BlendHouse: A Cloud-Native Vector Database System in ByteHouse

Zhaojie Niu, Xinhui Tian, Xindong Peng, Xing Chen
ByteDance Inc
{zhaojie.niu, tianxinhui, pengxindong, chenxing.xc}@bytedance.com

***Abstract*—The rise of unstructured data retrieval in the AI era has created an urgent need for vector databases that manage high-dimensional vector embeddings and provide efficient vector search capabilities for AI applications. Performance, elasticity, and isolation are the key factors for vector databases to serve modern AI applications effectively. Disaggregation of storage and compute is widely recognized as the most effective approach in both academia and industry. Existing work either redesigns specialized vector databases according to the disaggregated architecture or integrates vector search into generalized databases that already use this architecture. However, challenges still remain in building elastic and efficient vector search systems within the disaggregated architecture, such as higher data fetching latency and the highly stateful nature of vector index, which hinder the system's ability to simultaneously achieve high performance, high elasticity and resource isolation. Additionally, a recent trend has emerged to integrate vector search into general-purpose databases, yet the extensibility and generality of integration methodologies have not been systematically studied.**

In this paper, we present BlendHouse, a cloud-native and generalized vector database system built on top of the disaggregated storage and computation architecture. BlendHouse achieves high performance, high elasticity and resource isolation simultaneously via a suite of optimizations specific to the vector search workload regarding the disaggregated architecture and the relational database. Experimental results demonstrate that BlendHouse outperforms Milvus and pgvector in terms of read and write performance. The integration methodology illustrated in this paper is extensible and general, paving the way for more powerful data management systems in the AI era.

***Index Terms*—Vector Databases; Hybird Query Processing; Cloud Native Vector Databases; Disaggregated Storage and Computation Architecture**

I. Introduction

In the era of rapid advances in Large Language Models (LLMs), Retrieval-Augmented Generation (RAG) applications have become one of the most popular AI applications, leveraging a combination of Vector Databases and LLMs to answer queries grounded in unique datasets consisting of mixed-modality data efficiently through approximate nearest neighbor (ANN) indexing. Currently, there are two main categories of vector databases: specialized and generalized vector databases. Specialized vector databases are designed from scratch to explicitly manage vector data [1]–[6]. Generalized vector databases, on the other hand, are designed to support hybrid query processing in a single SQL query that includes both vector and non-vector data by integrating vector search into relational databases, following a one-size-fits-all design philosophy [7]–[9].

Generalized vector databases have garnered more attention for various reasons, including a reluctance to move data out of relational databases to reduce data silos and costs, the desire to use SQL, and the interoperability with other SQL operators such as filters, joins, or even fulltext search. An often-cited criticism of generalized vector databases is their low performance, as they are not natively designed to support vector data. However, some recent works demonstrate that there is no fundamental limitation in using a relational database to support efficient vector data management [10], and the generalized vector database performs comparably to specialized vector databases [9] on top of the shared-nothing architecture.

Performance, **Elasticity**, and **Isolation** are key for vector databases serving modern AI applications. High performance ensures low response times and high concurrency. Elasticity allows dynamic resource allocation based on demand. Isolation minimizes interference between workloads. Disaggregating storage and compute is widely seen as the best approach to support these features [13]–[15]. Existing work either redesigns specialized vector databases for this architecture [2], [3] or integrates vector search into general-purpose databases already using it [8]. Yet, challenges remain in achieving high performance, elasticity, and isolation simultaneously. Vector search depends on a stateful in-memory index, hindering efficient scaling. Additionally, remote read latency in disaggregated setups leads to lower performance than shared-nothing systems. Further exploration is needed to build elastic and efficient vector databases on disaggregated architectures. Recent trends show growing integration of vector search into general-purpose databases for better interoperability [7], [9], [10], but extensible methods for achieving excellent hybrid query performance are still underexplored.

Motivated by this, we present BlendHouse, a cloud-native and generalized vector database to achieve high performance, high elasticity and resource isolation simultaneously. We choose to build BlendHouse in ByteHouse [16], a modern data warehouse that adopts the disaggregated storage and computation architecture. Table I shows a comparison of BlendHouse with other vector databases. Our key contributions include:

1) A cloud-native vector search framework built on the generalized relational database, adopting a disaggregated architecture. We first explore the benefits of constructing vector search systems on this architecture. Next, we inte-

2375-026X/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDE65448.2025.00325

TABLE I: Comparing BlendHouse with Other Vector Databases

Vector Databases	General-purpose	Disaggregated Architecture	Full SQL Support	Filtered Search	Iterative Search	Similarity-based Partition	Auto Index	Index Algorithms
PASE [11]	✓	✗	✓	✓	✗	✗	✗	IVF_FLAT, HNSW
pgvector [7]	✓	✗	✓	✓	✗	✗	✗	IVF_FLAT, HNSW
ElasticSearch [12]	✓	✗	✗	✓	✗	✗	✗	HNSW
AnalyticDB-V [8]	✓	✓	✓	✓	✗	✓	✗	VGPQ
SingleStore-V [9]	✓	✗	✓	✓	✓	✗	✓	Pluggable (IVF, HNSW)
Milvus [1]	✗	✓	✗	✓	✗	✗	✓	Pluggable (IVF, HNSW)
Pinecone [3]	✗	✓	✗	✓	✗	✗	✗	Not disclosed
BlendHouse	✓	✓	✓	✓	✓	✓	✓	Pluggable (IVF, HNSW)

grate vector search into the generalized query execution pipeline of the relational database, incorporating an ANN scan operator, a rich set of physical implementations, and flexible execution strategies. The seamless interoperation of vector and relational operators enables BlendHouse to support hybrid query processing within a single SQL query involving both vector and non-vector data. Finally, we introduce optimizations tailored to the disaggregated architecture to enhance vector search performance.

