assertion Errors
Qdrant	3	1	OOM, FPE, DoS
Milvus	-	2	Query Inconsistencies
Faiss	3	-	OOBR, FPE, NPE
Hnswlib	1	-	DF, OOBR
Annoy	2	-	OOBR
pgvector	-	1	–
sqlite-vec	2	-	OOBR
Total	**13**	**6**	–

Overall, the ablation study demonstrates that all components of VDBFuzz contribute meaningfully to its effectiveness. While initial seeds provide a strong baseline, the addition of query-specific and index-specific strategies enables VDBFuzz to systematically and comprehensively explore the API space, achieving significantly higher coverage across diverse VDBMSs.

5.5 Real-world Bugs Discovered (RQ3)

To evaluate the effectiveness of VDBFuzz in uncovering real-world vulnerabilities, we applied it to test 8 VDBMS implementations, including native VDBMSs (Weaviate, Qdrant, and Milvus), VDBMS libraries (Faiss, Hnswlib, and Annoy), and extended VDBMSs (pgvector, sqlite-vec). For the evaluation, VDBFuzz generated and executed more than 10 million valid test cases, which cover strings specific to VDBMSs and particular data fields. VDBFuzz demonstrates the capability to uncover a wide range of security vulnerabilities, such as Out-of-Bound Read (OOBR), Float Point Exception (FPE), Double Free (DF). As illustrated in Table 11, it has successfully identified 13 crash bugs and 6 exception issues, which could potentially result in critical consequences, including sensitive data breaches and the malfunction of LLM services.

```
1  index := createEmptyHnswIndexForTests(t, testVectorForID)
2
3  for i, vec := range testVectors {
4      err := index.Add(uint64(i), vec)
5      require.Nil(t, err)
6  }
7
8  t.Run("searching within cluster 1", func(t *testing.T) {
9      position := 0
10     res, _, err := index.knnSearchByVector(testVectors[
           position], -1, 36, nil)
11     require.Nil(t, err)
12     assert.ElementsMatch(t, []uint64{0, 1, 2}, res)
13 })
```

Listing 2: Panic in knnSearchByVector with negative k.

Case Study#1 (Weaviate): Panic in knnSearchByVector. One notable vulnerability uncovered by VDBFuzz was an index out-of-bounds crash in Weaviate's HNSW implementation during a vector search operation, which is shown in Listing 2. The issue occurred when a negative value (k=-1) was passed to the knnSearchByVector function (line 10), which controls the number of nearest neighbors to retrieve. The negative value violated the function's implicit assumption that k would always be positive. This caused a panic due to an out-of-bounds access. This case highlights the importance of robust boundary checks for critical parameters to prevent such crashes, which could otherwise compromise the reliability and availability of the system.

Case Study#2 (Qdrant): Denial of Service. Another significant vulnerability identified by VDBFuzz was a Denial of Service (DoS) issue in Qdrant. This issue arises when an excessively large vector dimension (`dim`) is provided as input to the constructor, as shown in Listing 3 (line 7). The function `new` performs calculations based on the dimension size, such as determining `vector_size` and `chunk_capacity`. However, when `dim` is extremely large (e.g., 2^{63}), the multiplication `dim * mem::size_of::<T>()` can result in integer overflow, potentially leading to divide-by-zero in `chunk_capacity = CHUNK_SIZE / vector_size`. Such a panic results in the service crashing, causing a DoS. For VDBMSs that serve as the components of LLM-based applications, such DoS vulnerabilities are particularly severe, as they can directly render downstream retrieval-augmented generation or semantic search services unavailable.

```
1  // Vulnerable function in ChunkedVectors<T>
2  impl<T: Copy + Clone + Default> ChunkedVectors<T> {
3      pub fn new(dim: usize) -> Self {
4          assert_ne!(dim, 0, "Dimension cannot be 0");
5          let vector_size = dim * mem::size_of::<T>();
6          let chunk_capacity = CHUNK_SIZE / vector_size;
7          assert_ne!(chunk_capacity, 0, "Size is too big");
8      }
9  }
10 // PoC:
11 c = QdrantClient(host="127.0.0.1", port=6333)
12 c.recreate_collection(
13     collection_name="test",
14     vectors_config=models.VectorParams(size=2**63,
           distance=models.Distance.COSINE),
15 )
```

Listing 3: Vulnerability in ChunkedVectors<T> of Qdrant.

