VexDB: An Agent-ready Cross-platform Multi-modal Data System

Ji Sun
Department of Computer Science, Tsinghua University
Beijing, China
sunji@tsinghua.edu.cn

ABSTRACT

The rise of agentic applications is reshaping database requirements from serving structured, centrally deployed workloads to supporting multimodal data processing across on-premises, clouds, and edge devices. Existing systems are largely fragmented along deployment environments, hardware platforms, and data modalities, forcing developers to stitch together heterogeneous engines, file formats, and AI toolchains. This fragmentation makes it difficult to build agent-ready data systems that are portable and semantically expressive. We present VexDB, an agent-ready, cross-platform, multi-modal database system designed to unify data management across on-device, on-premises, and cloud-native deployments. VexDB introduces a semantic SQL layer with AI functions, semantic operators, and a cost-quality-aware optimizer and executor, enabling declarative access to structured and multimodal data for agentic workloads. Its storage layer combines a unified catalog, hybrid row/column data management, multimodal indexing, and log-based consistency mechanisms to support data discovery, semantic search, cross-modal retrieval, lineage, versioning, and governance. To achieve portability across heterogeneous environments, VexDB decouples the data architecture from the infrastructure architecture, allowing the same system abstractions to operate over diverse compute substrates, memory organizations, and persistence backends, ranging from CPUs/GPUs to SoCs/MCUs and from cloud object-oriented stores to local page-oriented storage. VexDB demonstrates a systems approach toward a unified database foundation for next-generation agentic applications.

VLDB Workshop Reference Format:
Ji Sun. VexDB: An Agent-ready Cross-platform Multi-modal Data System. VLDB 2026 Workshop: ADS 2026: The Joint Workshop on Agentic Data Systems and Data-Centric AI (The 1st ADS & 3rd DATAI).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/VexDB-THU/VexDB-Lite.

1 INTRODUCTION

The rapid rise of large language models (LLMs) and autonomous agents is reshaping the role of database systems. Traditional DBMSs were primarily designed for structured data, deterministic query processing, and relatively stable deployment environments. In contrast, emerging agentic workloads require databases to support semantic reasoning, multimodal retrieval, iterative plan/execute/observe loops, and natural-language-driven interaction over heterogeneous data. These workloads also span a much broader deployment spectrum, ranging from standalone servers and distributed clusters to end devices and cloud-native data lakes. As a result, an *agent-ready* database system must not only manage structured and unstructured data jointly, but also preserve semantic expressiveness, operational consistency, and deployment portability across highly heterogeneous hardware and storage environments.

Existing systems each address only part of this design space. SQLite is one of the most successful embedded databases: it is serverless, self-contained, and stores a complete transactional database in a single cross-platform file [28]. However, SQLite is centered on structured, local, single-node data management and does not provide native support for multimodal indexing, semantic operators, or elastic distributed and cloud-native deployment. DuckDB extends the embedded database model toward high-performance OLAP and has become a compelling in-process analytical engine [24]. Nevertheless, DuckDB is still primarily optimized for analytical processing on CPUs in desktop or server-class environments, rather than for agent-centric multimodal workloads that must also run on edge devices.

Recent vector and AI-oriented systems push in a different direction. Milvus positions itself as a high-performance vector database that runs from lightweight local or edge settings to large-scale distributed clusters, and supports unstructured and multimodal data through embedding-based retrieval [35]. Manu further advances this line by presenting a cloud-native vector database that decouples write and read components via log-centric services, targeting elasticity and billion-scale vector workloads [10]. These systems significantly improve vector search infrastructure, but they remain fundamentally vector-centric. They do not provide a unified semantic SQL abstraction over structured and multimodal data, nor do they aim to offer one unified architecture that simultaneously covers server standalone deployments, embedded deployments, and cloud-native lakehouse-style deployments. Besides, disaggregating storage and compute completely leads to higher query latency.

The research community has recently made substantial progress on semantic and agentic data processing. LOTUS introduces declarative semantic operators such as semantic filtering, joining, ranking, and aggregation, together with optimization techniques and accuracy guarantees for LLM-based data processing [23]. DocETL shows that agent-guided query rewriting and evaluation can substantially improve the accuracy of complex document-processing pipelines [26]. These works are important steps toward agent-ready data systems, but they focus on specific semantic operators or document-centric pipeline abstractions. They do not constitute a full cross-platform DBMS architecture with integrated storage management, logging, metadata governance, multimodal indexing,

Proceedings of the VLDB Endowment. ISSN 2150-8097.

and deployment-specific execution models. In other words, the current landscape remains fragmented: embedded databases excel at lightweight local execution, vector databases excel at embedding retrieval, and semantic operator systems excel at LLM-powered transformations, yet no single system unifies these capabilities into a portable database substrate for agentic applications.

This fragmentation motivates the design of **VexDB**. VexDB is an *agent-ready, cross-platform, multi-modal database system* designed to unify semantic query processing, multimodal data management, governance metadata, and heterogeneous deployment under one architecture. The key idea behind VexDB is to cleanly separate the *data architecture* from the *infrastructure architecture*. The data architecture captures semantic SQL processing, multimodal storage, indexing, cataloging, and logging. The infrastructure architecture captures the diverse execution and persistence substrates on which the system may run, including standalone servers, shared-nothing clusters, end devices, embedded SoCs/MCUs, and cloud-native storage backends. This separation allows VexDB to preserve a common logical interface and semantic behavior across deployment forms that differ significantly in compute resources, memory organizations, and storage abstractions.

VexDB makes three technical advances over prior systems. First, it introduces a *semantic SQL layer* that integrates AI functions, semantic operators, a semantic optimizer, and a semantic executor, enabling declarative multimodal query processing with explicit cost–quality tradeoff reasoning. Second, it provides a *unified storage layer* that combines a catalog for lineage, versions, privileges, schema, metadata, statistics, correlations, and quality signals with hybrid row/column structured storage, multimodal data management, multimodal indexes, and log-based consistency mechanisms. Third, it offers a *cross-platform deployment architecture* that allows the same system design to be instantiated over server-class CPUs/GPUs, distributed clusters, end-devices, and open cloud-native datalake backends.

In summary, this paper makes the following contributions:

- We present the overall architecture of VexDB, a unified agent-ready multi-modal database system that combines an application layer, a semantic SQL layer, a multimodal storage layer, and heterogeneous infrastructure backends.
- We design a semantic SQL stack for agentic workloads, including semantic operators, AI functions, semantic optimization, and execution mechanisms that bridge structured and multimodal data processing.
- We propose a storage architecture that integrates unified catalog services, hybrid row/column structured storage, multimodal data management, index management, and log management within one system.
- We demonstrate how VexDB can be instantiated across multiple deployment forms, including standalone server deployment, standalone end-device deployment, and cloud-native deployment.

The remainder of this paper is organized as follows. Section 2 presents the overall architecture of VexDB and its main components. Section 3 describes the semantic SQL layer. Section 4 presents the storage layer. Section 5 describes the standalone deployment of VexDB, including both server and end-device forms. Section 6 presents the cloud-native deployment.

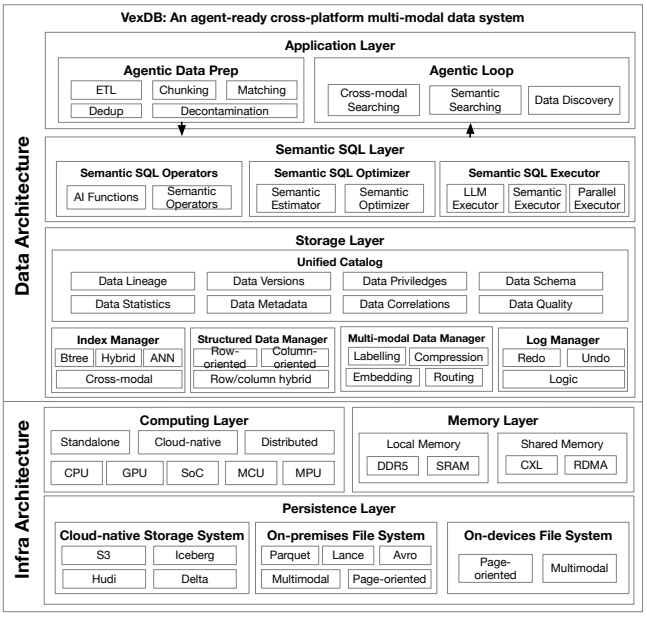

Figure 1: VexDB Architecture.