2) Extensible and generalized integration methodology. The extensible design enables the vector database to support vector index libraries in a pluggable fashion, allowing BlendHouse to readily adopt and integrate emerging vector indexes. As for generalizability, we focus on leveraging only the generic features available in both generalized databases and open-source index libraries for the physical plan implementations. This approach ensures that our methodology remains applicable to various systems.
3) Efficient hybrid query optimization for generalized vector databases. First, we provide the support for flexible physical plans and employ a cost-based optimizer to select the most efficient execution plan based on data statistics and workload characteristics. Second, we propose a semantic similarity-based partitioning policy that enables effective pruning of vector data during query processing. Third, we present several optimizations that tackle two key performance challenges in integrating hybrid queries with the relational model.

Through microbenchmarks and production workloads, we show that BlendHouse outperforms state-of-the-art cloud-native and shared-nothing vector databases in read/write performance. For mixed workloads, read-write separation eliminates interference. It also delivers high elasticity and real-time responsiveness during dynamic scaling. We evaluate the performance, accuracy, and resource use of various vector indexes, with our pluggable design meeting diverse user needs.

The rest of this paper is organized as follows: Section II introduces the cloud-native vector database system built in ByteHouse. Section III describes the extensible and generalized integration methodology. Section IV presents our optimizations for the hybrid query. Section V evaluates BlendHouse's performance through comprehensive experiments. Section VI describes the related work. Finally, Section VII concludes the paper and discusses future work.

II. Cloud-Native Vector Search Framework

In this section, we introduce the cloud-native vector search framework built on top of the disaggregated storage and computation architecture of ByteHouse.

A. System Architecture

BlendHouse extends ByteHouse, a cloud-native data warehouse developed by ByteDance in 2020, by incorporating vector processing capabilities. Users of ByteHouse can seamlessly use the same SQL interface within a single system to access both analytics and vector search services, without additional learning overhead or migration costs. By leveraging ByteHouse's disaggregated storage and computation architecture, BlendHouse supports key features such as read-write separation, multi-tenancy, and high elasticity. As shown in Figure 1, BlendHouse uses dedicated virtual warehouses (VWs) for building vector indexes and for handling vector search queries, preventing latency-sensitive query services from being impacted by resource-intensive index-building tasks. For various vector search applications, BlendHouse ensures multi-tenancy by isolating workloads physically across different VWs, minimizing mutual interference. Additionally, leveraging the disaggregated architecture, BlendHouse implements stateless VWs by persisting vector data and indexes in a remote distributed storage system, enabling real-time scaling of resources on demand and enhancing resource utilization.

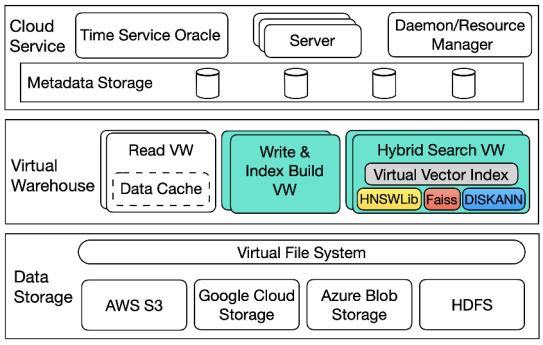


Fig. 1: Disaggregated storage and computation architecture of BlendHouse.

B. Hybrid Query SQL Dialect

Since ByteHouse provides full SQL support, we set up two integration guideline when designing the interface for vector

```
-- scalar partition and semantic partition
CREATE TABLE images
(
  id UInt64,
  label String,
  published_time DateTime,
  embedding Array(Float32),
  INDEX ann_idx embedding TYPE HNSW('DIM=960')
)
ORDER BY published_time
PARTITION BY (toYYYYMMDD(published_time),
    label)
CLUSTER BY embedding INTO 512 BUCKETS;

-- auto partition and build index
INSERT INTO images CSV INFILE 'img_data.csv';

SELECT id, dist, published_time FROM images
WHERE label = 'animal'
AND published_time >= '2024-10-10 10:00:00'
ORDER BY L2Distance(embedding, [
    query_embedding]) AS dist
LIMIT 100;
```

Example 1 : Sql dialects of hybrid query in BlendHouse

search. First, we should reuse the existing SQL syntax as much as possible to minimize the learning curves. Second, we should not disrupt existing SQL semantics. For instance, when a user incorporates vector search into a standard SQL query alongside filters, it is expected that the results will only include vectors meeting the filter criteria. For the read query, we introduce the distance functions as the customized sorting functions to calculate the semantic similarity and reuse the *ORDER BY* and *Limit* clause to represent searching for the specified number of the most similar rows. You can further use it together with other existing clauses such as *WHERE* to naturally support hybrid query processing in a single SQL query that includes both vector and non-vector data. For the vector index creation, we reuse the existing *INDEX* syntax and add several new index types to define the vector indexes such as HNSW and IVF.

Example 1 illustrates a hybrid query in BlendHouse, including table creation, vector column definition, index creation, and hybrid partitioning that combines scalar partitioning (by `published_time` and `label`) with semantic similarity-based partitioning (clustering vectors into 512 buckets). After defining the table and partitions, users can ingest data using standard `INSERT` statements, while BlendHouse handles partitioning and index building automatically. Queries use familiar `SELECT` statements with ORDER BY, LIMIT, and WHERE clauses for hybrid searches over vector and scalar data. BlendHouse optimizes execution using partition strategies and vector indexes, offering high performance while hiding complexity, thus balancing efficiency and usability for vector search workloads.

C. Vector Search Framework

Our work addresses the critical aspect of integrating vector search capabilities into the general query execution pipeline of relational databases, adopting a disaggregated storage and computation architecture—an area that has not been adequately explored in existing literature. We introduce the vector search framework from three perspectives: plan generation and optimization, scheduling, and execution. We use an example to show the execution flow of the filtered vector search query as shown in Figure 2. We highlight the components that need extension with dashed rounded rectangles and the new components with colored rounded rectangles. Reusing existing functionalities not only avoids redundant development but also fully utilizes ByteHouse's performance advantages.