6 DISCUSSION

Limitations. While we implemented VDBFuzz for 8 VDBMSs, our analysis for RQ1 and RQ2 was conducted exclusively on native VDBMS, including Weaviate, Milvus, and Qdrant. These systems were chosen because they expose REST APIs, which provide a direct baseline for comparisons. Additionally, VDBFuzz currently only performs fuzzing for crash-related bugs in vector databases, as crashes present the most direct and observable oracle for identifying potential issues. This excludes deeper considerations of vector search correctness or performance-related bugs, which may also significantly impact real-world applications. Furthermore, VDBFuzz relies on pattern-based mutation on collected seeds, which may not comprehensively cover all API functionalities or edge cases.

Future Work. Future work should explore the development of an oracle for evaluating the correctness of vector search results, as the inherent fuzziness of vector similarity search makes traditional differential testing unsuitable. Another promising direction is the generation of more diverse and comprehensive seed inputs to expand API coverage. LLMs could play a significant role in generating diverse API interactions, given their ability to understand and replicate complex input structures. However, because VDBMS and its APIs evolve rapidly, methods must be devised to ensure that LLMs can stay updated with the latest syntax and functionalities.

7 RELATED WORK

VDBMS Testing. VDBMSs have become more and more critical in the LLM era, yet systematic reliability testing for these systems remains underexplored [58]. Existing works on VDBMS testing have primarily focused on evaluating performance metrics, such as query latency, throughput, and scalability [4, 38, 45, 71]. These studies provide valuable insights but overlook reliability and robustness. The most relevant work MeTMaP introduced metamorphic testing to detect false vector matching problems [26]. However, its focus is on algorithm-level inaccuracies in vector similarity computations rather than implementation-specific defects within VDBMSs. Unlike traditional databases, VDBMSs face distinct challenges due to their reliance on fuzzy semantics in similarity search and hybrid query processing [58, 64]. To bridge this gap, VDBFuzz introduces boundary-specific mutations to discover crash bugs in VDBMS.

DBMS Fuzzing. Fuzz testing has been widely applied to traditional DBMSs to uncover vulnerabilities [41], with existing techniques primarily focusing on crash testing [21, 50], performance testing [27, 30], and logical testing [5, 48, 49]. Additionally, tools tailored for relational databases [5, 57] and NoSQL databases [20, 62, 67] address their respective architectural challenges, such as query correctness, sharding, and flexible schema handling. While these fuzzing techniques have demonstrated effectiveness for traditional databases, they cannot be directly applied to VDBMSs. To this end, VDBFuzz introduces the first fuzz testing framework specifically designed for VDBMSs.

8 CONCLUSION

In this paper, we presented VDBFuzz, the first fuzzing framework specifically designed to detect crash bugs in VDBMSs. We began by conducting an in-depth investigation of crash bugs across state-of-the-art systems and identified boundary condition failures and API sequence dependencies as primary root causes. Based on these insights, VDBFuzz systematically explores crash-inducing behaviors through seed collection, template-based input mutation, and API sequence mutation. Our evaluation on 3 representative VDBMSs, including Weaviate, Milvus, and Qdrant, demonstrated VDBFuzz's effectiveness, achieving up to 3x higher code coverage compared to state-of-the-art fuzzing tools and uncovering 19 previously unknown crash bugs. These findings highlight the potential of VDBFuzz to improve the robustness and reliability of VDBMSs.

Acknowledgement

This work was supported in part by the National Natural Science Foundation of China (grants No.62572209, 62502168) and the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057).