2 VEXDB ARCHITECTURE OVERVIEW

Figure 1 presents the architecture of VexDB. The central design principle of VexDB is the separation of the *data architecture* from the *infrastructure architecture*. This separation allows VexDB to expose a unified semantic interface while adapting its runtime behavior to highly heterogeneous hardware, memory, and storage substrates.

VexDB is organized into five major layers. The *application layer* hosts agent-oriented data preparation and agentic interaction loops. The *semantic SQL layer* provides declarative access to structured and multimodal data through semantic operators, AI functions, and cost-aware optimizer. The *storage layer* manages structured and multimodal data, indexes, metadata catalogs, and recovery logs. Beneath the data architecture, the *computing layer*, *memory layer*, and *persistence layer* form the infrastructure architecture that supports deployment-specific execution. Together, these layers provide a common system design that can be specialized for the deployment forms discussed in later sections.

2.1 Design Goals

VexDB is designed around four goals.

Agent readiness. VexDB is designed for workloads in which autonomous agents iteratively discover data, invoke semantic operations, inspect intermediate results, refine plans, and issue follow-up queries. This requires the database to support not only SQL execution, but also semantic search, cross-modal retrieval, metadata discovery, and AI-assisted data transformations.

Cross-platform deployment. The same logical system should run across environments with very different resource profiles, ranging from multi-core servers and GPUs to end devices. VexDB therefore avoids binding the logical query and storage model to any single execution substrate.

Multimodal unification. Modern agentic applications must jointly handle relational records, text, images, embeddings, labels, and derived semantic artifacts. VexDB unifies these assets within one system rather than delegating them to disconnected relational, vector, and file-processing subsystems.

Open and portable persistence. VexDB must operate over multiple persistence abstractions, including page-oriented device-local stores, on-premises files, and cloud-native open data-lake formats. This portability is necessary for deployment continuity from embedded and local environments to elastic cloud-scale analytics.

2.2 Application Layer

The application layer contains two main subsystems: *agentic data preparation* and the *agentic loop*. These subsystems correspond to the two dominant phases of agent-ready data workloads.

The *agentic data preparation* subsystem prepares data assets so that they can be efficiently processed by the semantic SQL layer. As shown in Figure 1, this subsystem includes ETL, chunking, matching, deduplication, and decontamination. ETL normalizes heterogeneous sources into format-compatible representations that can be ingested by the storage layer. Chunking converts long documents and multimodal objects into retrieval-friendly units that can be directly consumed by semantic operators such as `Extract`, `Semantic Filter`, and `Semantic Group`. Matching aligns entities across sources and modalities, while deduplication reduces redundancy among structured records, multimodal objects, and semantic artifacts, thereby improving both semantic selectivity estimation and query efficiency. Decontamination removes noisy, low-quality, or contaminated samples that would otherwise degrade embedding quality, candidate generation, and semantic verification. Collectively, these steps transform raw data into semantic-SQL-ready assets, including structured records, multimodal objects, embeddings, labels, and catalog metadata.

The *agentic loop* subsystem captures the online interaction pattern of autonomous agents, but its requests are ultimately executed through the semantic SQL layer rather than through an external search stack. It includes cross-modal searching, semantic searching, and data discovery. Cross-modal searching allows an agent to query one modality using another, such as retrieving images from text, text from visual evidence, or structured records from multimodal cues; in VexDB, such requests are compiled into semantic plans that combine embedding-based candidate generation with semantic verification. Semantic searching supports intent-oriented retrieval beyond lexical matching by invoking semantic operators over structured records and multimodal objects. Data discovery enables agents to identify relevant datasets, tables, objects, or views using unified-catalog metadata, semantic correlations, and index-supported retrieval. This layer is therefore the agent-facing entry point of VexDB, while the semantic SQL layer serves as the internal mechanism that translates agent requests into optimized in-database execution.

2.3 Semantic SQL Layer

The semantic SQL layer is the core abstraction that turns VexDB from a storage engine into an agent-ready database system. Its purpose is to preserve the declarative advantages of SQL while extending the language to multimodal and semantic operations required by agentic workloads.

2.3.1 Semantic SQL Operators. VexDB provides two classes of semantic SQL operators: *AI functions* and *semantic operators*. AI functions encapsulate model-backed inference primitives that can be invoked from declarative queries. Semantic operators elevate such capabilities into relationally composable query constructs so that semantic operations can be integrated with conventional relational predicates, projections, joins, and groupings. This design has two benefits. First, it provides a unified declarative interface over both structured and multimodal data, allowing agentic applications to express semantic and relational processing in a single query language. Second, by internalizing semantic processing as native database operators, VexDB enables cost- and efficiency-oriented optimization in the semantic SQL layer.

2.3.2 Semantic SQL Optimizer. Semantic processing introduces a new optimization space beyond conventional latency minimization. In VexDB, semantic execution must jointly consider runtime cost, response quality, and resource availability. For this reason, the semantic SQL optimizer contains two main components: an *estimator* and a *Pareto optimizer*.

The estimator predicts the cost and utility of candidate execution strategies, and the Pareto optimizer [25, 32, 33] then searches for plans that balance cost and quality under deployment-specific constraints. On a server-class machine, optimizer may favor more expensive high-quality inference or broader parallel exploration. On an embedded device, the same optimizer may prefer smaller models.

2.3.3 Semantic SQL Executor. The semantic SQL executor implements the optimized plan using three families of execution primitives shown in Figure 1: *LLM Executor*, *Semantic Executor*, and *Parallel Executor*. LLM executor executes LLM-based computations in an efficient manner, for example through batched inference and effective reuse via semantic caching. Semantic executor scans exploit traditional and multimodal indexes to locate the data relevent to the queries. Parallel executor coordinates available compute units when the deployment environment permits concurrency. This layered design allows VexDB to share one logical execution model while specializing physical execution to single-threaded embedded deployments, multicore servers, or distributed clusters.

2.4 Storage Layer

The storage layer is responsible for durable data management, multimodal organization, metadata governance, and recovery support. It consists of five subsystems: a unified catalog, an index manager, a structured data manager, a multimodal data manager, and a log manager.

2.4.1 Unified Catalog. The unified catalog provides the metadata foundation required for agent readiness. It stores data lineage, data versions, data privileges, schema, statistics, metadata, correlations, and data quality signals. These signals serve three purposes.

First, they improve discoverability. Agents can identify relevant tables, files, or multimodal objects not only through names and schemas, but also through quality indicators, lineage context, and

semantic metadata. Second, they improve correctness. Schema, privileges, and version information constrain what an agent can access and how query intent should be interpreted. Third, they improve governance and reproducibility. By maintaining lineage, quality metadata, and versions, VexDB supports reliable execution, auditing, and rollback across deployment environments.

2.4.2 Index Manager. The index manager supports four classes of indexes: *B-tree*, *hybrid*, *ANN*, and *cross-modal* indexes. B-tree indexes serve conventional structured predicates and ordered access. ANN indexes support embedding-based approximate nearest-neighbor retrieval. Cross-modal indexes support retrieval paths between modalities, such as text-to-image or image-to-text lookup. Hybrid indexes bridge structured filters and semantic ranking, which is critical for agentic workloads that often combine symbolic conditions with embedding-based similarity.

The placement and physical organization of index structures are critical to both query performance and retrieval accuracy. Therefore, regardless of the deployment form, local disk is the preferred location for index storage, and the system should exploit available memory as aggressively as possible for buffering, and accelerating index access.

2.4.3 Structured Data Manager. The structured data manager supports *row-oriented*, *column-oriented*, and *row/column hybrid* storage organizations. Row-oriented layouts are useful for update-heavy or point-access workloads. Column-oriented layouts are more suitable for analytical scans and projection-heavy semantic processing. Hybrid layouts provide a compromise for mixed workloads. This flexibility is essential because VexDB targets multiple workloads rather than a single workload profile.