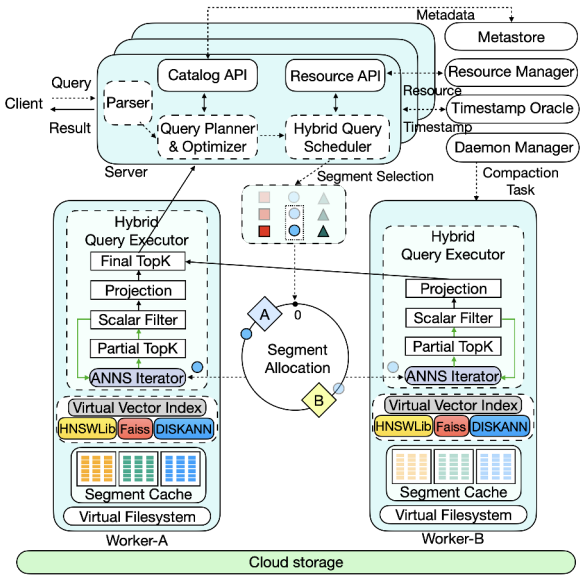


Fig. 2: Vector search execution pipeline.

Plan generation and optimization. BlendHouse generates and optimizes the hybrid query plan by following the common plan processing pipeline in relational databases. Given a hybrid query, BlendHouse's enhanced parser and planner detect the hybrid query pattern and construct the logical plan by extracting relevant components, including scalar filters, distance functions, top-k operations, and range constraints. BlendHouse introduces a new ANN scan operator, and this newly designed operator integrates with other existing operators, such as the scalar filter, to represent the hybrid query. Subsequently, BlendHouse's optimizer applies a combination of rule-based and cost-based optimizations to the logical plan to generate an efficient physical plan. For rule-based optimization, BlendHouse extends existing optimization rules by introducing distance top-k pushdown, distance range filter pushdown, and vector column pruning rules to minimize I/O and computational costs. For cost-based optimization, BlendHouse leverages the characteristics of hybrid query workloads to estimate the costs of equivalent physical plans, automatically selecting the execution strategy with the minimal cost. Based on the interaction order between non-vector and vector data, BlendHouse designs pre-filter and post-filter execution strategies to support the implementation of different physical

execution plans (section III-B). We adhere to the generalized characteristic of vector search workloads, ensuring that these optimizations and strategies can be applied to various vector search queries in existing systems.

Plan scheduling. In the disaggregated architecture, BlendHouse needs to schedule the relevant segments to the available workers before query execution. Since the vector indexes are required to be loaded into memory for optimal search performance, BlendHouse attempts to assign segments to workers where their corresponding vector indexes are cached. Instead of maintaining such a mapping relationship which would require additional maintenance costs, BlendHouse uses a ring-based consistent hash algorithm to ensure that the same segment is scheduled to the same worker when the topology of the VW keeps unchanged and minimize the portion of segments requiring redistribution when VW scales (section II-D). Considering the characteristics of hybrid query workloads, BlendHouse supports segment pruning during the scheduling phase based on query conditions. In addition to the common scalar-based pruning supported by existing systems, BlendHouse also introduces a pruning strategy based on semantic similarity, which can be adaptively adjusted at runtime to ensure accuracy (section IV-B). As shown in Figure 2, segments represented by circles are selected based on scalar column values, and among these circular segments, the two darker-colored ones are selected based on vector semantic similarity and assigned to corresponding workers for execution, which significantly reduces the I/O and computation costs.

Plan execution. We support the implementation of multiple ANN physical scan operators including *SearchWithFilter*, *SearchWithRange*, and *SearchIterator*, catering to different hybrid search execution strategies such as pre-filter and post-filter (section III.A). The rich set of physical operators enables new optimization opportunities for hybird search execution. For example, Figure 2 illustrates a efficient physical execution plan for filtered vector search implemented using the post-filter strategy and *SearchIterator*. The physical plan first employs the search iterator to retrieve the top-k most similar vectors based on distance search. Then, a partial top-k operator, pushed down before the scalar filter, calculates the top-k across multiple segments. Subsequently, filtering on scalar columns removes rows that do not satisfy the conditions. If fewer than k results remain, BlendHouse continues iterating through the same process until it returns the top-k most similar results that meet the criteria. By using the search iterator, BlendHouse performs the search incrementally in each iteration, avoiding the unnecessary repeated searches caused by existing methods that require searches from scratch with larger k. Finally, it reads the necessary scalar column data and merges the partial top-k results from multiple workers to obtain the global top-k. To minimize unnecessary vector index search costs and computational overhead for rows that do not satisfy predicates, BlendHouse leverages primary keys or related indices to efficiently skip these rows.

D. Optimizations on Disaggregated Architecture

In this section, we proposed several optimizations for the vector search system regarding the storage and computation disaggregated architecture.

Scaling-friendly segment allocation. Both planned VW scaling and random worker failures can result in vector index cache misses. When a cache miss occurs, BlendHouse reverts to brute-force distance calculations, leading to significant performance fluctuations. To mitigate the impact of scaling, we propose employing a ring-based consistent hashing algorithm for segment allocation in BlendHouse. When adding or removing any worker, the ring-based consistent hashing algorithm ensures that the portion of segments requiring redistribution is minimized, as demonstrated in related works [17]–[19]. In implementation, we use multi-probe consistent hashing [17] to achieve more balanced allocation with less space and compute complexity. Additionally, it offers a simpler implementation compared to the distributed hash table used in decentralized networks [19], [20]. For each segment allocation, multiple distinct hash functions are applied to the segment name, and the worker closest in the clockwise direction on the ring is chosen as the target worker. As illustrated in Figure 3, three hash functions are applied to a given segment. Hash2 yields the closest ring position to the next clockwise worker, thus allocating the segment to worker A.

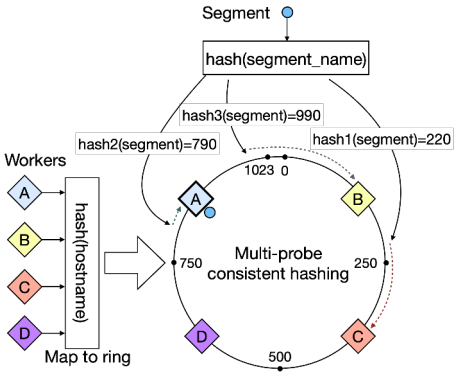


Fig. 3: Allocation with multi-probe consistent hashing.

