

# Toward Understanding Bugs in Vector Database Management Systems

Yinglin Xie\*, Xinyi Hou\*, Yanjie Zhao, Shenao Wang, Kai Chen<sup>†</sup> and Haoyu Wang<sup>†</sup>  
Huazhong University of Science and Technology, Wuhan, China  
{xieyinglin, xinyihou, yanjie\_zhao, shenawang, kchen, haoyuwang}@hust.edu.cn

**Abstract**—Vector database management systems (VDBMSs) play a crucial role in facilitating semantic similarity searches over high-dimensional embeddings from diverse data sources. While VDBMSs are widely used in applications such as recommendation, retrieval-augmented generation (RAG), and multi-modal search, their reliability remains underexplored. Traditional database reliability models cannot be directly applied to VDBMSs because of fundamental differences in data representation, query mechanisms, and system architecture. To address this gap, we present the first large-scale empirical study of software defects in VDBMSs. We manually analyzed 1,671 bug-fix pull requests from 15 widely used open-source VDBMSs and developed a comprehensive taxonomy of bugs based on symptoms, root causes, and developer fix strategies. Our study identifies five categories of bug symptoms, with more than half manifesting as functional failures. We further reveal 31 recurring fault patterns and highlight failure modes unique to vector search systems. In addition, we summarize 12 common fix strategies, whose distribution underscores the critical importance of correct program logic. These findings provide actionable insights into VDBMS reliability challenges and offer guidance for building more robust future systems.

## I. INTRODUCTION

Vector database management systems (VDBMSs) have emerged as a cornerstone of modern AI infrastructure, enabling the efficient management of high-dimensional embeddings and facilitating the semantic search across diverse data modalities. These systems underpin a wide range of transformative applications, including intelligent recommendation engines and cross-modal retrieval tasks such as using text queries to locate relevant video content [47], [48], [9].

In the context of large language model (LLM) ecosystems, VDBMS plays a pivotal role in retrieval-augmented generation (RAG). Toolkits such as LangChain [17] and LlamaIndex [19] integrate VDBMSs to ground LLM outputs with dynamically updated knowledge, thereby mitigating hallucination risks and improving factual consistency. Industry-leading systems like Vespa [34], Milvus [37], and Faiss [10] have been carefully engineered to support heterogeneous hardware platforms, ranging from GPU clusters to neuromorphic processors. As demands for context-aware, low-latency LLM applications continue to grow, vector search capabilities are becoming a core priority within enterprise LLM stacks [20], [42], [41].

As VDBMSs are increasingly used across a wide range of LLM applications, reliability is critically important. Software defects in these systems can result in serious consequences, such as corrupted vector indexes, inaccurate similarity computations, or even cascading failures in downstream LLM applications. For instance, inconsistencies in approximate nearest neighbor (ANN) algorithms can compromise recommendation accuracy in e-commerce platforms [40], while memory leaks in high-throughput environments may trigger system-wide instability in real-time scenarios [43]. These issues not only diminish user trust but also impose substantial operational overhead in terms of debugging, mitigation, and recovery.

Traditional database management systems (DBMSs) have been extensively studied with well-established models for transaction processing and SQL optimization. However, these approaches are not directly applicable to VDBMSs due to fundamental architectural differences. VDBMSs differ from conventional systems in several key aspects. First, they operate on unstructured data by transforming it into high-dimensional numerical vectors, rather than storing structured records with fixed schemas. Second, their query mechanisms rely on probabilistic similarity search algorithms, such as ANN, rather than deterministic operations defined by SQL. Third, the semantic relationships among vectors give rise to dynamic and evolving data topologies that are incompatible with static schema constraints. These features challenge the assumptions of traditional reliability assessment techniques, making them insufficient for evaluating VDBMS behavior.

Despite their growing importance, there remains a significant gap in our understanding of the reliability risks and software defects specific to VDBMSs. To address this gap, we present the first large-scale empirical study of software defects in VDBMSs. We analyzed 1,671 bug-fix pull requests (PRs) from 15 widely used open-source VDBMSs hosted on GitHub. Each PR was carefully examined and categorized according to observable symptoms and root causes. Based on this analysis, we identified 5 symptom categories and 31 fault patterns, as well as 12 classes of fix strategies adopted by developers. This empirical approach allowed us to capture real-world failure scenarios, uncover recurring bug patterns and failure modes, and summarize typical repair strategies, ultimately offering actionable insights into the unique reliability challenges faced by VDBMSs. Our artifacts are publicly available at <https://figshare.com/s/00034c934612a54b8620>.

\*Yinglin Xie and Xinyi Hou contributed equally to this work.

<sup>†</sup>The corresponding authors are Kai Chen (kchen@hust.edu.cn) and Haoyu Wang (haoyuwang@hust.edu.cn).

In summary, we make the following contributions:

- **Empirical Analysis:** We conduct the first large-scale empirical study of software defects in VDBMSs, analyzing 1,671 bug-fix PRs from 15 open-source VDBMSs.
- **Taxonomy and Patterns:** We develop a comprehensive taxonomy of bugs based on symptoms, root causes, and fix strategies, and identify 5 symptom categories and 31 recurring fault patterns unique to VDBMSs.
- **Actionable Insights:** We uncover common failure modes and repair strategies, providing 10 actionable insights that inform both the development and testing of more reliable and robust VDBMSs.

## II. BACKGROUND AND RELATED WORK

### A. Vector Database Management Systems (VDBMSs)

VDBMSs manage and retrieve high-dimensional vectors derived from unstructured data such as text, images, and audio. As shown in Figure 1, they commonly adopt a client-server architecture: the server handles vector storage, indexing, and similarity search, while the client exposes interfaces for querying, data insertion, and metadata operations.

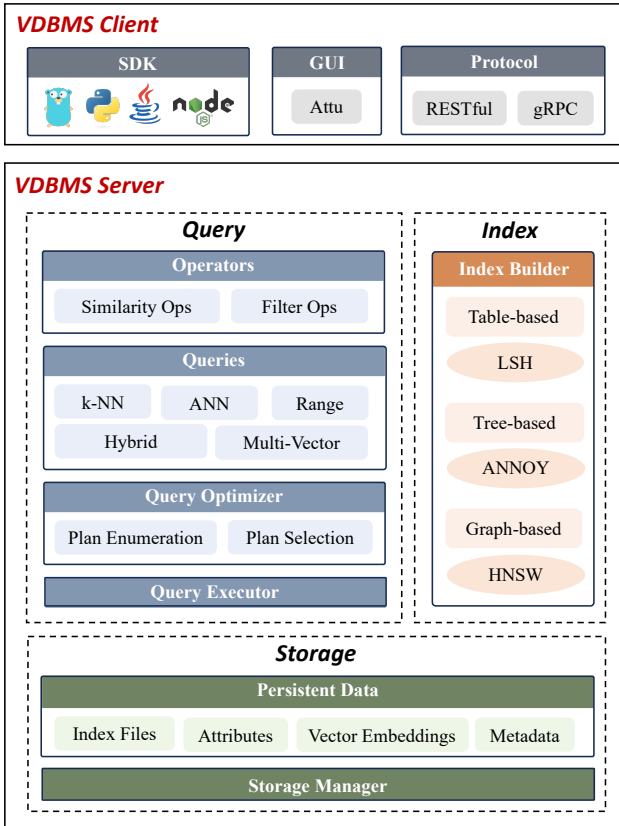


Fig. 1: Architecture of VDBMSs.

1) **VDBMS Server:** User requests, transmitted via the client, flow through three layers on the server: query, index, and storage. The query layer parses and plans the request, the index layer performs efficient vector retrieval, and the storage layer

provides access to the underlying data, which is then returned to the query layer for final processing and response.