2.4.4 Multi-modal Data Manager. The multi-modal data manager is responsible for non-relational data and their associated semantic artifacts, including labels, compressed multimodal objects, embeddings, and routing metadata. Labels represent curated or generated annotations that support semantic filtering, grouping, and downstream analytics. Compression is primarily applied to raw multimodal objects, such as images and other large binary payloads, in order to reduce storage cost while preserving query usability. Embeddings provide vector representations for semantic and cross-modal access. Routing metadata determines how different classes of data are placed across storage systems, for example by mapping different modalities to different physical backends and by scheduling hot and cold data to different storage tiers.

By explicitly materializing these assets and policies inside the DBMS, VexDB avoids pushing multimodal data management into loosely coupled external subsystems and instead treats multimodal state, storage reduction, and placement decisions as native parts of the database.

2.4.5 Log Manager. The log manager provides recovery and consistency support through *redo*, *undo*, and *logical* logs. Redo and undo logs support physical recovery and rollback semantics, while logical logs capture higher-level changes that are useful for synchronization, replication, and agent-visible version management. This design is particularly important because VexDB spans persistence backends with different update semantics, ranging from page-oriented local files to highly scalable cloud-native storage layers. The log manager therefore forms the bridge between a unified logical consistency model and heterogeneous physical storage substrates.

2.5 Infrastructure Architecture

Below the data architecture, VexDB exposes an infrastructure architecture that specializes system realization to concrete deployment environments. This architecture has three layers: computing, memory, and persistence.

2.5.1 Computing Layer. The computing layer is organized along two dimensions: deployment form and hardware target. Figure 1 shows three deployment forms: standalone, distributed, and cloud-native. The standalone form targets local single-node execution on servers or end devices. The distributed form captures scale-out execution across multiple compute nodes. The cloud-native form further combines distributed execution with disaggregated memory and storage services.

Within these deployment forms, VexDB may run over CPUs, GPUs, SoCs, MCUs, and MPUs. CPUs and GPUs dominate server-side and cloud deployments. SoCs, MCUs, and MPUs are central to end-device and embedded deployments, where power, memory, and storage are far more constrained. VexDB is designed so that the same data and query abstractions remain valid even when the physical execution substrate changes significantly.

2.5.2 Memory Layer. The memory layer is a central acceleration substrate for query processing. Different classes of memory provide different speed, capacity, and access-cost tradeoffs, and VexDB exploits them at different levels of the execution stack. Fast local memory, such as SRAM and DDR-class main memory, is the most effective medium for supplying data to compute units with low latency and high bandwidth. It is therefore the preferred location for hot metadata, active index regions, embeddings, and intermediate candidate sets. However, local memory is inherently capacity-limited. When the working set exceeds the available memory budget, the system must repeatedly evict and reload data, leading to frequent I/O operations and substantial performance degradation.

For this reason, VexDB uses memory differently under different deployment forms. On memory-constrained devices, the system emphasizes compact index layouts, aggressive index compression, and chunk-based data organization so that only a bounded working set needs to reside in memory at any point in time. This reduces memory pressure and limits the frequency of page replacement and I/O amplification. By contrast, in cloud deployments, VexDB can extend the effective memory pool through high-speed interconnects and shared-memory mechanisms such as CXL. This allows the system to access remotely attached memory with near-local efficiency, thereby expanding the usable memory space for hot indexes, semantic artifacts, and query working sets without falling back immediately to disk or object storage.

Thus, the role of the memory layer in VexDB is not merely to provide storage for runtime state. It is a key performance device that determines how much of the active query working set can be served at memory speed, and it directly shapes the physical organization of indexes, the granularity of data partitioning, and

the execution efficiency of semantic query processing across deployment environments.

2.5.3 Persistence Layer. The persistence layer allows VexDB to operate over three classes of durable backends: *cloud-native storage systems*, *on-premises file systems*, and *on-device file systems*. Cloud-native persistence includes object-store- and lakehouse-oriented ecosystems, such as S3-backed storage and open table formats including Iceberg, Hudi, and Delta. On-premises file systems support formats such as Parquet, ORC, Avro, Lance, page-oriented files and multimodal files. On-device deployments rely primarily on lightweight page-oriented files and local file system.

This design reflects an important architectural position of VexDB: persistence portability is a core design requirement. Instead of binding the system to a single datacenter-centric storage model, VexDB is designed to operate over file-based embedded storage, enterprise on-premises environments, and cloud-native open storage stacks through the same logical abstractions.

3 SEMANTIC SQL LAYER

The purpose of semantic sql layer is to bridge the agentic applications and the underlying storage layer through a unified declarative interface for both structured and multimodal data. Unlike a conventional SQL engine, which primarily supports deterministic processing over structured tuples, the semantic SQL layer must additionally support semantic reasoning, multimodal matching, semantic aggregation, and approximate retrieval while preserving the composability, optimization opportunities, and execution discipline of a database system.

Our design is inspired by recent semantic and agentic data analytics systems that translate high-level tasks into plans composed of relational and semantic operators. However, VexDB internalizes planning, optimization, and execution into the database engine, and semantic query processing in VexDB is a native in-database capability.

3.1 Running Example

We use a running example throughout this section. Consider an e-commerce setting in which VexDB stores structured product records, user review text, and advertisement images. An agent may issue a query such as:

> *Find products whose reviews express positive sentiment, whose advertisement images match the product description, and group the results by the main semantic theme of customer feedback.*

This task cannot be expressed using purely symbolic predicates. It requires extracting semantic signals from text and images across modalities, and grouping results by meaning rather than by exact lexical values. In VexDB, such a task is expressed as a semantic SQL query whose logical plan combines multimodal embedding generation, semantic operators, relational filtering, and semantic grouping. We use this example to illustrate how the semantic SQL layer is organized and executed.

To make the running example concrete, Listing 1 shows a representative semantic SQL query in VexDB. The query finds products whose reviews express positive sentiment, whose advertisement images semantically match the product description, and groups the results by the main semantic theme of customer feedback.

Listing 1: Example semantic SQL query in VexDB.

```
WITH product_text AS (
SELECT p.product_id,
       p.description,
       EMBED_TEXT(p.description) AS desc_emb
FROM Products p
),
ad_image AS (
SELECT a.ad_id,
       a.product_id,
       a.image,
       EMBED_IMAGE(a.image) AS image_emb
FROM Ads a
),
matched_ads AS (
SELECT pt.product_id,
       ai.ad_id,
       SEMANTIC_JOIN(
         pt.description,
         ai.image,
         'image matches product'
       ) AS image_match_score
FROM product_text pt
JOIN ad_image ai
  ON pt.product_id = ai.product_id
WHERE VECTOR_DISTANCE(pt.desc_emb, ai.image_emb) < 0.35
),
positive_reviews AS (
SELECT r.product_id,
       SEMANTIC_CLASSIFY(r.review_text) AS sentiment,
       EXTRACT(r.review_text, 'feedback theme') AS
     feedback_theme
FROM Reviews r
WHERE SEMANTIC_FILTER(r.review_text, 'positive review')
)
SELECT SEMANTIC_GROUP(
       pr.feedback_theme,
       'customer feedback theme'
     ) AS theme,
     COUNT(DISTINCT ma.product_id) AS product_count,
     AVG(ma.image_match_score) AS avg_match_score
FROM matched_ads ma
JOIN positive_reviews pr
ON ma.product_id = pr.product_id
WHERE ma.image_match_score > 0.8
GROUP BY theme
ORDER BY product_count DESC;
```

This query illustrates the boundary between AI functions and semantic operators in VexDB. The functions `EMBED_TEXT` and `EMBED_IMAGE` are AI functions because they only produce multimodal embedding representations. In contrast, `SEMANTIC_FILTER`, `SEMANTIC_CLASSIFY`,

EXTRACT, SEMANTIC_JOIN, and SEMANTIC_GROUP are semantic operators because they define logical semantic transformations over database records.

The query is written as a declarative SQL statement, but VexDB does not execute it as a fixed left-to-right pipeline. During optimization, the semantic SQL optimizer may rewrite the query into a cheaper but equivalent semantic plan. For example, it may use the embedding predicates on desc_emb and image_emb as a candidate-generation stage before invoking SEMANTIC_JOIN. It may also push conventional product filters below semantic operators, delay EXTRACT until the feedback theme is needed, and batch SEMANTIC_CLASSIFY calls across review partitions. These rewrites are performed inside the database engine rather than by an external agent or application-side orchestrator.