Vector search serving. The cache miss can not be completely eliminated when scaling, to avoid the performance fluctuations, we propose a vector search serving approach in BlendHouse. This approach involves adding a search RPC interface wrapper for each worker, allowing remote access of the vector index cache. Since ANN scan is a lightweight operator compared with the end-to-end query running cost [10], enabling minimal additional resource sharing for vector index search operations between workers can improve the effectiveness of resource usage by eliminating brute force calculation, resulting the overall performance improvement. When VW scales, BlendHouse schedules segments based on the latest topology, and records the previous workers they are mapped to before the scaling. When a worker receives a segment, if the corresponding vector index does not exist in the index

cache, BlendHouse attempts to call the search RPC interface of the previous worker. As shown in Figure 4, when the new node C joins, it immediately begins executing the two assigned segments (marked in yellow and red). It would call the search APIs of nodes A and B, the nodes responsible for the segments before scaling, thus avoiding the need to fall back to expensive brute-force calculations or wait for segments to fully load, as seen in Manu [2], which leads to resource idling when an index cache miss occurs.

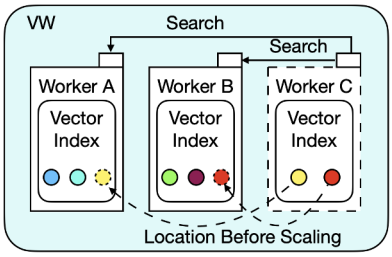


Fig. 4: Vector Search Serving.

Hierarchical vector index cache. On top of the disaggregated architecture, BlendHouse utilizes hierarchical cache design for the vector index: the in-memory index cache for best performance and local disk index cache to avoid repeatedly reading from remote shared storage. We use the Least Recently Used (LRU) policy for cache management. For the in-memory cache of vector index, we use separate cache spaces to independently manage metadata and actual data with different access patterns and sizes, preventing them from interfering with each other.

Cache-aware vector index preload. For the real-time hybrid query application which reads and writes data frequently, we provide the configurable preload feature to automatically load the vector index into both the local disk and memory of the target VW. The preload uses the same consistent hashing algorithm as the query scheduler of BlendHouse to decide the mapping between the newly built vector indexes and the workers based on the topology of the target VW. The preload feature eliminates the cache miss for the vector index of the newly ingested data and minimizes the query latency.

E. Fault Tolerance of BlendHouse

BlendHouse employs a robust fault tolerance mechanism utilizing its disaggregated architecture to ensure high availability and reliability. In case of query failures from node or network errors, the system retries at the query level. Within a VW, the disaggregated design decouples data persistence from compute nodes, allowing immediate query execution on remaining nodes upon node failure without waiting for recovery. Failed nodes recover within seconds and resume serving requests. By integrating multi-probe consistent hashing with vector search serving, BlendHouse reduces performance fluctuations from vector index cache misses, ensuring stable query latencies. Additionally, it enforces physical isolation between VWs, preventing failure cascades, and supports multiple VW replicas for critical workloads to enhance availability through redundancy.

III. Extensible and Generalized Integration

We introduce the design of an extensible and generalized methodology to integrate vector search into the generalized database in this section.

A. Extensible Vector Index Management

We have defined standardized APIs as interfaces for vector search systems, enabling easy integration with various vector index libraries. Currently, we support three popular open-source index libraries: hnswlib, faiss, and diskann. By implementing these interfaces, BlendHouse supports six index types, categorized into three groups: Graph-based (HNSW, HNSWSQ), IVF-based (IVFFLAT, IVFPQ, IVFPQFS), and Disk-based (DISKANN).

Pluggable index library. The design of vector indexes involves various trade-offs, with each type offering distinct advantages. For example, IVFPQFS provides faster index build times and lower memory usage than HNSW, but at the cost of reduced recall and search performance. Additionally, multiple implementations of the same algorithm exist, such as Faiss [21], Hnswlib [22], and Knowhere [23], each offering different versions of vector indexes. Given this diversity, it is essential to design the system with a pluggable vector index architecture. This approach allows the integration of different index types and implementations, optimizing performance based on specific constraints and ensuring compatibility with future algorithms as they emerge. As a result, BlendHouse supports pluggable vector indexes.

Virtual vector index. We define a set of interfaces for the vector search system, divided into two main categories as shown in Figure 5. At the execution layer, BlendHouse offers three search interfaces: *SearchWithFilter*, *SearchWithRange*, and *SearchIterator*. *SearchWithFilter*, supported by all index libraries, performs basic vector search and filtered vector search, retrieving the top-k results by distance while satisfying filter conditions. *SearchWithRange* returns all vectors within a specified distance threshold from the target vector. *SearchIterator* enables incremental searches over multiple rounds based on distance. At the storage layer, BlendHouse provides five interfaces: *CreateIndex*, *LoadIndex*, *SaveIndex*, *AddWithIds*, and *Train*, which handle index creation, loading, saving, adding vectors with IDs, and training (e.g., K-means for IVF index). BlendHouse integrates multiple index libraries, including Hnswlib, Faiss, and DiskANN, using their native APIs through the defined interfaces, ensuring transparency to upper-layer components. Its native support for the vector representations of these libraries eliminates extra data conversion and copying overhead.

B. Generalized Vector Search System Integration

We introduce the details of integrating vector index into ByteHouse in this section.