**Query.** This layer includes four main components: operators, queries, the query optimizer, and the query executor. Operators perform tasks like similarity computation and result filtering, supporting query types such as k-nearest neighbor (k-NN) [6], approximate search [2], and range queries. For advanced use cases, it supports hybrid queries [46], which combine vector similarity with attribute filters, and multi-vector queries, where multiple vectors and aggregated scores represent a single entity. The query optimizer selects efficient execution strategies, which are then carried out by the query executor through interaction with other VDBMS layers.

**Index.** The index layer is designed to accelerate similarity search over large collections of high-dimensional vectors by building specialized data structures that efficiently narrow down candidate results during query execution. Various indexing strategies can be employed, including table-based methods such as locality-sensitive hashing (LSH) [24], tree-based approaches like random projection trees [8], and graph-based structures such as HNSW [21].

**Storage.** This layer is responsible for managing all persistent data in the system, including vector embeddings, structured attributes, index files, and metadata. Its primary function is to ensure reliable storage and retrieval of this information while supporting efficient updates and access for query processing.

2) **VDBMS Client:** The client acts as the interface between users or applications and the VDBMS server. Most systems provide multi-language client SDKs, commonly written in Python, Java, Go, and JavaScript, which facilitate integration across diverse platforms. Some VDBMSs also offer graphical user interfaces to enhance client-side interaction; for example, Milvus provides a dedicated visual client called Attu [3]. Clients typically support two main communication protocols: REST APIs for lightweight metadata and control operations, and gRPC [12] for high-performance data transfer, particularly when handling large batches of vectors.

VDBMSs are essential for LLM applications using high-dimensional embeddings, providing core infrastructure for storing, searching, and managing vector data. They enable similarity-based retrieval for tasks like recommendation, anomaly detection, and semantic search, and also support RAG by supplying external knowledge to improve LLM responses.

### B. Reliability of VDBMSs

The reliability of traditional DBMSs has long been a central topic of research. Liu *et al.* [18] conducted a comprehensive analysis of 423 database access bugs across seven large-scale Java applications, while Cui *et al.* [7] examined 140 transaction-related bugs in six widely used database systems. Rigger *et al.* [29] further introduced the Non-Optimizing Reference Engine Construction (NoREC) method to detect optimization bugs in query engines. These studies collectively reveal that DBMSs are susceptible to reliability issues stemming from diverse sources, including server-client interfacing, transaction handling, and query optimization.

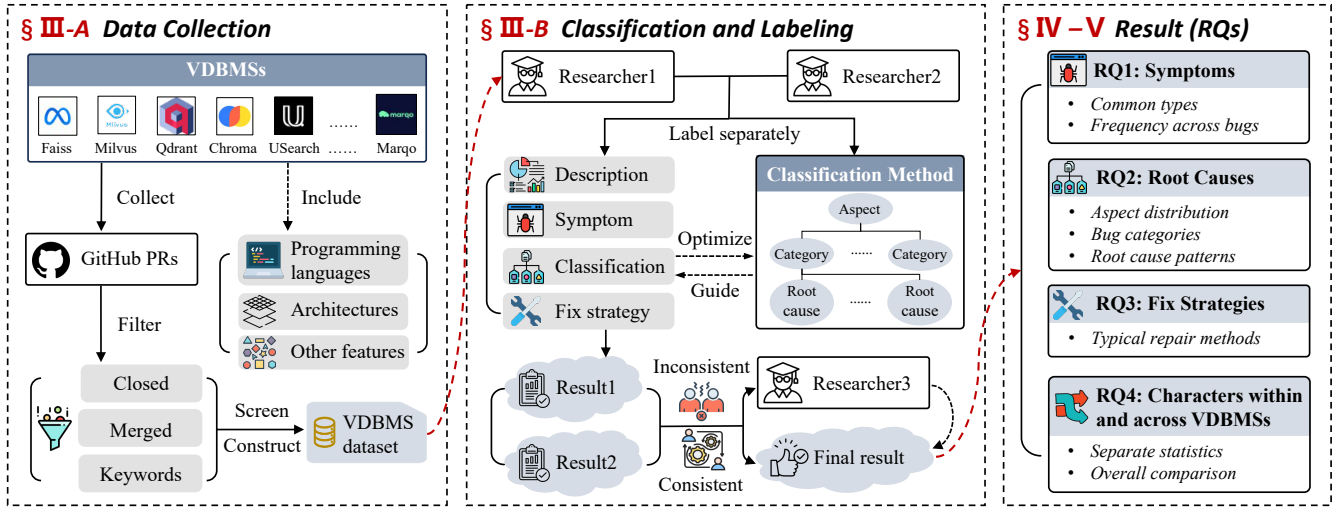


Fig. 2: Overview of the methodology for investigating bug characteristics in VDBMSs.

In contrast, VDBMSs introduce a new set of reliability challenges due to their fundamental reliance on high-dimensional and semantically rich embeddings. Unlike traditional databases that operate over structured and ordinal data, VDBMSs must process unstructured data where query semantics are often ill-defined [31]. Moreover, the computational cost of vector comparisons is significantly higher, and building efficient yet consistent index structures for such data remains difficult [38], [45], [13]. The complexity further increases with hybrid queries that integrate vector similarity with attribute-based filtering, complicating both query planning and execution. Although recent work has introduced various techniques to improve system efficiency, such as quantization-based compression [11], [15], [23] and learned partitioning methods [39], [24], [30], these approaches largely focus on performance optimization. They offer limited solutions to deeper reliability concerns, such as incorrect results, inconsistent index states, or unpredictable behavior under dynamic workloads. Given the growing use of VDBMSs in critical LLM applications, addressing their reliability challenges is both urgent and essential. This motivates our investigation into how these systems behave in practice and what factors underlie their failures.

### III. METHODOLOGY

As shown in Figure 2, we designed a multi-step methodology involving data collection, classification, and labeling to systematically investigate bug characteristics in VDBMSs.

#### A. Data Collection

1) *VDBMS Selection*: We selected 15 VDBMSs as our research subjects, as listed in Table I, including Faiss [10], Milvus [37], and Weaviate [44]. The selection followed several criteria: we prioritized widely adopted and actively maintained systems, using GitHub stars as a proxy for popularity, and included only those with over 14,000 stars as of November 29, 2024 to ensure broad usage and representativeness. Horizontally, the selected VDBMSs span a range of implementation

languages: Faiss [10], Hnswlib [21], and Annoy [1] are in C++; Qdrant [27] and pgvector.rs [26] in Rust; Chroma [5], txtai [32], Deep Lake [14], and Voyager [36] in Python; and Vespa [34] in Java and C++. This diversity allows us to examine how language-level features affect bug characteristics. The systems also vary in architecture and indexing strategies, enabling broader analysis of how design choices impact reliability. Vertically, the selected VDBMSs span a full decade, from 2013 to 2023, capturing the evolution of the field over time. The selection offers a diverse and representative foundation for our empirical analysis.

TABLE I: Overview of Selected VDBMSs and Filtered PRs.