At execution time, the semantic SQL executor maps the optimized plan to its three execution primitives. Embedding predicates and vector similarity search are served by *Index Scan*; semantic operators such as SEMANTIC_CLASSIFY and SEMANTIC_JOIN are realized through *Inference*; and independent review partitions or candidate image pairs can be scheduled through *Parallel* execution. Thus, the example query exposes semantic analytics through SQL while keeping operator orchestration, optimization, and execution inside VexDB.

3.2 Semantic SQL Operators

The first component of the layer is the semantic SQL operator subsystem. As shown in Figure 1, it contains two parts: *AI Functions* and *Semantic* operators.

3.2.1 AI Functions: Multimodal Embedding Functions. In VexDB, AI Functions are deliberately scoped to *multimodal embedding functions*. Their role is to transform raw multimodal inputs, such as text, images, audio, or other unstructured content, into embeddings that can be consumed by semantic operators, the index manager, and the executor. These functions provide the foundational encoding mechanism for semantic processing.

Treating embedding generation as a native database function is important for two reasons. First, embeddings are not merely auxiliary features; they are the common representation on which semantic filtering, candidate generation, ANN search, cross-modal indexing, and semantic verification all depend. Making embedding generation native to the database therefore gives VexDB a uniform semantic representation layer across heterogeneous modalities and allows embeddings to participate directly in query processing rather than being produced in an external preprocessing pipeline. Second, once embedding generation is internalized as a database operator, the system can optimize its full lifecycle, including materialization, reuse, caching, refresh, and consistency with downstream indexes. This enables the optimizer and executor to decide when embeddings should be precomputed, when they should be generated on demand, and how they should be shared across semantic operators, which is essential for reducing cost and improving execution efficiency.

In the running example, the product description text and advertisement images are first mapped into a shared embedding space. The resulting embeddings are then used to generate candidate text-image matches before more expensive semantic verification is applied. Embedding functions are therefore not application-side preprocessing utilities; they are native database functions that participate directly in plan construction and execution.

3.2.2 Semantic Operators. Above embedding functions, VexDB exposes a semantic operator family whose scope follows the operator model used in AgenticData [31]. Specifically, VexDB supports the following semantic operators:

- **Extract.** This operator derives structured attributes or semantic keys from unstructured content. It is used when downstream query logic depends on values that are not explicitly materialized in the original representation.
- **Semantic Classify.** This operator infers sentiment-oriented semantic labels from textual or text-like content.
- **Semantic Filter.** This operator selects tuples or multimodal objects according to semantic predicates expressed in natural language or semantic criteria.
- **Semantic Group.** This operator organizes tuples or objects into semantically meaningful categories, such as latent themes or conceptual classes.
- **Semantic Join.** This operator links records or objects based on semantic relatedness instead of exact equality or symbolic key matching.

These operators are designed for semantic analytics over unstructured data or textual columns, while remaining composable with conventional relational operators.

3.3 Semantic SQL Optimizer

The second component of the semantic SQL layer is the semantic SQL optimizer. Its role is to transform a semantic SQL query into an efficient execution path under both resource and quality constraints. Unlike conventional query optimization, where the objective is primarily to minimize execution latency or I/O cost, semantic query optimization must additionally reason about inference cost, embedding generation cost, semantic selectivity, and the quality impact of approximate execution. As shown in Figure 1, the optimizer consists of two submodules: an *Estimator* and a *Pareto Optimizer*. The estimator quantifies the cost and quality of alternative subplans, while the Pareto optimizer selects execution paths that balance these objectives.

3.3.1 Estimator. The estimator predicts the cost and quality implications of alternative semantic subplans. In conventional query optimization, estimation focuses on cardinality, selectivity, and operator cost. In VexDB, this is not sufficient, because the dominant cost of a semantic plan often comes from model invocation rather than from relational processing alone. The estimator must therefore model not only the size of intermediate results, but also the semantic cost of producing them.

In VexDB, the monetary or service cost of a semantic operator is modeled from three factors: the number of invocations, the expected input size, and the expected output size. The number of invocations is derived from cardinality estimation. The expected input size is estimated from prompt templates and data statistics such as average text length or object size. The expected output size is derived from the operator semantics, for example whether the operator produces a key, a short label, or a longer extracted description. The total semantic cost of a plan can then be estimated as the sum, over all

semantic operators, of the expected invocation count multiplied by the unit cost of processing their input and output.

Cardinality and selectivity estimation are driven by the unified catalog and by runtime statistics. Structured predicates use conventional statistics, while semantic predicates rely on semantic summaries, sampled evaluation, and progressive estimation. These estimates are particularly important because semantic operator cost is highly sensitive to the number of records that reach each semantic stage.

3.3.2 Pareto Optimizer. Based on the estimator outputs, the Pareto optimizer searches for execution paths that balance cost and quality rather than optimizing only a single scalar objective. This is necessary because semantic queries often admit multiple physical plans. For example, a semantic operator may be realized through direct large-model inference, smaller-model approximation, or embedding-based candidate generation followed by selective verification. These alternatives differ substantially in latency, invocation cost, and result quality.

VexDB therefore formulates semantic optimization as a multi-objective optimization problem. The optimizer enumerates alternative execution paths and retains those at Pareto frontier with respect to execution latency, semantic cost, and expected quality.

The optimizer performs three classes of optimizations during this process.

Relational pruning before semantic evaluation. The first class of optimization pushes inexpensive relational operators ahead of expensive semantic operators whenever correctness permits. Structured filters are applied before semantic extraction, semantic filtering, or semantic joins so that fewer records reach model-backed stages. Likewise, semantic operators that do not produce values required immediately by downstream operators are postponed to avoid premature materialization of semantic artifacts. When an extraction operator produces keys used by later predicates or joins, it is placed as late as possible, immediately before those keys are needed. These transformations reduce the invocation count of semantic operators without changing query semantics.

Join ordering and pushdown. The second class of optimization reduces the cost of joins and filters by minimizing intermediate result size. Since semantic query plans often involve multiple branches and joins across heterogeneous sources, poor join order can dramatically amplify downstream semantic cost. VexDB therefore refines join ordering using a dynamic-programming-based search over join alternatives. The optimizer prefers join orders that minimize intermediate cardinality before expensive semantic evaluation. In addition, when a structured filter or join condition can be pushed into the data server or scan operator, VexDB pushes it down so that unnecessary records are eliminated earlier in the execution path.

Quality-bounded operator cascading. The third class of optimization reduces semantic cost while preserving plan quality. For many semantic operators, VexDB can choose among three implementation tiers: embedding-based candidate generation, smaller-model evaluation, and larger-model verification. The optimizer uses these tiers to construct cascaded execution paths. A candidate set may first be pruned using embeddings, then filtered by a smaller model, and finally verified by a larger model only where necessary. This reduces cost while keeping the final answer quality within a controlled bound.

VexDB models quality-bounded model selection across the entire semantic plan as a constrained optimization problem. Initially, the optimizer assumes the highest-quality model for every semantic operator. It then iteratively replaces selected operators with cheaper alternatives, including smaller models or embedding-based implementations, as long as the estimated plan quality remains above a specified threshold. At each step, the optimizer chooses the downgrade that yields the largest cost reduction per unit of quality loss. In effect, the optimizer greedily traverses the space of model assignments while preserving a global quality bound. This procedure is especially useful for plans that contain multiple semantic operators, because the most cost-effective downgrade is often not the same across operators.

3.4 Semantic SQL Executor

The semantic SQL executor is responsible for realizing the execution path selected by the optimizer inside the database engine. Unlike a conventional SQL executor, it must coordinate model-backed semantic evaluation and index-driven candidate generation within one runtime. VexDB therefore focuses on two execution-level techniques: cache-aware batched inference and parallel index scan.

Cache-aware batched inference. A major cost of semantic query execution comes from repeated model invocations over large numbers of records or multimodal objects. Executing these invocations one by one leads to poor hardware utilization and low cache reuse. VexDB addresses this problem through cache-aware batched inference. Instead of evaluating semantic operators tuple by tuple, the executor groups invocations that share the same operator type, model family, prompt template, and input structure into batches. This improves throughput in two ways. First, batched execution increases compute utilization by amortizing model invocation overhead across many records. Second, it improves semantic-cache hit rate because adjacent invocations are more likely to reuse the same prompt prefix, tokenized template, intermediate embeddings, or previously materialized semantic artifacts.