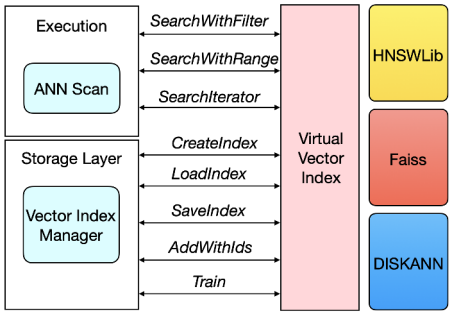


Fig. 5: The abstraction of vector search library

Per segment vector index. BlendHouse adopts a per-segment vector index approach, building a vector index for each generated immutable segment when vector indexing is defined for the table. This design offers several advantages. First, since each segment becomes immutable after commit, the per-segment vector index is built once when the segment is created and remains unchanged, eliminating the need for index rebuilding. Second, the per-segment vector index significantly improves bi-directional mapping between non-vector and vector data in filtered vector search scenarios. This improvement stems from the per-segment vector index storing row offsets rather than primary keys, allowing direct efficient mapping between non-vector and vector data through row offsets, thus avoiding the more expensive LSM-tree lookups by primary key. Third, building per-segment vector indexes on immutable segments allows us to treat the vector index algorithm as a black box, facilitating easy integration with various vector index algorithms, similar to SingleStore-V [9].

Pre-filter strategy. BlendHouse supports pre-filtering by leveraging efficient structural filtering and vector indexing capabilities. The process begins by generating a bitset that identifies all rows meeting the specified filtering conditions. This is followed by a bitset ANN scan operation on the vector index, utilizing an interface commonly available in mainstream vector indexes. This design choice ensures broad compatibility and generalizability across various vector library implementations.

Post-filter strategy. If the filtering overhead on scalar columns is high, it becomes non-trivial to perform the vector search first to select the top-k rows and then apply the filters, as this approach may yield fewer than k rows, which is undesirable. BlendHouse addresses this issue by providing an iterative interface that exposes a Next() method for the ANN index to support incremental search across multiple iterations. We extend the hnswlib library to enable iterative-based search. For indexes that do not support this interface, we adopt a generic iterator, as used by SingleStore-V [9], by wrapping the standard top-k interface of vector index algorithms. This approach retries by restarting the approximate nearest neighbor search from scratch with an initial k. If more rows are needed, k doubles in each iteration, causing redundant search overhead. Notably, repeated runs yield identical results for the same k.

Realtime update. For updates, we address the challenge of real-time modifications, which are either unsupported or prohibitively expensive in many vector indexes. Our framework enables efficient updates without requiring vector index update capabilities. As shown in Figure 6, BlendHouse achieves real-time updates through multi-versioning and delete bitmap techniques. Instead of in-place updates on the existing version, it generates a new version with the updated vector data and the corresponding vector index, and marks deletions in the old data using a delete bitmap [24]. During queries, the delete bitmap filters out obsolete rows from the old version, ensuring retrieval of the latest data from the newly generated version. BlendHouse periodically compacts data and removes bitmaps to maintain optimal query efficiency, representing a significant advancement in hybrid query systems by enabling real-time modifications without compromising query performance.

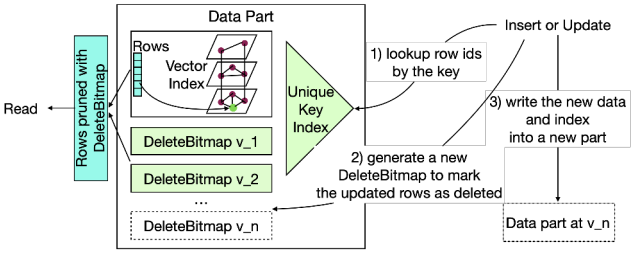


Fig. 6: Vector index updating in BlendHouse

Vector index compaction. LSM-based storage engine of BlendHouse continuously compacts multiple small segments into larger segments in the background. Our per-segment vector index design enables seamless integration of index building into the segment compaction process. When the background compaction task merges multiple small segments into a larger one, it automatically builds a new vector index for the resulting segment, thus achieving natural vector index consolidation through the existing compaction mechanism.

Auto index. In the context of the LSM-based storage engine, the segment sizes across different levels can vary significantly, leading to diverse vector index sizes in BlendHouse due to its per-segment indexing design. Our experiments reveal that for certain types of vector indexes, such as the IVF-based index, the selection of build-related parameters has a substantial impact on performance and must be dynamically adjusted according to the index size for optimal vector search performance. Specifically, we found that the parameter K_{IVF}, which controls the clustering of vectors during indexing, plays a crucial role in search performance. Figure 7 demonstrates that choosing the appropriate K_{IVF} value based on vector index size N is critical for vector search performance for IVF index. To address this, BlendHouse introduces an automatic index optimization approach, where, based on the vector index size N, key parameters like K_{IVF} are automatically selected at build time to ensure high performance. This automatic parameter selection is informed by Faiss guidelines [25] and supported by auto-tuning tools [26], [27]. For data ingestion, we prioritize quick auto-selection using rule-based methods, while for background compaction tasks, we combine the rule-

based methods with auto-tuning tools to continuously refine and optimize performance over time.

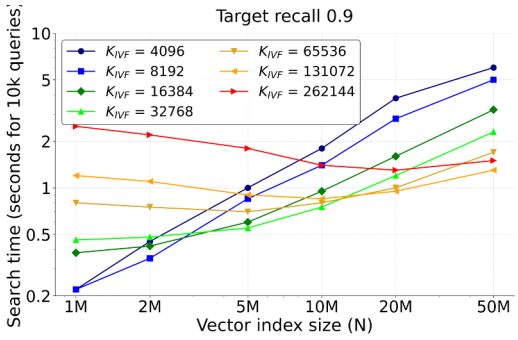


Fig. 7: Search time as a function of the rows size N with different K_{IVF} settings for index IVF$\{K_{\text{IVF}}\}$, PQ64 $\times$ 4fs,RFlat.

IV. Efficient Hybrid Query Optimization

This section introduces the optimizations on the hybrid query execution in BlendHouse.

A. Cost-based Plan Optimization

As introduced in Section II, once the optimizer of BlendHouse detects a vector search pattern, it translates the logical plan into multiple physical execution plans that perform nearest neighbor searches on unstructured columns and automatically selects a physical plan with the minimal cost. We illustrate the physical execution plans of the filtered vector search query in Figure 8. To identify the optimal plan among three plans for different scenarios, we propose an accuracy-aware cost model inspired by AnalyticDB-V [8] for hybrid query optimization. All notations used by the cost model are listed in Table II.