VDBMS	#Forks	#Stars	Language	#Closed	#Filtered
Faiss [10]	3.8k	34.3k	C++	1,260	13
Milvus [37]	3.1k	34.0k	Go, C++	23,392	412
Qdrant [27]	1.6k	23.0k	Rust	3,939	35
Chroma [5]	1.6k	19.2k	Python	1,892	126
Annoy [1]	1.2k	13.7k	C++	263	2
Weaviate [44]	0.9k	13.0k	Go	4,013	109
txtai [32]	0.7k	10.7k	Python	38	8
Deep Lake [14]	0.6k	8.5k	Python	2,478	167
Vespa [34]	0.6k	6.1k	Java, C++	31,969	412
LanceDB [16]	0.4k	6.1k	Rust, Python	1,074	89
Marqo [22]	0.2k	4.8k	Python	771	119
Hnswlib [21]	0.7k	4.6k	C++	197	8
Usearch [33]	0.2k	2.6k	C++	364	39
pgvector.rs [26]	0.1k	2.0k	Rust	341	115
Voyager [36]	0.1k	1.4k	Python	63	17
<b>Total</b>	<b>15.8k</b>	<b>184.0k</b>	—	<b>72,054</b>	<b>1,671</b>

2) *PR Collection*: We then sourced PRs from GitHub and applied a set of filtering criteria. We included only PRs that were **closed** and merged into the **main development branch**, which could be either **main** or **master**, depending on the project. To ensure relevance, we required that the PR title or labels contain **at least one** of the following keywords: *bug*, *error*, *fail*, *failure*, *fault*, *flaw*, *mistake*, *issue*, *problem*, *question*, *matter*, *trouble*, *crash*, *exception*, *fix*, *repair*, *defect*, *debug*, or *correct*. Milvus had an exceptionally large number

of closed PRs (23,392). Due to the high cost of manual inspection, we selected a subset in reverse chronological order, covering August to December 2024, yielding 412 PRs. Vespa had a similar case, with 31,969 closed PRs. We adopted the same strategy and selected the most recent 412 PRs from December 2024. Following the above steps, we obtained a comprehensive and representative dataset for our analysis. Detailed statistics, including the number of filtered PRs for each VDBMS, are provided in [Table I](#).

### B. Classification and Labeling

1) *Taxonomy Construction*: We collected 1,671 merged PRs from 15 widely used VDBMS projects. To enable systematic analysis, we labeled each PR using six key attributes: *description*, *symptom*, *affected component*, *root cause*, *fix strategy*, and *bug type*. These dimensions capture both observable symptoms and underlying causes of VDBMS bugs. Among them, the **symptom** and **fix strategy** dimensions were adapted from prior studies on defect characterization [28], [4]. Symptoms are grouped into five categories ([Figure 3](#)), while fix strategies are summarized into 12 recurring patterns ([Table II](#)), reflecting common failure modes and repair actions. For the **root cause** dimension, we developed a hierarchical taxonomy grounded in the architecture of VDBMSs. Building on the system decomposition in [25], which separates a VDBMS into query processor and storage manager, we define five high-level categories: **Query**, **Storage**, **Index**, **Parsing & Interaction**, and **Configuration**. Each category is further refined into subcategories and specific root causes.

2) *Annotation and Refinement*: We adopted a multi-stage collaborative procedure to annotate all 1,671 PRs using the taxonomy described above. The process began with a pilot study, in which one researcher labeled a random sample of 150 PRs. During this phase, the root cause taxonomy was iteratively refined based on observed patterns, resulting in 20 initial root causes. To assess annotation consistency, a second researcher independently annotated the same 150 PRs using the refined taxonomy. Inter-rater agreement, measured by Cohen’s Kappa [35], exceeded 0.95, indicating near-perfect consistency. Disagreements were resolved by a third researcher serving as an arbitrator. As annotation continued on the remaining PRs, new bug types emerged. These were either mapped to existing categories or temporarily assigned to a “pending” category. Through iterative discussion, the taxonomy was expanded to 41 root causes, and later consolidated into a final set of 31 to reduce redundancy and ambiguity. Using the finalized taxonomy, symptom categories, and fix strategies, we re-annotated the dataset for consistency. Non-bug PRs were excluded, and those involving multiple distinct bugs were split into separate entries. After filtering, we obtained 1,463 confirmed bugs. The final taxonomy of root cause includes 5 top-level categories, 14 subcategories, and 31 leaf-level root causes. This high-quality, consistently labeled dataset serves as the basis for our subsequent analysis.

## IV. TAXONOMY AND PATTERNS

This section reports our empirical analysis of 1,463 real-world bug-fixing PRs in VDBMSs, examining symptoms, root causes, and fix strategies according to the following research questions (RQs).

**RQ1 What are the typical symptoms exhibited by bugs in VDBMSs, and how are they distributed?** We identify common bug symptoms in VDBMSs, analyze their frequency and diversity, and examine how they reflect different severity levels across categories.

**RQ2 What are the most bug-prone components in VDBMSs, and what are the key root causes?** We analyze the distribution of bugs across VDBMS components, categorize their types, and identify all root causes.

**RQ3 What fix strategies are commonly adopted for different types of bugs in VDBMSs?** We analyze how developers fix bugs by identifying common strategies and their association with different bug types.

### A. RQ1: Symptoms.

Based on our observations, the symptoms were grouped into five major categories as shown in [Figure 3](#). These categories capture the diverse ways in which bugs can manifest, ranging from critical disruptions such as system crashes to less severe but still impactful issues like misleading log messages or incorrect documentation.

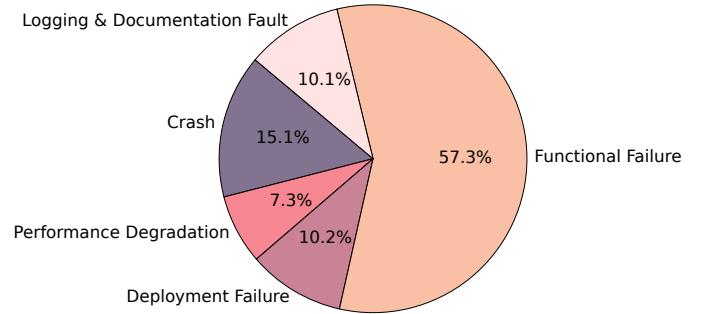


Fig. 3: Distribution of symptoms.

1) *Crash*: Crash is the most severe symptom observed in VDBMSs, as it directly causes service termination and loss of availability. Crash-related bugs account for 15.1% of all symptoms in our dataset, with the highest proportions found in Qdrant (40.7%) and txtai (40%), indicating these systems are particularly prone to critical failures. In VDBMSs, crashes typically manifest as abrupt process terminations, segmentation faults, or unrecoverable runtime exceptions. These symptoms are frequently observed in core components such as storage backends, query pipelines, and system integration layers. Compared to traditional DBMSs, VDBMSs are more susceptible to crash bugs due to their reliance on high-performance native libraries (e.g., BLAS, Faiss) and complex vector algorithms, which increase the risk of memory errors and environment-dependent failures. The greater algorithmic complexity and



varied deployment scenarios in VDBMSs also make edge-case crashes harder to test, reproduce, and diagnose.

**Insight 1:** *To reduce crash risks, VDBMS developers should test edge cases, abnormal inputs (e.g., NaN or out-of-bound vectors), and concurrency. Components written in low-level languages like C or C++ require particular attention due to their vulnerability to memory errors.*

2) **Functional Failure:** This is the most common symptom in our dataset, accounting for 57.3% of all cases. These bugs cause incorrect behaviors such as inaccurate query results, missing search matches, inconsistent indexing outputs, or invalid status responses. They often stem from logical inconsistencies or algorithmic flaws in query processing and indexing, which are highly complex and sensitive to semantic similarity definitions. For example, a functional bug in Faiss was triggered by misconfigured IVF search parameters, producing “-1” placeholder IDs in top-k results. Since vector queries rely on approximate similarity and complex indexes like quantized or graph-based ANN structures, result correctness is especially vulnerable to logic bugs in similarity projection or quantization. Fixing these issues typically requires major changes to logic or data flow.

**Insight 2:** *Robust functional testing in VDBMSs should target both typical and edge-case query scenarios, with particular emphasis on validating the logic and parameter handling of approximate search and indexing algorithms.*

3) **Performance Degradation:** Performance bugs comprise 7.3% of all cases, typically manifesting as excessive resource use, slow response times, or throughput bottlenecks. These often result from inefficient indexes, misconfigured parameters, or suboptimal query execution. For example, in Vespa (PR #31,664), a linear-weight ranking profile with a high hits value caused nodes to process too many documents, significantly slowing queries. Unlike crashes or functional failures, performance degradation is usually gradual and workload-dependent.