Deployment-specific parallel execution. The second key technique is deployment-specific parallel execution. In VexDB, parallelism is not implemented as a single uniform mechanism, because the available compute and memory resources differ substantially across deployment forms. Instead, the executor adapts its parallelization strategy to the underlying environment while preserving the same semantic SQL abstraction.

In standalone server deployment, VexDB exploits both CPU parallelism and GPU acceleration. CPU-side parallelism is used to partition structured records, embeddings, and index regions into independent work units, enabling concurrent candidate generation and semantic verification across multiple threads. In addition, the executor offloads high-throughput vector-intensive tasks to the GPU whenever beneficial. This is particularly important for semantic index building and semantic index scan. During index construction, GPU kernels can accelerate embedding preprocessing, distance computation, and graph- or partition-based index

construction over large embedding sets. During query execution, GPU acceleration is used to speed up vector similarity computation and candidate generation over ANN, hybrid, and cross-modal indexes. As a result, the standalone server deployment combines multicore CPU parallelism with GPU-assisted semantic indexing and retrieval.

In end-device deployment, the executor adopts a lighter-weight parallel execution model. Since the engine runs inside the caller process and typically operates under a small device-memory budget, the main objective is to avoid excessive synchronization and memory contention. Accordingly, VexDB uses bounded parallelism over data partitions, while keeping the working set within device-local memory.

In cloud-native deployment, parallel execution is coordinated through a load-balancing layer rather than being tied to a fixed owner node for each request. Multiple compute nodes can accept incoming queries, and each node can execute semantic index scan and downstream processing over its assigned work units. With the support of high-speed interconnects and the global shared buffer, a compute node can efficiently access both local memory and remote memory. As a result, parallel execution is no longer strictly constrained by local data placement, since candidate data, index pages, and semantic artifacts can be fetched from remote nodes with near-local efficiency. At the same time, the scheduler remains locality-aware: whenever possible, it assigns work to nodes that can directly reuse hot data already resident in local memory, thereby reducing remote accesses and further improving execution efficiency. In this way, cloud-native parallel execution in VexDB integrates request-level load balancing, cross-node memory access, and locality-aware scheduling into a unified execution model.

4 STORAGE LAYER

The storage layer organizes structured records, multimodal objects, and semantic artifacts, and maintains the metadata, indexes, and logs required for semantic query processing. This layer enables the semantic SQL layer to operate over structured records, multimodal objects, embeddings, labels, and metadata within one database system.

4.1 Unified Catalog

The unified catalog is the metadata backbone of VexDB. Unlike a conventional schema catalog, it acts as a semantic control plane for agent-ready query processing.

The catalog is constructed in three stages. First, during ingestion, VexDB extracts structural metadata from structured records and multimodal objects, including schema, modality type, encoding, physical location, ownership, and stable object identity. Second, during storage-layer transformations such as ETL, chunking, deduplication, decontamination, embedding generation, and semantic labeling, VexDB records lineage and version relationships between source assets and derived semantic artifacts. Third, during maintenance and execution, the system continuously collects statistics and quality signals, including value distributions, label distributions, embedding-cluster summaries, confidence scores, duplication indicators, and semantic correlations across assets.

A key role of the unified catalog is to support selectivity estimation in the semantic SQL optimizer. For structured predicates, the catalog maintains conventional statistics such as row counts, distinct-value counts, min/max ranges, null fractions, and histograms, which are used for standard cardinality and selectivity estimation. For semantic predicates, the catalog maintains semantic statistics, including label distributions, semantic-group summaries, and historical token hit rates. These statistics provide priors for estimating the selectivity of semantic filters, semantic extraction predicates, and multimodal matching conditions. For semantic joins and cross-modal access paths, the catalog further stores semantic correlations among structured records, multimodal objects. These correlations allow the optimizer to refine selectivity estimates by accounting for dependency across branches instead of assuming independent filtering effects.

Internally, the catalog is represented as a hybrid metadata graph. Nodes correspond to datasets, tables, columns, multimodal objects, embeddings, labels, indexes, and versions, while edges encode lineage, derivation, containment, version succession, access scope, and semantic correlations. VexDB also provides external relational access interfaces over this metadata graph so that the agentic applications can consume catalog signals efficiently.

4.2 Index Manager

The index manager of VexDB is centered on two index designs that target semantic query workloads beyond conventional scalar or vector retrieval: a *hybrid index* for scalar-vector joint queries and a *cross-modal index* for retrieval across different modalities. Both are designed as storage-resident structures that can be directly consumed by the semantic SQL executor.

4.2.1 Hybrid Index. The hybrid index is designed for scalar-vector joint queries under sparse scalar predicates. In this setting, neither of the two straightforward execution paths is satisfactory. A vector-first path retrieves many embedding-near candidates that later fail the scalar predicate, while a scalar-first path may still require scanning and re-ranking a large number of vectors inside the selected scalar region. This mismatch is particularly severe when the scalar predicate is selective but the remaining vector set is still large enough to make local similarity evaluation expensive.

To address this problem, VexDB organizes the hybrid index as a balanced scalar tree augmented with local vector subindexes. The top-level structure is a balanced tree over scalar values. Internal nodes store scalar separators and child pointers, and selected internal nodes additionally maintain vector subindexes over the vectors contained in their covered scalar ranges. Leaf nodes store scalar keys and references to structured records and embeddings, but do not maintain vector subindexes.

This design exposes multiple access granularities. When the scalar predicate is highly selective, execution descends to a narrow scalar region and performs local vector evaluation only within that region. When the scalar predicate is less selective, execution can stop at an internal node and invoke the vector subindex attached to that node, thereby avoiding a large leaf-level scan. In effect, the hybrid index provides a family of access paths indexed by scalar range and vector search scope.

The key implementation issue is where vector subindexes should be materialized. VexDB does not attach a vector subindex to every node, since doing so would incur excessive storage and update overhead. Instead, vector subindexes are materialized selectively on internal nodes whose covered ranges are large enough to benefit from vector-based pruning. At query time, the optimizer uses scalar selectivity estimates to choose the appropriate access level dynamically. A highly selective predicate leads to deeper tree descent and fine-grained local ranking, while a less selective predicate leads to earlier termination at a higher internal node and direct use of the corresponding vector subindex. This cardinality-aware subindex selection is the key to making scalar-vector joint search efficient across different selectivity regimes.

4.2.2 Cross-Modal Index. The cross-modal index is designed for retrieval across modalities such as text-to-image and image-to-text search. A standard ANN index over data embeddings alone is often insufficient in this setting because of the modality gap: semantically related items from different modalities may be geometrically far apart in the shared embedding space. To address this, VexDB implements a query-aware and data-aware cross-modal index rather than a data-only vector graph.

The index is organized in three layers. The first layer is a *query graph* built over representative training queries. Queries are clustered into semantic groups, and the graph stores their proximity relationships. The second layer is a set of *query-data links* connecting each representative query to relevant data embeddings. These links encode how query semantics map to data-space regions. The third layer is a *data graph* over the embedded multimodal objects. Unlike a conventional proximity graph, this data graph contains not only short-range edges induced by vector similarity, but also long-range edges induced by shared query semantics.

The construction procedure is query-aware from the start. VexDB first builds the query graph and semantic groups, then derives query-data links, and finally augments the data graph using both geometric proximity and query-induced relationships. As a result, the graph contains semantic shortcuts that bridge regions which are far apart geometrically but frequently co-activated by cross-modal queries.

At query time, the cross-modal index does not traverse the entire graph uniformly. Instead, it first maps the incoming query to one or more semantic groups and activates only the associated edge regions. This restricts graph exploration to neighborhoods that are likely to be relevant under the current query semantics, thereby reducing both distance computations and working-set size during index scan.

The physical layout of the cross-modal index is also query-aware. Out-neighbor lists are partitioned by semantic edge groups, and edges likely to be traversed together are co-located on disk using hotness-aware placement. This reduces page misses and I/O amplification for out-of-core search. Because query distributions change over time, the cross-modal index also supports incremental repair. When drift is detected, VexDB updates only the affected query-data links and local graph regions rather than rebuilding the entire structure. This localized repair keeps maintenance overhead bounded while preserving search quality under workload evolution.