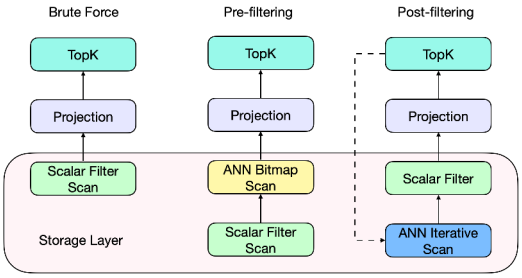


Fig. 8: Diverse hybrid query physical plans.

Plan A (brute force). Brute force search plan directly calculates the distance and sorts on the rows after the scalar filtering without dependency on the vector index. It works quite well when the number of rows satisfying the filter condition is small. Plan A starts from a structured index scan with cost T_0, and $s \times n$ records are supposed to be qualified. Then, similarities between the query vector and qualified vectors are computed via DISTANCE function. The total cost of plan A is formulated in Equation 1:

$$\text{cost}_A = T_0 + s \times n \times c_d \quad (1)$$

TABLE II: Notations used by hybrid query optimization

Notation	Meaning
n	the total number of tuples in database
s	the proportion of tuples qualifying the structured predicate (estimated with histograms [28])
β	the proportion of visited tuples in ANN scan (configured with setting ef_search or nprobe)
γ	the proportion of visited tuples in ANN bitmap scan (configured with setting ef_search or nprobe)
c_p	the total time cost to perform bitmap test for each record in ANN bitmap Scan
c_d	the total time cost to fetch a vector and compute pairwise distance
c_c	the total time cost to fetch a code and run ADC
σ	amplification factors of *ANNS Scan* operators

Plan B (pre-filtering). If the number of tuples returned by a structured index scan reaches a certain level (e.g., ten thousands of rows), the optimizer tries to insert an ANN bitmap scan node into physical plans to speed up the nearest neighbor search. It first performs a structured index scan with cost T_0 to get a set of rowids referring to the qualified tuples that are collected. Then we build a bitmap over this tuple set and start the ANN bitmap scan. The scan traverses $\gamma \times n$ records, amplified by a factor of $\frac{1}{s}$ based on selectivity, performs bitmap test for each visited record with cost c_p, and skips it if its rowid is not in the bitmap. For the qualified records, we then estimate the approximate squared distances between the query vector and each vector's code with asymmetric distance computation (ADC) [29] at cost c_c. Instead of reporting exact k rowids, we provide an optional refine process by re-ranking with DISTANCE function on the amplified $\sigma \times k (\sigma > 1)$ records. The total cost of plan B is formulated in Equation2:

$$\text{cost}_B = T_0 + \gamma \times n \times \frac{1}{s} \times (c_p + s \times c_c) + \sigma \times k \times c_d \quad (2)$$

Plan C (post-filtering). When the scalar filter cost is high and the query selectivity is low, it becomes meaningful to perform post-optimization. In plan C, we switch the execution order of predicate filter and vector index scan in contrast with plan B. The input size is significantly trimmed for predicate filter, whose running time becomes negligible. It executes in multiple iterations. For each iteration, it starts with an ANN scan to get k records and performs predicate filtering. If the current number of qualified records is less than $\sigma \times k$, it starts the next iteration. In total, the ANN scan requires traversing $\beta \times n$ records, amplified by a factor of $\frac{1}{s}$ based on selectivity, to obtain sufficient results under filtered conditions and calculate the ADC at cost c_c. The total cost of plan C is formulated in Equation 3:

$$\text{cost}_C = \beta \times n \times \frac{1}{s} \times c_c + \sigma \times k \times c_d \quad (3)$$

B. Efficient Segment Pruning

BlendHouse provides complex data management strategies which enable efficient segment pruning for hybrid query execution.

Scalar partition. Users specify a partition key, which can be a single column, multiple columns, or even an arbitrary

expression. During data ingestion, the system computes the value of the partition key and organizes data with different key values into separate data segments. Each segment is associated with a specific partition key value. During query execution, segments are filtered based on query predicates related to the partition key.

Semantic partition. Users specify the number of semantic similarity-based partitions, or buckets, guiding the system to perform k-means clustering [30] during ingestion, dividing vectors into segments based on semantic similarity, each represented by a centroid. For scalar partitioning, segment pruning relies on partition key values and query filters, while semantic partitioning prunes segments based on the similarity between the query vector and segment centroids. Accurate cardinality estimation ensures correct query results, but if inaccuracies arise, BlendHouse adaptively adjusts at runtime, dynamically scheduling more or fewer segments as needed.

C. Workload-aware Optimization

Our investigation uncoveres two performance challenges when implementing hybrid query in ByteHouse to accommodate the relational model and we propose efficient optimizations to address the two challenges accordingly.

Read amplification. ByteHouse's column-oriented storage optimizes analytics but creates inefficiencies for hybrid queries, especially when retrieving non-consecutive scalar column data after vector searches. This happens because scalar column data is organized by insertion order or sorting key, while vector search results follow semantic similarity patterns. We address this by reducing read granularity and implementing an adaptive in-memory caching mechanism that leverages the LRU algorithm for management while utilizing separate buffers for different data types—one for frequently accessed small metadata and another for larger, less frequently accessed data chunks. This separation minimizes interference between distinct access patterns. To protect the cache from thrashing by hybrid queries that read large amounts of data, BlendHouse provides a row limit setting that dynamically controls whether to use the cache during query execution.

Query processing overhead. Many hybrid query workloads feature repetitive patterns with varying parameters, leading to redundant planning and excessive optimizations by optimizers designed for complex relational queries. To mitigate this, we introduce a specialized query plan caching mechanism that uses parameterized query plan representations to capture hybrid query structures while accommodating variable filters, similarity thresholds, and search vectors. An extended plan matching algorithm enables rapid identification and adaptation of cached plans for similar queries. Additionally, we implement a short-circuit query processing mechanism that bypasses unnecessary optimizations for straightforward hybrid queries, reducing overhead and improving query latency.