4) **Deployment Failure:** Deployment failures in VDBMSs typically occur during build, installation, or startup phases. Common symptoms include missing dependencies, incompatible compiler versions, broken build scripts, and misconfigured environment variables. These issues account for 10.2% of cases in our dataset and are often amplified by reliance on specialized native libraries (e.g., BLAS, Faiss, SIMD) and integration with containerization frameworks. Unlike runtime failures, deployment bugs can prevent system launch, posing a major barrier to adoption, especially for new users and in CI/CD environments. Robust and portable deployment processes are essential for improving VDBMS reliability.

**Insight 3:** *VDBMS developers should ensure portability and environment isolation by using containerized builds,*

*automated installation scripts, and pre-check tools to detect missing dependencies or incompatible settings.*

5) **Logging & Documentation Fault:** This category accounts for 10.1% of all recorded symptoms and includes missing, misleading, or outdated logs and documentation. Faulty logging may obscure root causes by omitting key runtime information, while poor documentation, such as incorrect usage examples or missing configuration guidance, hinders adoption and usage. For VDBMSs with complex queries and configurations, clear logs and up-to-date documentation are essential for troubleshooting and user onboarding.

#### B. RQ2: Root Causes.

We construct a three-level taxonomy to classify bugs in VDBMSs, as shown in Figure 4. The percentages indicate the distribution of bugs at each level of the taxonomy. This taxonomy includes five categories: **Query** (711 bugs), **Storage** (182 bugs), **Index** (58 bugs), **Parsing & Interaction** (125 bugs), and **Configuration** (387 bugs).

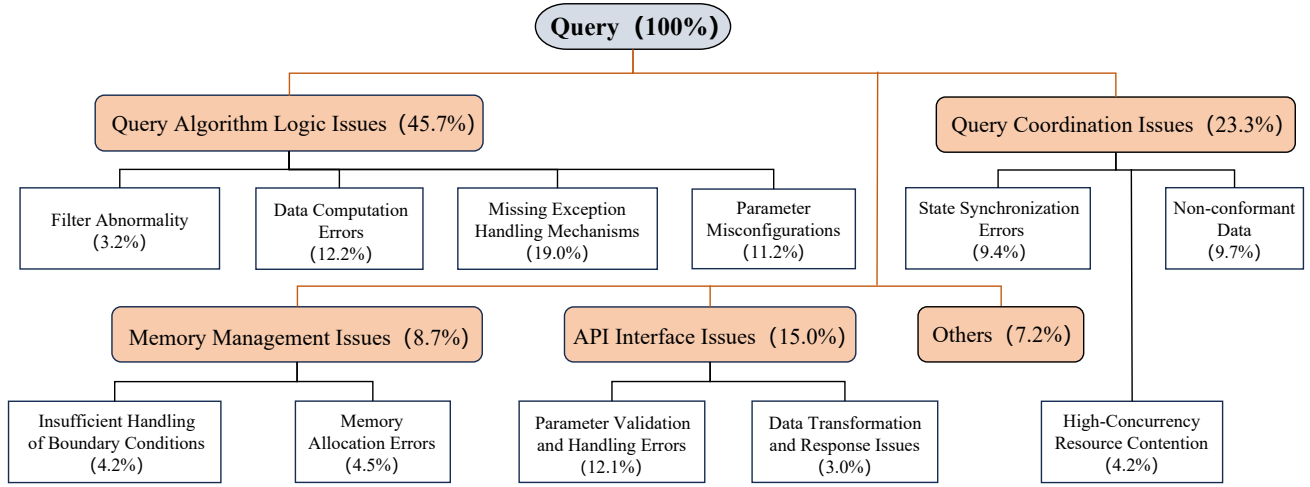
1) **Query:** Query is the most bug-prone component, comprising 48.6% of all cases. **Query algorithm logic issues** (45.7%) reflect challenges in ensuring correct query execution. **Query coordination issues** (23.3%) involve failures in managing concurrent execution and synchronization. **API interface issues** (15.0%) result from miscommunication between external APIs and internal handlers, while **memory management issues** (8.7%) stem from improper memory allocation or release during query processing.

**Query Algorithm Logic Issues.** Incomplete or faulty logic in query algorithms, particularly when edge cases or unexpected inputs are overlooked, commonly leads to these bugs. **Missing exception handling mechanisms**, such as failing to check for empty or null fields, which can lead to crashes. **Parameter misconfigurations**, using illegal or ill-typed arguments, often result in subtle inaccuracies. **Data computation errors**, like miscalculating similarity scores or buffer lengths, directly affect query reliability. **Filter abnormality** refers to incomplete or incorrect filtering, causing improper inclusion or exclusion of data. These issues reflect the complexity of vector query logic and the challenge of anticipating all corner cases.

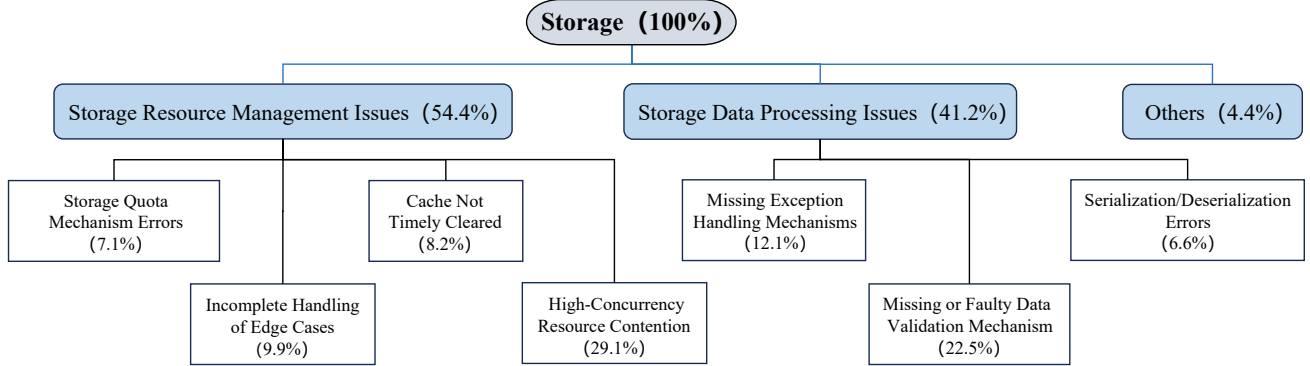
**Insight 4:** *Ensuring the reliability of query algorithms in VDBMS requires thorough boundary testing and exception handling to address edge cases and unexpected parameter configurations.*

#### Query Coordination Issues.

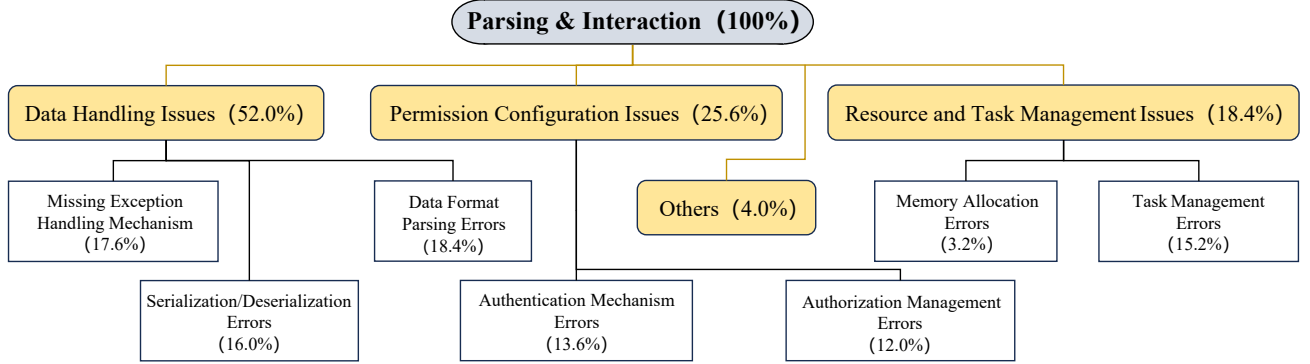
Coordinating query stages or handling concurrent sessions can introduce subtle, hard-to-detect failures. One cause is **non-conformant data**, where mismatched formats or inconsistent encoding disrupt execution. For example, in Weaviate (PR #1,421), omitting the `_additional` field led to incomplete downstream outputs. **State synchronization errors** occur when components use outdated or inconsistent states, such as failing to terminate after a shutdown signal. **High-concurrency**