4.3 Structured Data Manager

The structured data manager uses a lightweight hybrid materialization mechanism for frequent agentic query patterns. Instead of continuously reorganizing the base table layout, VexDB keeps the original row- or column-oriented representation as the canonical copy and creates additional hybrid copies only for query patterns that are identified as highly frequent and stable. Each hybrid copy is defined by a row predicate and a column projection: the row predicate is derived from frequently used structured or semantic filters, while the column projection is derived from frequently accessed column combinations. The unified catalog records the metadata of each hybrid copy, including its predicate scope, projected columns, physical location, version, hotness, and dependency on the base data. During query optimization, VexDB can route matching queries to these hybrid copies to reduce read amplification and avoid scanning irrelevant rows or columns. To preserve consistency, updates are committed through a cascading transaction protocol: the base representation is treated as the canonical source, and dependent hybrid copies are updated in the same commit chain or marked unavailable until their changes are applied. This design keeps the structured data manager simple while still allowing VexDB to accelerate the most common agentic access patterns without uncontrolled layout fragmentation.

4.4 Multi-modal Data Manager

VexDB adopts a layered storage organization for multimodal data. At the base layer, the system stores raw *multimodal objects*. Above this layer, it stores derived *semantic artifacts*, including chunks, embeddings, labels, extracted fields, and routing metadata. A third association layer links multimodal objects and semantic artifacts back to the corresponding structured records. This organization decouples raw-object persistence from semantic-artifact lifecycle and allows re-embedding, re-labeling, and re-routing without rewriting the source objects.

4.4.1 Text storage and management. Textual content in VexDB includes documents, manuals, logs, and other unstructured or semi-structured text-bearing fields. VexDB manages text in two forms. First, it stores the original text object itself, preserving the full raw content and its metadata, such as source, encoding, language, timestamp, and ownership. Second, it materializes derived semantic artifacts over the text, including chunks, extracted attributes, labels, and embeddings.

Instead of treating a document as a single monolithic object, VexDB decomposes it into chunk-level semantic artifacts so that the semantic SQL layer can perform chunk-level retrieval, filtering, and aggregation. Each chunk has a chunk identifier and is linked to its source text object through lineage edges in the unified catalog. This allows the system to support both document-level and chunk-level semantic operations. For example, candidate generation may occur at chunk granularity, while final output may be aggregated back to the document or record level.

Text embeddings are associated with the source text objects or chunks. This metadata is important because embeddings may need to be refreshed when the underlying model changes or when routing policies select a different embedding family. Labels over text,

such as topic labels, sentiment labels, or semantic group assignments, are stored similarly. The resulting structure enables VexDB to support both symbolic text predicates and semantic text retrieval within one storage model.

4.4.2 Image storage and management. Unlike text, image payloads are usually not directly consumed by relational operators. Instead, the dominant access path is through embeddings and labels. For this reason, the multi-modal data manager is optimized to keep raw image objects and image-derived semantic artifacts decoupled but tightly associated. At the raw-object layer, VexDB stores each image as a multimodal object with stable identity and associated metadata such as source, format, resolution, compression type, and creation context.

Derived semantic artifacts for images include embeddings, labels, and object- or region-level descriptors when needed. Image embeddings are materialized as core semantic artifacts and linked to the raw image object through lineage edges. Labels may represent object tags, category assignments, scene descriptions, or semantic group memberships derived by the semantic SQL layer. When the application requires finer-grained access than whole-image retrieval, VexDB can further decompose an image into region-level semantic artifacts, each linked back to its source image.

5 STANDALONE DEPLOYMENT

We describe the standalone deployment of VexDB in two forms: standalone deployment on a server and standalone deployment on an end device. Both forms preserve the full logical architecture of VexDB, including the semantic SQL layer, the storage layer, and the unified catalog, but they differ in process boundary, user interface, and memory organization. The server form runs VexDB as a single database service on one server, while the end-device form embeds VexDB directly into the caller process and exposes the system through an in-process Function API rather than a client-server protocol.

5.1 Standalone Deployment on a Server

Figure 2 shows the standalone deployment of VexDB on a server. In this mode, VexDB runs as a single database instance and exposes standard access interfaces. The deployment consists of a client layer, a compute layer, a memory layer, a storage layer, and local physical storage.

5.1.1 Client Layer. In standalone server mode, VexDB provides *VSQL* as its native query interface and exposes compatibility layers for standard drivers such as *JDBC*, *Psycopg*, and *ODBC*. This allows existing applications, analytical tools, and agent systems to connect to VexDB as they would to a conventional database server.

From the client perspective, the system behaves as a single-node multimodal DBMS. A request submitted through VSQL or a driver API may include symbolic predicates, multimodal embedding functions, semantic operators, or combinations of them. The request is compiled into a semantic plan and executed entirely within the local VexDB instance.

5.1.2 Compute Layer. The compute layer is the execution core of the standalone server deployment. As shown in Figure 2, it contains four tightly connected core functions: *Semantic Index Building*, *Semantic Index Scan*, *Semantic Execution*, and *Semantic Catalog Construction*.

Semantic Index Building. This function constructs and refreshes the indexes used by the semantic SQL layer, including ANN indexes, hybrid scalar-vector indexes, and the query-aware and data-aware cross-modal index described in Section 4. Because the deployment is single-node and local, index construction can fully exploit the compute resources of the server without distributed coordination overhead.

Semantic Index Scan. This function provides local access paths during semantic plan execution. When a query requires candidate generation over embeddings, scalar-vector pruning, or cross-modal retrieval, the compute layer invokes the corresponding local index scans and passes the resulting candidates to later semantic verification stages.

Semantic Execution. Semantic execution realizes the optimized semantic plan using inference, index scan, and local parallel execution. Semantic operators, embedding functions, and conventional relational operators all execute inside the same local engine. This avoids the fragmentation of systems that offload semantic operations to external services.

Semantic Catalog Construction. The standalone server deployment also performs local construction and maintenance of the unified catalog. During data loading, ETL, chunking, labeling, embedding generation, and query execution, the system continuously updates schema metadata, lineage, versions, privileges, statistics, semantic correlations, and data quality signals.

Together, these four functions make the compute layer a local semantic processing engine rather than a conventional SQL executor augmented with external AI components.

5.1.3 Memory Layer. The memory layer in standalone server mode contains three components shown in Figure 2: a *Vector Buffer*, a *Shared Buffer*, and *GPU Memory*.

The *Shared Buffer* caches structured records, metadata pages, index pages, and small multimodal objects. It serves as the primary page-level cache for structured and metadata-centric access paths. The *Vector Buffer* caches vector blocks that are repeatedly accessed by ANN search, hybrid indexing, and cross-modal retrieval. Separating vector-oriented caching from conventional page buffering improves locality for semantic workloads and prevents embedding-heavy access from polluting the shared page cache.

When GPU acceleration is available, *GPU Memory* is used to accelerate embedding computation, vector similarity evaluation, and portions of semantic verification. In this deployment, the memory hierarchy therefore reflects the heterogeneous nature of semantic query processing: structured records, embeddings, and execution state exhibit different reuse patterns and benefit from different placement strategies.

5.1.4 Storage Layer and Physical Storage. At the storage level, the standalone server deployment materializes the four persistent object classes shown in Figure 2: *Meta Data*, *Index Data*, *Scalar/Vector Data*, and *Raw Data*. These logical classes are mapped to two physical storage forms: *Page-oriented Files* and *Multimodal Files*.

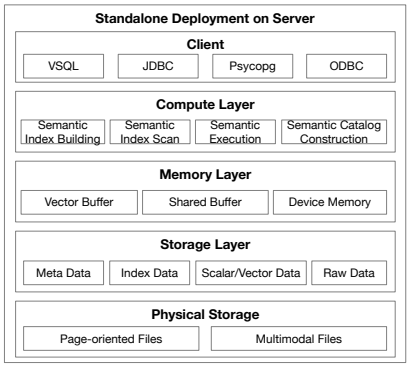

Figure 2: Standalone deployment of VexDB on a server.