REFERENCES

[1] activeloopai. 2025. DeepLake: Database for AI. https://github.com/activeloopai/deeplake. Accessed: 2025-07-15.
[2] adriecafe. 2025. Voyager. https://github.com/adrielcafe/voyager. Accessed: 2025-07-15.
[3] asg017. 2025. sqlite-vec: A vector search SQLite extension that runs anywhere! https://github.com/asg017/sqlite-vec. Accessed: 2025-07-15.
[4] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020). https://doi.org/10.1016/J.IS.2019.02.006
[5] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2060–2071. https://doi.org/10.1109/ICSE48619.2023.00174
[6] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. 2005. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th International Conference on World Wide Web* (Chiba, Japan) *(WWW '05)*. Association for Computing Machinery, New York, NY, USA, 651–660. https://doi.org/10.1145/1060745.1060840
[7] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2023. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/no-grammar-no-problem-towards-fuzzing-the-linux-kernel-without-system-call-descriptions/
[8] Shangtong Cao, Ningyu He, Xinyu She, Yixuan Zhang, Mu Zhang, and Haoyu Wang. 2024. WASMaker: Differential Testing of WebAssembly Runtimes via Semantic-Aware Binary Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1262–1273. https://doi.org/10.1145/3650212.3680358
[9] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3772–3785. https://doi.org/10.14778/3685800.3685805
[10] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, and Zhi Xue. 2022. SFuzz: Slice-based Fuzzing for Real-Time Operating Systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 485–498. https://doi.org/10.1145/3548606.3559367
[11] Weimin Chen, Xiapu Luo, Haipeng Cai, and Haoyu Wang. 2024. Towards Smart Contract Fuzzing on GPUs. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2255–2272. https://doi.org/10.1109/SP54263.2024.00229
[12] chroma-core. 2025. Chroma: The AI-native open-source embedding database. https://github.com/chroma-core/chroma. Accessed: 2025-07-15.
[13] Microsoft RESTler Contributors. 2025. RESTler: Stateful REST API Fuzzing Tool. https://github.com/microsoft/restler-fuzzer. Accessed: 2025-07-15.
[14] Schemathesis Contributors. 2025. Schemathesis: Catch API bugs before your users do. https://github.com/schemathesis/schemathesis. Version 4.0.9, Accessed: 2025-07-15.
[15] Facebook Research. 2025. Unit Test of Faiss. https://github.com/facebookresearch/faiss/tree/main/tests. Accessed: 2025-07-15.
[16] facebookresearch. 2025. Faiss: A library for efficient similarity search and clustering of dense vectors. https://github.com/facebookresearch/faiss. Accessed: 2025-07-15.
[17] Faiss. 2024. IndexIVFPQFastScan crashes with certain nlist values. https://github.com/facebookresearch/faiss/issues/4089. Accessed: 2025-07-15.
[18] Faiss. 2025. Faiss indexes. https://github.com/facebookresearch/faiss/wiki/Faiss-indexes. Accessed: 2025-07-15.
[19] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 146:1–146:12. https://doi.org/10.1145/3597503.3639210
[20] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin : Grammar-Free DBMS Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 49:1–49:12. https://doi.org/10.1145/3551349.3560431
[21] Jingzhou Fu, Jie Liang, Zhiyong Wu, Yanyang Zhao, Shanshan Li, and Yu Jiang. 2025. Understanding and Detecting SQL Function Bugs: Using Simple Boundary Arguments to Trigger Hundreds of DBMS Bugs. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 1061–1076. https://doi.org/10.1145/3689031.3696064
[22] Xiyue Gao, Zhuang Liu, Jiangtao Cui, Hui Li, Hui Zhang, Kewei Wei, and Kankan Zhao. 2023. A Comprehensive Survey on Database Management System Fuzzing: Techniques, Taxonomy and Experimental Comparison. *CoRR* abs/2311.06728 (2023). https://doi.org/10.48550/ARXIV.2311.06728 arXiv:2311.06728
[23] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 345–358. https://www.usenix.org/conference/atc23/presentation/hao
[24] Zihong He, Weizhe Lin, Hao Zheng, Fan Zhang, Matt W. Jones, Laurence Aitchison, Xuhai Xu, Miao Liu, Per Ola Kristensson, and Junxiao Shen. 2024. Human-inspired Perspectives: A Survey on AI Long-term Memory. *CoRR* abs/2411.00489

(2024). https://doi.org/10.48550/ARXIV.2411.00489 arXiv:2411.00489

[25] Xun Jiang, Feng Li, Han Zhao, Jiaying Wang, Jun Shao, Shihao Xu, Shu Zhang, Weiling Chen, Xavier Tang, Yize Chen, Mengyue Wu, Weizhi Ma, Mengdi Wang, and Tianqiao Chen. 2024. Long Term Memory: The Foundation of AI Self-Evolution. *CoRR* abs/2410.15665 (2024). https://doi.org/10.48550/ARXIV.2410.15665 arXiv:2410.15665