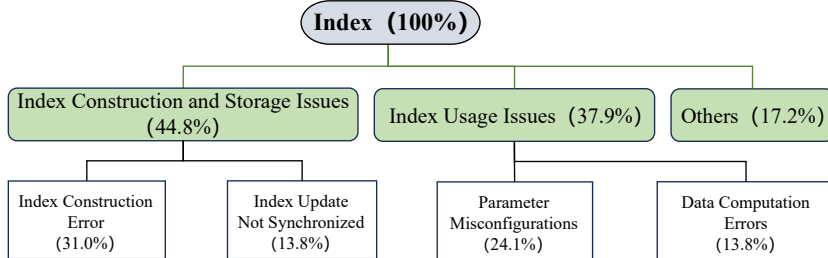
(a) Classification result for bugs in *Query*.



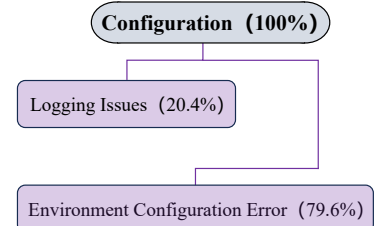
(b) Classification result for bugs in *Storage*.



(c) Classification result for bugs in *Parsing & Interaction*.



(d) Classification result for *Index*.



(e) Classification result for *Configuration*.

Fig. 4: Overview and example of the bug classification method.

**resource contention** arises in parallel workloads, where poor thread or resource management results in race conditions, deadlocks, or degraded performance. Achieving reliable coordination requires consistent data interfaces, accurate state tracking, and careful concurrency control, which is particularly challenging in asynchronous and distributed VDBMS environments.

#### API Interface Issues.

System instability can arise from surface-level interfaces. **Parameter validation and handling errors** occur when servers fail to verify input completeness, type, format, or range, or when dependencies and defaults are not properly enforced, leading to unexpected query behaviors. Another cause is **data transformation and response issues**, often due to serialization or deserialization errors. For example, in Vespa (PR #29,449), a misconfigured Jackson ObjectMapper resulted in malformed JSON during request parsing and response generation. VDBMSs commonly process complex, nested queries with embeddings and hybrid parameters, where minor inconsistencies in input or response formatting can propagate system-wide. Robustness requires not just basic validation, but schema-aware design to ensure semantic consistency throughout the query lifecycle.

**Memory Management Issues.** High-dimensional vector computations introduce irregular, data-dependent memory access patterns that often lead to subtle memory errors. A common issue is **insufficient handling of boundary conditions**, where the system fails to guard against edge cases such as buffer overflows or null pointer dereferences. For example, complex expressions evaluated over input vectors without proper checks can cause out-of-bounds access and crashes. Memory bugs also stem from **incorrect allocation strategies**, including inaccurate memory estimation or poorly timed release. Premature deallocation can trigger use-after-free errors, while delayed release may cause memory leaks or bloat. In VDBMSs, memory usage is hard to predict due to factors like vector dimensionality, batch size, and filter complexity. To meet low-latency demands, systems often reuse buffers aggressively, increasing the risk of lifetime violations. These challenges are amplified by the large, high-dimensional nature of vector data, which is typically allocated in batches.

**Insight 5:** Memory allocation and usage in VDBMSs must account for the inherent complexity of vector data and various boundary conditions. The high dimensionality of vectors, coupled with data-dependent access patterns, demands precise memory estimation to avoid issues.

2) **Storage:** As shown in Figure 4b, a total of 182 bugs are associated with the storage layer. These fall into two main categories: **Storage resource management issues** (54.2%), which concern the efficient utilization of storage resources and their impact on overall system performance; and **Storage data processing issues** (41.2%), which arise during data transformation, migration, or maintenance operations.

**Storage Data Processing Issues.** VDBMSs are prone to data handling faults during storage operations, especially when serialization logic or validation mechanisms are poorly designed. **Serialization/Deserialization errors** arise from mismatched formats, missing fields, or incompatible schema versions, leading to failures in persisting or restoring data. In some cases, valid data is rejected or causes crashes during deserialization. **Missing or faulty validation** allows incorrect or corrupted records to enter the database, affecting downstream operations. **Missing exception handling** can also hinder recovery from abnormal events. For example, in Milvus (PR #36,149), the system failed to restore data after a crash, while in pgvector (PR #242), a power loss left the database unrecoverable due to insufficient fault-tolerance logic. Unlike traditional DBMSs with well-defined schemas, VDBMSs often store high-dimensional data and index metadata in custom formats. This increases the risk of serialization mismatches, validation gaps, and deserialization errors, highlighting the need for specialized testing techniques.

**Insight 6:** Testing VDBMSs requires going beyond conventional strategies by incorporating various data schema fault injections to ensure robustness across heterogeneous and evolving data formats.

**Storage Resource Management Issues.** Efficient storage resource utilization is critical in VDBMSs. In addition to the previously mentioned causes in § IV-B1, such as **insufficient handling of boundary conditions** and **high-concurrency resource contention**, two other root causes are particularly relevant in this context. The first is **storage quota mechanism errors**, where flawed strategies for allocating storage space, such as incorrect sharding logic, result in data being stored or retrieved incorrectly. The second is **cache not timely cleared**, where improper cache management, such as the failure to release memory after use, can lead to performance degradation or even system instability under sustained workloads. Existing VDBMSs have adopted several practical strategies to optimize storage resource management, such as vector compression methods. However, our results show that bugs related to this category still account for 54.4% of all storage-related issues.

**Insight 7:** Stress testing VDBMS should focus on high-concurrency workloads, large-scale data ingestion, and edge cases like irregular sharding to ensure robust storage resource management under real-world conditions.

3) **Parsing & Interaction:** According to Figure 4c, 125 bugs are attributed to the parsing and interaction layer, stemming from client input parsing, result interpretation, and user-system interaction. **Data handling issues** (52%), including failures in input parsing, format interpretation, and memory management during request and response processing. **Permission configuration issues** (25.6%) involve authentication and access control, where misconfigurations result in failed or insecure interactions. **Resource and task management issues** (18.4%)

concern incorrect memory allocation, poor task coordination, and errors in asynchronous execution.

**Data Handling Issues.** Data pipelines in VDBMSs are often fragile, breaking during request parsing or response reconstruction. A key root cause is *missing exception handling*, where the system fails to detect or recover from malformed inputs, leading to crashes or undefined behavior. Another common issue is *data format parsing errors*, caused by deviations from expected input structures, resulting in parsing failures or silent corruption. Response deserialization errors also occur when fields are missing, mismatched, or incompatible with the schema expected by the client. These failures highlight the need for defensive parsing strategies, such as schema validation, input fuzzing, and fallback mechanisms, to improve robustness against faulty inputs.

**Permission Configuration Issues.** These issues mainly stem from *authentication mechanism errors* or *authorization management errors*. Authentication failures involve flawed user verification, allowing unauthorized access or blocking legitimate users. Authorization errors arise from misconfigured access rules, which can block valid actions or expose sensitive operations. Such problems are critical in VDBMSs, which need fine-grained permissions for tasks like collection access, embedding uploads, and index changes. Operating in machine-to-machine environments or integrating with larger platforms, these systems are prone to subtle permission bugs from inconsistent roles, token scopes, or default settings.