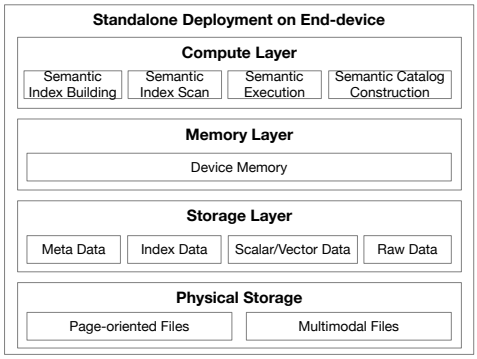

Figure 3: Standalone deployment of VexDB on an end-device.

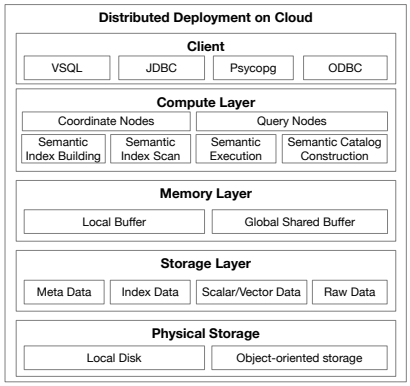

Figure 4: Distribuited deployment of VexDB on cloud.

Metadata and index structures are typically stored in page-oriented files for efficient local update and buffering. Structured records and vector-organized semantic artifacts may be stored in page-oriented or vector-aware local layouts depending on access pattern. Large raw multimodal objects, such as full text files or image binaries, are stored as multimodal files so that they can be managed independently from page-level metadata and index state.

5.2 Standalone Deployment on an End Device

The end-device deployment shares the same logical architecture as the standalone server deployment, including the semantic SQL layer, the storage layer, and the unified catalog. We therefore focus only on the differences. Compared with the server form, the end-device deployment changes the invocation interface, the memory organization, and the way indexes are built and maintained under tight local resource constraints.

5.2.1 Invocation interface. The most visible difference is that the end-device deployment removes the client layer entirely. VexDB is embedded into the caller process and exposed through an in-process Function API rather than through clients. This design eliminates inter-process communication overhead, which is especially important for latency-sensitive agent loops and interactive local applications.

5.2.2 Memory-constrained execution. Unlike the standalone server deployment, which uses a richer local memory hierarchy with a shared buffer and a vector buffer, the end-device deployment operates with a much smaller memory budget and uses only device-local memory, as shown in Figure 3. Consequently, Embeddings and semantic artifacts are processed in small batches, and cold objects are fetched on demand from local persistent storage rather than being retained in large resident buffers. This design keeps the runtime compact while preserving the same logical query semantics.

On an end device, VexDB cannot assume that all scalar/vector data, embeddings, or index structures fit in memory during index construction. It therefore builds each index incrementally under a bounded memory. For ANN indexes, embeddings are generated in batches and inserted into partition-local vector structures so that only a limited number of vectors need to reside in device memory at a time. For hybrid scalar-vector indexes, the scalar tree is built first, and only selected local vector subindexes are materialized for the scalar regions that benefit most from vector pruning, which is especially important under sparse scalar predicates. For the cross-modal index, query groups, query-data links, and local graph regions are constructed incrementally, while only the active construction state is kept in device memory and cold regions are spilled to local page-oriented files or multimodal files.

6 CLOUD-NATIVE DEPLOYMENT

The cloud-native deployment of VexDB is designed to balance elasticity and query performance in disaggregated environments. Unlike traditional storage-compute-decoupled systems that place most query-serving data in shared storage, VexDB does not treat shared storage as the default location for hot data. Instead, frequently accessed embeddings, index data, and semantic artifacts remain primarily stored on the local disks of compute nodes, while shared storage is used mainly for large raw multimodal objects. This design reflects the access characteristics of agentic workloads: semantic query latency is dominated by hot indexes, embeddings, and semantic artifacts rather than by bulk scans over raw multimodal objects. Keeping these hot structures local preserves low-latency access, while object storage provides the elasticity required for large multimodal payloads.

As shown in Figure 4, the compute layer of the cloud-native deployment remains logically identical to that of the standalone system, including semantic index building, semantic index scan, semantic execution, and semantic catalog construction. The key difference lies in the underlying memory and storage organization. In addition to the local buffer maintained at each compute node, VexDB introduces a *global shared buffer*, which acts as a memory pool shared across nodes. This memory pool provides the primary shared access path for hot data. Rather than fetching hot pages, embeddings, or semantic artifacts directly from persistent storage, compute nodes first attempt to access them through the global shared buffer.

6.1 Global Shared Buffer

The global shared buffer is enabled by high-speed interconnect technologies such as CXL and related memory-sharing protocols, which allow remote memory to be accessed with near-local efficiency. VexDB manages this shared memory pool through a *page*

owner module. The page owner tracks which node currently owns a page and mediates access control. From the perspective of the compute layer, the shared buffer exposes a uniform access interface: a request can be satisfied from local memory if the page is resident locally, or fetched remotely from the owning node if it resides elsewhere in the shared pool. In this way, VexDB obtains native memory elasticity without immediately falling back to slower disk or object storage. The result is that hot data can be reused across nodes with far lower latency than shared-disk access, significantly improving the efficiency of semantic index scan and semantic execution.

6.2 Storage Layer

The storage layer is correspondingly disaggregated according to data hotness and access pattern. Large raw multimodal objects, such as original documents, images, audio, and other binary payloads, are stored in low-cost object storage. These objects are space-intensive and are usually accessed less frequently than derived semantic structures during query processing. By contrast, frequently accessed data, including embeddings, index data, and semantic artifacts, are placed preferentially on local disks attached to compute nodes. This separation allows VexDB to combine high throughput and low latency for semantic queries with high capacity and elasticity for raw multimodal storage.

6.3 Data Sharding

The architecture does not assume that all workloads benefit equally from shared access. For workloads with nearly perfect sharding, a fully shared execution model may degrade performance by weakening data locality and introducing unnecessary remote accesses. To address this, VexDB introduces two execution roles in its cloud-native deployment: *coordinator nodes* and *query nodes*. A coordinator node decomposes a query into shard-local subtasks, dispatches them to the corresponding query nodes, and merges the returned partial results. Each query node then executes its assigned subtask primarily over locally cached data and local disk-resident index structures.

When a workload exhibits strong shard locality, the coordinator preferentially schedules each subtask to the query node that already holds the relevant hot shard-specific data in its local cache or local disk, thereby approximating shared-nothing execution. Remote memory access through the global shared buffer remains available as a fallback path, but it is no longer the primary access path for shard-local hot data. In this way, VexDB can move smoothly between two regimes: a shared-buffer-driven regime that improves cross-node reuse for skewed or mixed workloads, and a locality-preserving regime that retains the efficiency of shard-local execution for well-partitioned workloads.

7 RELATED WORK

Work closely related to VexDB spans four directions: vector and multimodal database systems, semantic query processing and natural-language database interfaces, hybrid storage and execution for mixed workloads, and cloud-native or disaggregated database architectures.

Vector and multimodal database systems. Recent systems have substantially improved vector-native and vector-augmented data management, including Milvus [35], Manu [10], SingleStore-V [5], and GaussDB-Vector [30], while Pan et al. [21] summarize this design space. Related multimodal and cross-modal access paths include RoarGraph [6], ThalamusDB [11], Starling [36], and Select Edges Wisely [37]. These systems improve vector serving and cross-modal retrieval, but they do not provide a unified semantic SQL layer that integrates semantic operators, multimodal storage management, cross-platform deployment, and metadata-driven optimization.

Semantic query processing and natural-language database interfaces. A second line of work extends database processing with semantic operators or language-driven access. LOTUS [23] introduces declarative semantic operators and accuracy-aware optimization for LLM-based data processing, and DocETL [26] studies agent-guided rewriting for document-centric pipelines. On the interface side, Katsogiannis et al. [13] and Chung et al. [8] revisit natural-language interfaces and NL2SQL, while Pan et al. [22] discuss broader LLM-knowledge-graph integration. These works advance semantic querying and language-grounded access, but they stop short of a full DBMS architecture that jointly integrates semantic SQL, multimodal indexing, workload-aware storage, and deployment-specific realization.