[26] Zhi Jing, Yongye Su, Yikun Han, Bo Yuan, Haiyun Xu, Chunjiang Liu, Kehai Chen, and Min Zhang. 2024. When Large Language Models Meet Vector Databases: A Survey. *CoRR* abs/2402.01763 (2024). https://doi.org/10.48550/ARXIV.2402.01763 arXiv:2402.01763

[27] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woon-Hak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proc. VLDB Endow.* 13, 1 (2019), 57–70. https://doi.org/10.14778/3357377.3357382

[28] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. https://doi.org/10.1109/TPAMI.2010.57

[29] lancedb. 2025. LanceDB: The Multimodal AI Lakehouse. https://github.com/lancedb/lancedb. Accessed: 2025-07-15.

[30] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 225–236. https://doi.org/10.1145/3510003.3510093

[31] Yingfan Liu, Jiangtao Cui, Zi Huang, Hui Li, and Heng Tao Shen. 2014. SK-LSH: an efficient index structure for approximate nearest neighbor search. *Proc. VLDB Endow.* 7, 9 (May 2014), 745–756. https://doi.org/10.14778/2732939.2732947

[32] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[33] marqo ai. 2025. marqo: Unified embedding generation and search engine. https://github.com/marqo-ai/marqo. Accessed: 2025-07-15.

[34] Milvus. 2025. Integration Test of Milvus. https://github.com/milvus-io/milvus/tree/master/tests/integration. Accessed: 2025-07-15.

[35] milvus-io. 2025. Milvus: High-performance, cloud-native vector database built for scalable vector ANN search. https://github.com/milvus-io/milvus. Accessed: 2025-07-15.

[36] MongoDB. 2025. MongoDB Atlas Vector Search. https://www.mongodb.com/products/platform/atlas-vector-search. Accessed: 2025-07-15.

[37] Mozilla. 2025. grcov: Rust tool to collect and aggregate code coverage data for multiple source files. https://github.com/mozilla/grcov Version 0.10.0, Accessed: 2025-07-15.

[38] MyScale. 2023. MyScale Vector Database Benchmark. https://myscale.github.io/benchmark/. Accessed: 2025-07-15.

[39] neuml. 2025. txtai: All-in-one open-source AI framework for semantic search, LLM orchestration and language model workflows. https://github.com/neuml/txtai. Accessed: 2025-07-15.

[40] nmslib. 2025. Hnswlib: Header-only C++/python library for fast approximate nearest neighbors. https://github.com/nmslib/hnswlib. Accessed: 2025-07-15.

[41] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *VLDB J.* 33, 5 (2024), 1591–1615. https://doi.org/10.1007/S00778-024-00864-X

[42] pgvector. 2025. pgvector: Open-source vector similarity search for Postgres. https://github.com/pgvector/pgvector. Accessed: 2025-07-15.

[43] Pinecone. 2025. Pinecone: The vector database to build knowledgeable AI. https://www.pinecone.io/. Accessed: 2025-07-15.

[44] PostgreSQL. 2025. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/. Accessed: 2025-07-15.

[45] Qdrant. 2024. Vector Database Benchmarks. https://qdrant.tech/benchmarks/. Accessed: 2025-07-15.

[46] Qdrant. 2025. Qdrant: High-performance, massive-scale vector database and vector search engine for the next generation of AI. https://github.com/qdrant/qdrant. Accessed: 2025-07-15.

[47] Qiniu. 2023. goc: A Comprehensive Coverage Testing System for The Go Programming Language. https://github.com/qiniu/goc Version 1.4.5, Accessed: 2025-07-15.

[48] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1140–1152. https://doi.org/10.1145/3368089.3409710

[49] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020.* USENIX Association, 667–682. https://www.usenix.org/conference/osdi20/presentation/rigger

[50] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2022. SQLsmith: A random SQL query generator. https://github.com/anse1/sqlsmith. https://github.com/anse1/sqlsmith Version 1.4.

[51] spotify. 2024. annoy. https://github.com/spotify/annoy. Accessed: 2025-07-15.