**Insight 8:** *Permission bugs in VDBMSs can block legitimate users or expose sensitive operations to unauthorized clients. Regular audits of configuration files and role definitions help detect misconfigurations early and ensure access controls align with intended security policies.*

**Resource and Task Management Issues.** The stability of VDBMSs often relies on careful control of memory usage and asynchronous task execution. *Memory allocation errors* occur when memory is allocated incorrectly, either too little or too much, during parsing or interaction, leading to performance issues or crashes. *Task management errors* result from background operations that are not properly scheduled or monitored, causing inconsistent behavior. For example, in LanceDB (PR #407), a misconfigured timer led to irregular task execution and incorrect results. Robust VDBMSs require strict memory bounds, controlled asynchronous task lifecycles, and integrated observability mechanisms to detect anomalies early. These capabilities help prevent minor issues from propagating into major system failures.

4) *Index:* As shown in Figure 4d, the Index aspect accounts for 58 bugs, representing only 4.0% of the total. These are categorized into two types: *index construction and storage issues*, which occur during index building, persistence, or updates; and *index usage issues*, which arise during query execution using the index. Despite their small number, indexing bugs can significantly affect system behavior. Structural or usage flaws may propagate to other components, resulting in

incorrect queries or failures in data operations. As such, they often underlie issues in the query or storage layers and are critical to VDBMS robustness.

**Index Construction and Storage Issues.** Errors in building and maintaining index structures often lead to stale results or incomplete recall during search. Failures in this category often stem from *index construction error*, where improper initialization, incorrect parameter configurations, or flawed data handling procedures result in incomplete or invalid index structures. Equally influential is *index update not synchronized*, a condition in which updates to the underlying data are not correctly propagated to the index, leading to stale entries and inconsistent search outcomes.

**Insight 9:** *For VDBMS testing, it is essential to validate that index structures are properly initialized, configured, and populated. Tests should also cover data updates and deletions to ensure index entries stay synchronized.*

**Index Usage Issues.** Bugs in this category typically reflect misuses of the index during query execution. Some problems stem from *parameter misconfigurations*, where inappropriate settings for parameters such as search depth or probe count result in degraded accuracy or unnecessary performance overhead. Others involve *data computation errors*, which can cause the system to return incomplete or misleading results. These issues do not compromise the structural integrity of the index but significantly undermine its utility during runtime.

5) *Configuration:* A total of 387 bugs (26.5% of all cases) stem from configuration issues, making it one of the most significant sources of failure. As shown in Figure 4e, a large portion involves *environment configuration errors*, such as mismatches between documentation and actual API behavior, library or runtime incompatibilities, and incorrect dependency setups across deployment platforms. Another cause is *logging-related problems*, where missing, unclear, or misleading log messages obscure root causes and hinder debugging. These bugs are particularly challenging due to their diffuse nature: they often arise not from a single module but from complex interactions among system components, external tools, and user environments. This highlights the need for holistic testing, tighter integration between code and tooling, and better alignment between documentation and actual behavior.

### C. RQ3: Fix Strategies.

We identify 12 representative fix strategies from an analysis of VDBMS bug-fixing commits. Each strategy reflects how developers resolve bugs by targeting configuration, code logic, data handling, or environment settings. Table II summarizes their distribution across four key areas: **Environment**, **Data**, **Program**, and **API**. We further group and analyze these strategies to understand their frequency and use cases.

1) *Environment:* Fix strategies targeting the environment layer account for a substantial portion of all observed cases (239 in total). These are mainly used to resolve misconfigurations, compatibility issues, or incorrect setup procedures.



TABLE II: Fix strategies adopted across four system layers.

Object	Fix Strategy	Count
Environment	Upgrade version	30
	Modifying dependency configurations	34
	Modify environment variable values	41
	Update configuration files or instructions	134
Data	Modify or add data processing logic	40
	Modify data default values or initialization	46
Program	Fix or complete program logic	706
	Modify parameter values	54
	Adjust memory allocation strategy	71
	Add validation and safeguard mechanisms	188
	Remove redundant logic	60
API	Adjust API usage	45
Others	Model modification, Error message update...	14
<b>Total</b>		<b>1,463</b>

The most common strategy is *updating configuration files or instructional materials* (134 cases), which includes correcting default templates, adding missing setup steps, or clarifying deployment guides. Another frequent approach is *modifying environment variable values* (41 cases) to fix path errors, resource visibility issues, or to enable or disable specific runtime behaviors. *Upgrading versions* (30 cases) typically addresses library incompatibilities, outdated dependencies, or deprecation warnings. Finally, *modifying dependency configurations* (34 cases) involves adding missing packages, adjusting version constraints, or removing conflicting dependencies. These changes are usually made through files such as `requirements.txt`, `poetry.lock`, or `Dockerfiles`.

2) *Data*: Fix strategies at the data layer mainly fall into two categories. The first involves *modifying or adding data processing logic* (40 cases), such as correcting preprocessing errors or adding missing normalization steps in ingestion pipelines. The second addresses *modifying data default values or initialization* (46 cases), which often includes adjusting placeholder values, setting appropriate vector dimensions, or ensuring proper initialization of data fields to prevent runtime exceptions or incorrect computations.

3) *Program*: Program-level fix strategies are the most prevalent, addressing 1,079 bugs by modifying core logic to fix semantic errors, complete missing functionality, and handle edge cases. *Fixing or completing program logic* (706 cases), which includes correcting control flow, adding missing conditional branches, or fixing flawed computations. Developers also often *add validation and safeguard mechanisms* (188 cases) to prevent runtime exceptions and enforce correctness, such as checks for null values, vector dimensions, or operation preconditions. Other strategies include *removing redundant logic* (60 cases), often left over from earlier development or refactoring, and *modifying parameter values* (54 cases) to adjust thresholds, batch sizes, or resource limits. In performance-critical paths, *adjusting memory allocation strategies* (71 cases) helps avoid out-of-memory errors and improve efficiency, particularly during large-scale vector operations.

**Insight 10:** Many program-level bugs in VDBMS stem from logic and edge-case errors that are hard to detect with standard unit tests.

4) *API*: Bugs related to incorrect or outdated API usage are addressed through a single but important strategy: *adjusting API usage* (45 cases). These fixes typically involve modifying invocation methods, correcting argument formats, or reordering dependent function calls. Such issues are often triggered by breaking changes in upstream libraries or misuse of internal interfaces within the VDBMS engine. To mitigate such issues, it is essential to strengthen interface contracts and improve change management practices. Effective measures include enforcing version checks, monitoring deprecation warnings, and isolating internal APIs from external exposure, especially in fast-evolving codebases.

## V. CORRELATION AND CROSS-SYSTEM ANALYSIS

This section explores bug characteristics within and across 15 VDBMSs, guided by the following RQ:

**RQ4 How do symptoms, root causes, and fix strategies differ across VDBMSs?** Our goal is to understand how variations in bug manifestation and resolution across VDBMSs relate to system-specific factors such as architectural design or programming language.

### A. RQ4: Analysis across VDBMSs.

1) *Bug Symptoms Reflect Architectural Design*: As shown in Table III, functional failures are the most common bug symptom across the evaluated systems. This pattern is especially pronounced in native VDBMSs such as Milvus (62.3%) and Chroma (55.2%), where performance-critical query execution pipelines are built from scratch. These systems often lack mature query engines, making them more prone to execution errors. In contrast, extended VDBMSs that build on top of existing DBMS infrastructures tend to exhibit a broader range of symptoms, including deployment and configuration failures. Their reliance on stable components reduces deep execution bugs, but increases the likelihood of integration-related issues. Qdrant is a notable outlier, with 40.7% of its bugs located in the storage layer. This can be attributed to its architectural decision to implement a custom storage backend, including its write-ahead log, snapshotting, and segment persistence mechanisms. These patterns suggest that **bug symptoms tend to concentrate in areas where systems deviate most from standardized infrastructure**, highlighting the cost of custom architecture in terms of reliability risks.