V. Evaluation

In this section, we present a comprehensive and detailed evaluation of BlendHouse, utilizing both micro-benchmarks and real-world production workloads.

A. Experiment Setup

1) Hardware and Software Configuration: Our hardware configuration consists of machines equipped with Intel Xeon Gold 6230 CPU (80 cores) and 376 GB memory. We conduct experiments on both single machine and kubernetes cluster environments. Each pod in the kubernetes environment has 60 cores and 450 GB memory. We use a single machine to study fundamental performance of different systems and verify optimization effects. For evaluating system elasticity, we employ a cluster setup. We compare *BlendHouse*, based on the latest *ByteHouse* implementation, with both specialized and generalized vector databases. For specialized vector databases, we use *Milvus* (2.4.5), as it is a leading specialized vector database. For generalized vector databases, we use *pgvector* (0.7.4), as it is a popular implementation based on PostgreSQL and its performance has been demonstrated to be as good as commercial vector databases built on PostgreSQL [10]. Another major reason we choose them is because they are open source. We can perform profiling and tuning on them in evaluation. We use Queries Per Second (QPS) and latency as metrics to evaluate performance, and use recall to indicate accuracy.

2) Micro Benchmark: We evaluate the joint optimization, vector search and hybrid query performance, and workload-specific optimizations of BlendHouse using three datasets: Cohere [31], OpenAI [32], and LAION [33]. Table III provides a concise summary of all datasets.

For the Cohere and OpenAI datasets, we directly utilize VectorBench [34], a popular benchmark framework maintained by Milvus, to test vector search and hybrid query performance. The evaluation workloads encompass two key query patterns: (1) a pure top-k vector similarity search that selects IDs based on the distance between stored embeddings and the query vector, and (2) a hybrid query that combines vector similarity with an scalar filter condition of varying selectivity.

For LAION dataset, we create a workload with multiple predicates across different columns. We sample 1,000 vectors as query vectors for vector similarity search over image embeddings. For regex queries over image captions, we generate patterns using 2-10 random regex tokens (e.g., "^[0-9]"). We also add a scalar filter over the similarity column, where each query specifies a random range between a threshold and 1.0, capturing different levels of caption-image similarity requirements. The threshold is set to 0.3 to be a good heuristic for semantic image-text-content matching, as suggested by LAION team [33].

3) Production Workload: To evaluate BlendHouse's elasticity and compare its performance with Milvus in a real-world scenario, we utilize a production workload from ByteDance's online image search business. We construct a dataset of 30 million entries by sampling data over a three-day period from our production environment. The query task involves finding the k most similar images to the user's input image among the rows that meet multiple column query conditions.

TABLE III: Datasets

	# Vectors	Dim	Source Data	Structured Data	Predicate Operators
Cohere	1,000,000	768	text	random int.	ranges$(x1, x2)$
OpenAI	5,000,000	1536	text	random int.	ranges$(x1, x2)$
LAION	1,000,448	512	images	text captions & random float.	regex-match(y) & ranges$(x1, x2)$

B. Micro-benchmark Evaluation

1) Vector Search Performance: We evaluate the performance of BlendHouse against Milvus and pgvector using VectorBench. Our assessment encompasses both write operations and query performance, focusing on vector-only and hybrid queries. We first evaluate the end-to-end write performance of all systems on two datasets. All systems use HNSW index with the same construction parameters. As shown in Table IV, BlendHouse achieves better ingestion performance compared to Milvus and pgvector. The key advantage stems from BlendHouse's pipelined approach to vector index construction, where it incrementally and concurrently builds the vector index while writing segments. This approach significantly reduces the end-to-end ingestion time.

TABLE IV: Load time of different systems (Seconds)

System	Cohere	OpenAI
BlendHouse	559.1	5397.8
Milvus	783.3	9448.1
pgvector	1225.5	10068.4

Then, we evaluate the read performance of different systems on the two ingested datasets, utilizing VectorBench's vector search workload and hybrid query workloads with 1% and 99% selectivity, without implementing any partitioning strategies. Figure 9 illustrates the QPS results at 99% recall for three systems, demonstrating that BlendHouse outperforms Milvus and pgvector across both datasets and various workloads. In a simple vector search query, both BlendHouse and pgvector achieve higher QPS than Milvus, primarily due to their highly optimized execution engines. The key optimizations include vectorized execution [35] and code generation [36]. For hybrid query with 1% selectivity, another reason for their higher QPS is that both BlendHouse and pgvector select post-filter policy, which have lower costs compared to pre-filtering policy. For hybrid query workloads with 99% selectivity, since very few rows meet the conditions, both BlendHouse and Milvus chose to use the brute force method to calculate distances, achieving very high QPS. Meanwhile, while BlendHouse makes this choice automatically through its cost-based optimizer, pgvector only supports post-filter implementation, which leads to extremely low recall ($\leq 10\%$) when executing hybrid query workload with 99% selectivity. BlendHouse performs best for all workloads in VectorBench because of its efficient query engine and cost-based optimizer which automatically selects the optimal execution policy according to the workload characteristic.

We also examine the detailed performance at different accuracy levels, as shown in Figure 10. BlendHouse consistently achieves higher QPS than Milvus and pgvector across almost all recall scenarios, further confirming the efficiency of BlendHouse's query engine and query optimizer.

2) Evaluation of Cache Miss: We study the impact of vector index cache miss on vector search performance in Figure 11. When the vector index cache miss occurs, BlendHouse fallbacks to use brute force distance calculation, which results in a 14.5x increase in latency. Through our proposed vector search serving optimization, the latency increases by only 16.6% due to the remote RPC calls. It demonstrates the effectiveness of vector serving to minimize the performance fluctuation when cache miss occurs.

3) Performance Interference of Mixed Workloads: BlendHouse supports read and write isolation on top of the disaggregated architecture. To study its benefit, we show the performance interference by forcibly configuring the same VW for both read and write workload. We vary the write concurrency and show the impact on the read performance as illustrated in Figure 12. We can see that higher write concurrency causes lower read throughput, which validates the benefit of the disaggregated architecture. BlendHouse is capable to completely eliminate the interference between read and write workload by provisioning dedicate VWs separately for vector search and vector index building.