[52] sqlite. 2025. sqlite: Official Git mirror of the SQLite source tree. https://github.com/sqlite/sqlite. Accessed: 2025-07-15.

[53] Toni Taipalus. 2024. Vector database management systems: Fundamental concepts, use-cases, and current challenges. *Cogn. Syst. Res.* 85 (2024), 101216. https://doi.org/10.1016/J.COGSYS.2024.101216

[54] unum cloud. 2025. USearch. https://github.com/unum-cloud/usearch. Accessed: 2025-07-15.

[55] vespa engine. 2025. Vespa. https://github.com/vespa-engine/vespa. Accessed: 2025-07-15.

[56] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2614–2627. https://doi.org/10.1145/3448016.3457550

[57] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021.* IEEE, 328–337. https://doi.org/10.1109/ICSE-SEIP52600.2021.00042

[58] Shenao Wang, Yanjie Zhao, Yinglin Xie, Zhao Liu, Xinyi Hou, Quanchen Zou, and Haoyu Wang. 2025. Towards Reliable Vector Database Management Systems: A Software Testing Roadmap for 2030. *CoRR* abs/2502.20812 (2025). https://doi.org/10.48550/ARXIV.2502.20812 arXiv:2502.20812

[59] Weaviate. 2024. Uncontrolled concurrency in batch delete may crash server. https://github.com/weaviate/weaviate/issues/5093. Accessed: 2025-07-15.

[60] Weaviate. 2025. Unit Test of Weaviate. https://github.com/weaviate/weaviate/tree/main/test. Accessed: 2025-07-15.

[61] Weaviate. 2025. Weaviate: Open-source vector database that stores both objects and vectors. https://github.com/weaviate/weaviate. Accessed: 2025-07-15.

[62] Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. 2022. Unicorn: detect runtime errors in time-series databases with hybrid input synthesis. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 251–262. https://doi.org/10.1145/3533767.3534364

[63] Xingrui Xie, Han Liu, Wenzhe Hou, and Hongbin Huang. 2023. A Brief Survey of Vector Databases. In *2023 9th International Conference on Big Data and Information Analytics (BigDIA)*. 364–371. https://doi.org/10.1109/BigDIA60676.2023.10429609

[64] Yinglin Xie, Xinyi Hou, Yanjie Zhao, Shenao Wang, Kai Chen, and Haoyu Wang. 2025. Toward Understanding Bugs in Vector Database Management Systems. *CoRR* abs/2506.02617 (2025). https://doi.org/10.48550/ARXIV.2506.02617 arXiv:2506.02617

[65] Hanxiang Xu, Wei Ma, Ting Zhou, Yanjie Zhao, Kai Chen, Qiang Hu, Yang Liu, and Haoyu Wang. 2025. CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025 - Companion Proceedings, Ottawa, ON, Canada, April 27 - May 3, 2025.* IEEE, 243–254. https://doi.org/10.1109/ICSE-COMPANION66252.2025.00079

[66] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2241–2253. https://doi.org/10.1145/3318464.3386131

[67] Yupeng Yang, Yongheng Chen, Rui Zhong, Jizhou Chen, and Wenke Lee. 2024. Towards Generic Database Management System Fuzzing. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/yang-yupeng

[68] Qiang Zhang, Yuheng Shen, Jianzhong Liu, Yiru Xu, Heyuan Shi, Yu Jiang, and Wanli Chang. 2024. ECG: Augmenting Embedded Operating System Fuzzing via LLM-Based Corpus Generation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 43, 11 (2024), 4238–4249. https://doi.org/10.1109/TCAD.2024.3447220

[69] Yunan Zhang, Shige Liu, and Jianguo Wang. 2024. Are There Fundamental Limitations in Supporting Vector Data Management in Relational Databases? A Case Study of PostgreSQL. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3640–3653. https://doi.org/10.1109/ICDE60146.2024.00280

[70] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2024. Chat2Data: An Interactive Data Analysis System with RAG, Vector Databases and LLMs. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4481–4484. https://doi.org/10.14778/3685800.3685905

[71] zilliztech. 2025. VectorDBBench: A Vector Database Benchmark Tool. https://zilliz.com/benchmark. Accessed: 2025-07-15.