2) *Query-Related Bugs Vary with Functional Scope*: A notable difference across VDBMSs lies in the query layer. Mostly-vector systems, such as Chroma, focus on fast approximate nearest neighbor search with limited query complexity. These systems usually support a single index type and fixed execution flow, avoiding query optimizers or planners. As a result, query-related bugs are relatively rare (29.0%). In contrast, mostly-mixed systems like Milvus (65.0%) and Marqo (50.5%) support more expressive queries, including filtering,

TABLE III: Distribution of bug symptoms and the aspects of taxonomy across 15 vector databases.

VecDB	Symptom (%)					Taxonomy (Aspect) (%)					Total
	Crash	Perf Degrad.	Deploy Fail.	Func Error	Log/Doc	Query	Storage	Index	Parse/Inter.	Config	
Faiss	11.1%	/	77.8%	11.1%	/	/	11.1%	11.1%	/	77.8%	9
Milvus	17.1%	11.6%	4.4%	62.3%	4.7%	65.0%	18.2%	6.6%	1.7%	8.5%	363
Qdrant	40.7%	11.1%	18.5%	22.2%	7.4%	18.5%	40.7%	3.7%	/	37.0%	27
Chroma	17.2%	2.3%	23.0%	55.2%	2.3%	29.0%	15.1%	4.3%	7.5%	44.1%	93
Annoy	/	/	/	/	100.0%	/	/	100.0%	/	/	1
Weaviate	28.3%	6.5%	6.5%	53.3%	5.4%	42.4%	31.5%	4.3%	3.3%	18.5%	92
txtai	40.0%	20.0%	/	/	40.0%	20.0%	/	/	/	80.0%	5
Deep Lake	10.7%	4.8%	6.0%	66.7%	11.9%	45.3%	13.0%	0.6%	19.3%	21.7%	168
Vespa	6.6%	8.9%	9.5%	62.2%	12.7%	52.6%	4.9%	1.1%	10.3%	31.0%	348
LanceDB	17.1%	3.9%	15.8%	47.4%	15.8%	13.2%	14.5%	5.3%	15.8%	51.3%	76
Marqo	7.5%	2.8%	8.4%	63.6%	17.8%	50.5%	6.5%	3.7%	2.8%	36.4%	107
Hnswlib	33.3%	/	16.6%	33.3%	16.6%	33.3%	/	16.6%	/	50.0%	6
Usearch	21.1%	2.6%	21.1%	44.7%	10.5%	50.0%	/	2.6%	10.5%	36.8%	38
pgvectors	13.4%	8.0%	13.4%	52.7%	12.5%	46.9%	4.4%	6.2%	2.7%	39.8%	113
Voyager	11.8%	/	47.1%	41.2%	/	23.5%	/	5.9%	11.8%	58.8%	17

hybrid search, and multiple index types. These features introduce additional complexity and larger interaction surfaces between components such as planners, optimizers, and indexing modules, increasing the likelihood of query-related failures. **Richer query functionality expands retrieval power but also increases the risk of bugs**, especially when query logic interacts with multiple tightly coupled modules.

3) *Fix Strategies Reveal Differences in Modularity and Runtime Environment*: Distribution of fix strategies across VDBMSs<sup>1</sup> shows notable differences between systems. Milvus resolves 59.8% of its bugs through direct logic modifications, reflecting its monolithic and execution-centric design. Chroma demonstrates a more varied pattern, with 45.2% logic fixes, but also more dependency (14.0%) and environment (9.7%) changes, likely due to its Python-based implementation and ML toolchain integration. Systems like txtai, LanceDB, and Vespa rely more on configuration and environment-level fixes, indicating greater use of flexible deployments and scripting, which influence both failures and fix patterns.

## VI. DISCUSSION

### A. Implications

Our findings offer practical guidance for VDBMS developers, architects, and researchers to improve reliability and robustness in real-world systems.

1) *For VDBMS Developers*: **Prioritize logic robustness in query modules**. Query Algorithm Logic Issues are the most common root cause (45.7%, see Figure 4a), often due to missing exception handling or incorrect similarity computations. Developers should strengthen defensive logic, particularly for parameter validation and boundary conditions. **Address concurrency and memory management**. Issues like state synchronization failures and memory mismanagement (see Figure 4b) highlight the need for better resource control under high-concurrency workloads. Using concurrency-safe data structures and finer-grained lifecycle management can help reduce these bugs. **Expand automated test coverage for**

**edge cases**. Since many logic bugs are triggered by rare inputs (e.g., empty vectors, null IDs), developers should systematically adopt automated test suites to cover these scenarios.

2) *For VDBMS Architects*: **Ensure configuration is robust and introspectable**. Configuration bugs account for 26.5% of cases (see Figure 4e), such as missing environment variables and incompatible dependencies. We recommend startup validation, machine-checkable formats, and self-reporting diagnostics to catch misconfigurations. **Emphasize unit testing**. Incorrect code logic is the main root cause of hard-to-trace failures. Architects should ensure key modules like index updates and query coordination are thoroughly tested with both normal and edge-case inputs.

3) *For VDBMS Researchers*: **Explore memory-safe and concurrency-resilient execution frameworks**. Many crashes are caused by memory management and query coordination issues, such as use-after-free, state desynchronization, and high-concurrency contention. Researchers should investigate runtime systems or execution models optimized for bursty, data-dependent query workloads. **Advance automated bug detection**. Functional failures are the most common symptom (57.3%) and are difficult to detect with traditional analysis. Future work should develop domain-specific verification or fuzzing tools for vector quantization and similar operations.

### B. Limitations

**Granularity of taxonomy**. Given the diversity and complexity of VDBMS bugs, our taxonomy may not capture all fine-grained distinctions. To address this, we used an iterative annotation process, refining our classification criteria in parallel with the labeling. When updates were made, previously labeled cases were revisited and adjusted to ensure consistency.

**Manual labeling**. Our classification of bug symptoms, root causes, and fix strategies relied on manual labeling, which inevitably introduces some subjectivity. To mitigate this, two researchers independently labeled each sample, and disagreements were resolved by a third domain expert through discussion. Ultimately, the consensus was reached on all cases.

<sup>1</sup>Detailed distributions of fix strategies across 15 VDBMSs are available in our supplementary artifacts at <https://figshare.com/s/00034c934612a54b8620>.

## VII. CONCLUSION

VDBMSs are essential to modern AI applications, yet their reliability is not well understood. This study presents the first large-scale empirical analysis of VDBMS bugs, based on 1,671 bug-fix pull requests from 15 open-source systems. We developed a taxonomy of bugs covering symptoms, root causes, and fix strategies, identifying five major symptom categories, 31 recurring fault patterns, and 12 common fix strategies. Our findings highlight unique failure modes in vector search workloads and emphasize the central role of program logic in bug fixes. These insights offer practical guidance for improving the robustness of future VDBMSs.