Hybrid storage, execution, and mixed workloads. Classic column-store and compiled-execution systems such as C-Store [29] and Neumann's query compiler [20] established the value of compressed layouts and hardware-conscious execution. Hybrid OLTP/OLAP and mixed-layout systems, including HyPer [14], Oracle Database In-Memory [15], Archipelago [4], and DuckDB [24], show that different access patterns benefit from different physical organizations. At the storage-structure level, H-Store [12], The End of a Myth [38], Bw-Tree [17], and fast serializable MVCC [16] illustrate how concurrency control and physical design interact with system architecture. VexDB differs in that its storage decisions are driven not only by transactional or analytical access, but also by semantic predicates, multimodal artifacts, and agentic projection patterns.

Cloud-native and disaggregated database architectures. A final line of work studies large-scale cloud and disaggregated systems. Dremel [19], F1 [27], and Scuba [1] illustrate scalable distributed analytics, while Aurora [34], Snowflake [9], and Amazon Redshift Reinvented [3] show how cloud-native databases redesign storage and execution for datacenter environments. Delta Lake [2] extends this trend to lakehouse storage, and recent work such as filter-agnostic vector search on PostgreSQL [18] and proxy-model-based semantic cost reduction [7] shows how semantic access paths can be optimized in production systems. Compared with these systems, VexDB unifies semantic SQL, multimodal storage and indexing, workload-aware hybrid structured layout, and deployment-specific realization under one architecture, with cloud-native deployment emphasizing shared-buffer-based hot-data access and locality-aware execution rather than placing all data in shared storage by default.

8 ACKNOWLEDGE

This work was supported by National Key RD Program of China (2023YFB4503600), NSF of China (62525202, 62232009, 62402409), Huawei, Beijing Shuzhi Yinhang Technology Co., Ltd.

REFERENCES

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. 2013. Scuba: diving into data at facebook. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1057–1067. https://doi.org/10.14778/2536222.2536231

[2] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[3] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. https://doi.org/10.1145/3514221.3526045

[4] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 583–598. https://doi.org/10.1145/2882903.2915231

[5] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chunping Wu, SzuPo Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proceedings of the VLDB Endowment* 17 (2024), 3772–3785.

[6] Meng Chen, Kai Zhang, Zhenying He, Yinan Jing, and X. Sean Wang. 2024. RoarGraph: A Projected Bipartite Graph for Efficient Cross-Modal Approximate Nearest Neighbor Search. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2735–2749. https://doi.org/10.14778/3681954.3681959

[7] Yeounoh Chung, Rushabh Desai, Jian He, Yu Xiao, Thibaud Hottelier, Yves-Laurent Kom Samo, Pushkar Kadilkar, Xianshun Chen, Sam Idicula, Fatma Ozcan, Alon Halevy, and Yannis Papakonstantinou. 2026. 100x Cost & Latency Reduction: Performance Analysis of AI Query Approximation Using Lightweight Proxy Models. https://arxiv.org/abs/2603.15970. SIGMOD 2026.

[8] Yeounoh Chung, Gaurav T. Kakkar, Yuyang Gan, Brenton Milne, and Fatma Ozcan. 2025. Is Long Context All You Need? Leveraging LLM's Extended Context for NL2SQL. *Proceedings of the VLDB Endowment* 18, 8 (2025), 2735–2747. https://doi.org/10.14778/3742728.3742761

[9] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[10] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: A Cloud Native Vector Database Management System. *Proceedings of the VLDB Endowment* 15, 12 (2022). https://arxiv.org/abs/2206.13843

[11] Seungho Jo and Immanuel Trummer. 2024. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *Proceedings of the ACM on Management of Data* 2, 3, Article 186 (2024). https://doi.org/10.1145/3654989

[12] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, and John Hugg. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[13] George Katsogiannis-Meimarakis, Michael Xydas, and Georgia Koutrika. 2023. Natural Language Interfaces for Databases with Deep Learning. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3878–3881. https://www.vldb.org/pvldb/vol16/p3878-katsogiannis-meimarakis.pdf

[14] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the IEEE International Conference on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[15] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle Database In-Memory: A Dual Format In-Memory Database. In *Proceedings of the IEEE International Conference on Data Engineering*. 1253–1258. https://doi.org/10.1109/ICDE.2015.7113373

[16] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.* 5, 4, 298–309. https://doi.org/10.14778/2095686.2095689

[17] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the IEEE International Conference on Data Engineering*. 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[18] Duo Lu, Helena Caminal, Manos Chatzakis, Yannis Papakonstantinou, Yannis Chronis, Vaibhav Jain, and Fatma Ozcan. 2026. An In-Depth Study of Filter-Agnostic Vector Search on a PostgreSQL Database System: Experiments and Analysis. *Proceedings of the ACM on Management of Data* 4, 3, Article 134 (2026). https://doi.org/10.1145/3802011 SIGMOD 2026.

[19] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 330–339. https://doi.org/10.14778/1920841.1920886

[20] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[21] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of Vector Database Management Systems. *The VLDB Journal* 33, 5 (2024), 1591–1615. https://doi.org/10.1007/S00778-024-00864-X

[22] Shuo Pan, Longbin Luo, Yao Wang, Chenyu Chen, Jiahui Wang, and Xindong Wu. 2024. Unifying Large Language Models and Knowledge Graphs: A Roadmap. *IEEE Transactions on Knowledge and Data Engineering* 36, 7 (2024), 3580–3599.

[23] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proceedings of the VLDB Endowment* (2025). https://doi.org/10.14778/3749646.3749685

[24] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. https://doi.org/10.1145/3299869.3320212

[25] Matthew Russo, Sivaprasad Sudhir, Gerardo Vitagliano, Chunwei Liu, Tim Kraska, Samuel Madden, and Michael Cafarella. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. arXiv:2505.14661 [cs.DB] https://arxiv.org/abs/2505.14661

[26] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *Proceedings of the VLDB Endowment* 18, 9 (2025), 3035–3048. https://arxiv.org/abs/2410.12189

[27] Jeffrey Shute, Mircea Oancea, Stephan Ellner, Benjamin Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Christopher Whipkey, Xin Chen, Beat Jegerlehner, and Kyle Littlefield. 2013. F1: A Distributed SQL Database That Scales. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1068–1079. https://doi.org/10.14778/2536222.2536232

[28] SQLite Consortium. 2025. About SQLite. https://sqlite.org/about.html. Official documentation, accessed 2026-07-07.

[29] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth O'Neil, Patrick O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. 553–564.

[30] Ji Sun, Guoliang Li, James Pan, Jiang Wang, Yongqing Xie, Ruicheng Liu, and Wen Nie. 2025. GaussDB-Vector: A Large-Scale Persistent Real-Time Vector Database for LLM Applications. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4951–4963.

[31] Ji Sun, Guoliang Li, Peiyao Zhou, Yihui Ma, Jingzhe Xu, and Yuan Li. 2025. AgenticData: An Agentic Data Analytics System for Heterogeneous Data. arXiv:2508.05002 [cs.DB] https://arxiv.org/abs/2508.05002

[32] Immanuel Trummer and Christoph Koch. 2014. Approximation Schemes for Many-Objective Query Optimization. arXiv:1404.0046 [cs.DB] https://arxiv.org/abs/1404.0046

[33] Immanuel Trummer and Christoph Koch. 2016. A Fast Randomized Algorithm for Multi-Objective Query Optimization. arXiv:1603.00400 [cs.DB] https://arxiv.org/abs/1603.00400

[34] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[35] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 2614–2627.

[36] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proceedings of the ACM on Management of Data* 2, 1 (2024). https://doi.org/10.1145/3639269

[37] Ziyang Yue, Bolong Zheng, Ling Xu, Kanru Xu, Shuhao Zhang, Yajuan Du, Yunjun Gao, Xiaofang Zhou, and Christian S. Jensen. 2025. Select Edges Wisely: Monotonic Path Aware Graph Layout Optimization for Disk-based ANN Search. *Proceedings of the VLDB Endowment* 18 (2025), 4337–4349.

[38] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The end of a myth: distributed transactions can scale. *Proc. VLDB Endow.* 10, 6 (Feb. 2017), 685–696. https://doi.org/10.14778/3055330.3055335