## REFERENCES

- [1] Annoy, “Annoy,” <https://github.com/spotify/annoy>, 2013.
- [2] S. Arya and D. M. Mount, “Approximate nearest neighbor queries in fixed dimensions,” in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’93. USA: Society for Industrial and Applied Mathematics, 1993, p. 271–280.
- [3] Attu, “Attu,” <https://zilliz.com.cn/attu>, 2022.
- [4] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, “Toward understanding deep learning framework bugs,” 2024. [Online]. Available: <https://arxiv.org/abs/2203.04026>
- [5] Chroma, “Chroma,” <https://github.com/chroma-core/chroma>, 2022.
- [6] T. Cover and P. Hart, “Nearest neighbor pattern classification,” *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.
- [7] Z. Cui, W. Dou, Y. Gao, D. Wang, J. Song, Y. Zheng, T. Wang, R. Yang, K. Xu, Y. Hu, J. Wei, and T. Huang, “Understanding transaction bugs in database systems,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3597503.3639207>
- [8] A. Dhesi and P. Kar, “Random projection trees revisited,” in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23. Curran Associates, Inc., 2010. [Online]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2010/file/3def184ad8f4755ff269862ea77393dd-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2010/file/3def184ad8f4755ff269862ea77393dd-Paper.pdf)
- [9] J. Dong, X. Chen, M. Zhang, X. Yang, S. Chen, X. Li, and X. Wang, “Partially relevant video retrieval,” in *Proceedings of the 30th ACM International Conference on Multimedia*, ser. MM ’22. ACM, Oct. 2022, p. 246–257. [Online]. Available: <http://dx.doi.org/10.1145/3503161.3547976>
- [10] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, “The faiss library,” 2024.
- [11] R. Gray, “Vector quantization,” *IEEE ASSP Magazine*, vol. 1, no. 2, pp. 4–29, 1984.
- [12] gRPC, “gRPC,” <https://github.com/grpc/grpc>, 2025.
- [13] R. Guo, X. Luan, L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu, Z. Cao, Y. Qiao, T. Wang, B. Tang, and C. Xie, “Manu: A cloud native vector database management system,” 2022. [Online]. Available: <https://arxiv.org/abs/2206.13843>
- [14] S. Hambardzumyan, A. Tuli, L. Ghukasyan, F. Rahman, H. Topchyan, D. Isayan, M. Harutyunyan, T. Hakobyan, I. Stranic, and D. Buniatyan, “Deep lake: a lakehouse for deep learning,” 2023. [Online]. Available: <https://www.cidrdb.org/cidr2023/papers/p69-buniatyan.pdf>
- [15] H. Jégou, M. Douze, and C. Schmid, “Product quantization for nearest neighbor search,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2011.
- [16] LanceDB, “Lancedb,” <https://github.com/lancedb/lancedb>, 2023.
- [17] LangChain, “Langchain,” <https://www.langchain.com/>, 2022.
- [18] W. Liu, S. Mondal, and T.-H. Chen, “An empirical study on the characteristics of database access bugs in java applications,” 2024. [Online]. Available: <https://arxiv.org/abs/2405.15008>
- [19] LlamaIndex, “Llamaindex,” <https://docs.llamaindex.ai/>, 2023.
- [20] J. Luan, “The next stop for vector databases: 8 predictions for 2023,” <https://zilliz.com/blog/the-next-stop-for-vector-databases-8-predictions-for-2023>, 2022.
- [21] Y. A. Malkov and D. A. Yashunin, “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.
- [22] Marqo, “Marqo,” <https://github.com/marqo-ai/marqo>, 2022.
- [23] Y. Matsui, Y. Uchida, H. Jégou, and S. Satoh, “A survey of product quantization,” *ITE Transactions on Media Technology and Applications*, vol. 6, no. 1, pp. 2–10, 2018.
- [24] M. Muja and D. Lowe, “Fast approximate nearest neighbors with automatic algorithm configuration,” vol. 1, 01 2009, pp. 331–340.
- [25] J. J. Pan, J. Wang, and G. Li, “Survey of vector database management systems,” *The VLDB Journal*, vol. 33, no. 5, pp. 1591–1615, 2024.
- [26] pgvector.rs, “pgvector.rs,” <https://github.com/tensorchord/pgvector.rs>, 2023.
- [27] Qdrant, “Qdrant,” <https://github.com/qdrant/qdrant>, 2020.
- [28] L. Quan, Q. Guo, X. Xie, S. Chen, X. Li, and Y. Liu, “Towards understanding the faults of javascript-based deep learning systems,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22. ACM, Oct. 2022, p. 1–13. [Online]. Available: <http://dx.doi.org/10.1145/3551349.3560427>
- [29] M. Rigger and Z. Su, “Detecting optimization bugs in database engines via non-optimizing reference engine construction,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE ’20. ACM, Nov. 2020, p. 1140–1152. [Online]. Available: <http://dx.doi.org/10.1145/3368089.3409710>
- [30] C. Silpa-Anan and R. Hartley, “Optimised kd-trees for fast image descriptor matching,” in *2008 IEEE Conference on Computer Vision and Pattern Recognition*, 2008, pp. 1–8.
- [31] J. Tagliabue and C. Greco, “(vector) space is not the final frontier: Product search as program synthesis,” 2023. [Online]. Available: <https://arxiv.org/abs/2304.11473>
- [32] txtai, “txtai,” <https://github.com/neuml/txtai>, 2020.
- [33] A. Vardanian, “USearch by Unum Cloud,” Oct. 2023. [Online]. Available: <https://github.com/unum-cloud/usearch>
- [34] Vespa, “Vespa,” <https://github.com/vespa-engine/vespa>, 2016.
- [35] S. M. Vieira, U. Kaymak, and J. M. C. Sousa, “Cohen’s kappa coefficient as a performance measure for feature selection,” in *International Conference on Fuzzy Systems*, 2010, pp. 1–8.
- [36] Voyager, “Voyager,” <https://github.com/spotify/voyager>, 2023.
- [37] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu et al., “Milvus: A purpose-built vector data management system,” in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2614–2627.
- [38] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, “Milvus: A purpose-built vector data management system,” in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2614–2627. [Online]. Available: <https://doi.org/10.1145/3448016.3457550>
- [39] J. Wang, T. Zhang, J. Song, N. Sebe, and H. T. Shen, “A survey on learning to hash,” 2017. [Online]. Available: <https://arxiv.org/abs/1606.00185>
- [40] M. Wang, X. Xu, Q. Yue, and Y. Wang, “A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search,” 2021. [Online]. Available: <https://arxiv.org/abs/2101.12631>
- [41] S. Wang, Y. Zhao, X. Hou, and H. Wang, “Large language model supply chain: A research agenda,” *CoRR*, vol. abs/2404.12736, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2404.12736>
- [42] S. Wang, Y. Zhao, Z. Liu, Q. Zou, and H. Wang, “Sok: Understanding vulnerabilities in the large language model supply chain,” *CoRR*, vol. abs/2502.12497, 2025. [Online]. Available: <https://doi.org/10.48550/arXiv.2502.12497>
- [43] S. Wang, Y. Zhao, Y. Xie, Z. Liu, X. Hou, Q. Zou, and H. Wang, “Towards reliable vector database management systems: A software testing roadmap for 2030,” 2025. [Online]. Available: <https://arxiv.org/abs/2502.20812>
- [44] Weaviate, “Weaviate,” <https://github.com/weaviate/weaviate>, 2016.
- [45] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, “Analyticdb-v: a hybrid analytical engine towards query fusion for structured and unstructured data,” *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3152–3165, Aug. 2020. [Online]. Available: <https://doi.org/10.14778/3415478.3415541>

- [46] W. Wu, J. He, Y. Qiao, G. Fu, L. Liu, and J. Yu, "Hqann: Efficient and robust similarity search for hybrid queries with structured and unstructured constraints," 2022. [Online]. Available: <https://arxiv.org/abs/2207.07940>
- [47] Z. Wu, "Bhakti: A lightweight vector database management system for endowing large language models with semantic search capabilities and memory," 2025. [Online]. Available: <https://arxiv.org/abs/2504.01553>
- [48] Q. Yu, X. Wang, S. Liu, Y. Bai, X. Yang, X. Wang, C. Meng, S. Wu, H. Yang, H. Xiao, X. Li, F. Yang, X. Feng, L. Hu, H. Li, K. Gai, and L. Zou, "Who you are matters: Bridging topics and social roles via llm-enhanced logical recommendation," 2025. [Online]. Available: <https://arxiv.org/abs/2505.10940>