4) Evaluation of Different Vector Indexes: We show the ingestion and search performance of three most useful index types as shown in Table V and Figure 13, respectively. We also show the memory consumption of different index types from a production dataset in Table VI. These indexes have different usage scenarios. BH-HNSW is recommended for workload requiring high search accuracy. BH-HNSWSQ is suitable for the workload which requires high search efficiency and low resource consumption. BH-IVFPQFS is chosen for the workload with high write frequency under limited cost budget.

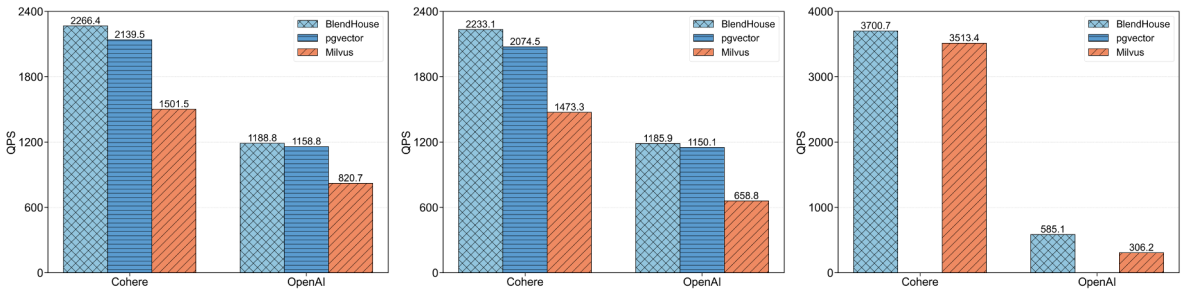
TABLE V: Load time of different index types (Seconds)

Index	Cohere	OpenAI
BH-HNSW	559.1	5397.8
BH-HNSWSQ	351.6	3484.0
BH-IVFPQFS	264.9	3046.9

TABLE VI: Memory consumption of different index types

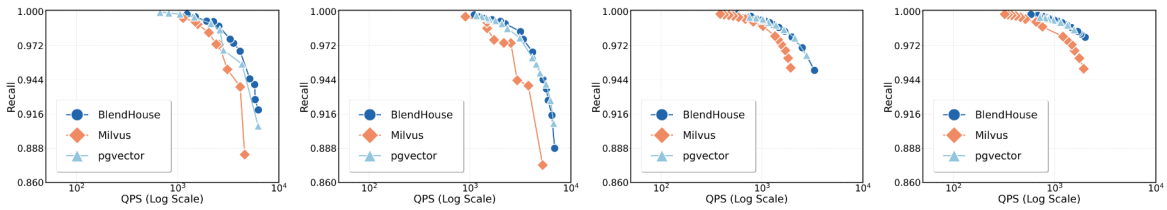
Index	Size (GB)
BH-HNSW	596.0
BH-HNSWSQ	238.4
BH-IVFPQFS	91.2

5) Evaluation of Updating Overhead: We study the impact of updates on performance by varying the number of rows to be updated and examine the effect of compaction by disabling and enabling it. When compaction is disabled, the vector



(a) Simple Query (b) Selectivity 0.01 (c) Selectivity 0.99

Fig. 9: QPS comparison of BlendHouse, pgvector, and Milvus using HNSW index at recall@99.



(a) Cohere (b) Cohere Selectivity 0.01 (c) OpenAI (d) OpenAI Selectivity 0.01

Fig. 10: Recall vs QPS of BlendHouse, pgvector and Milvus using HNSW index.

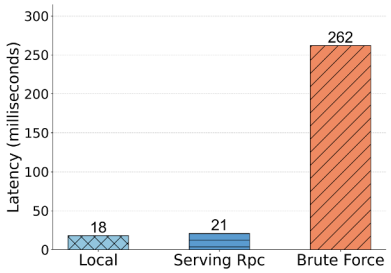


Fig. 11: Latency of local search, vector search serving and brute force calculation.

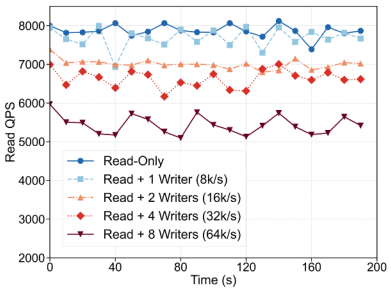
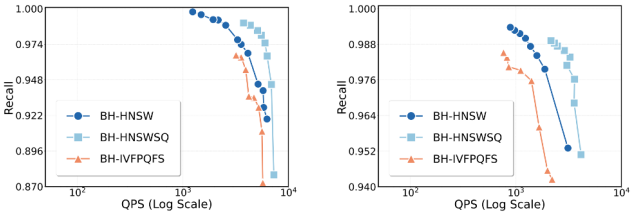


Fig. 12: The QPS of isolated workload and mixed workload.

search performance decreases with the increasing of updating rows due to the higher cost to combine the latest value for the updated rows during query execution. When compaction is enabled, the old rows marked as deleted are cleaned after compaction, the vector search performance gets back to normal without extra combining overhead.



(a) Cohere Dataset (b) OpenAI Dataset

Fig. 13: Recall vs QPS of different types of vector indexes.

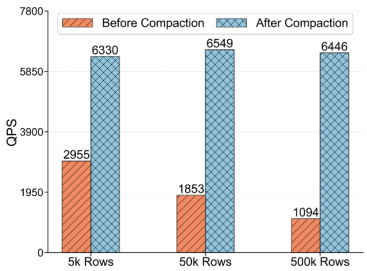


Fig. 14: The impact of update and compaction on performance.

6) Evaluation of Cost-based Plan Optimizations: We evaluate the effective cost-based optimizer of BlendHouse by manually enabling and disabling it. As shown in Figure 15, for hybrid query workloads with 1% selectivity, BlendHouse's cost-based optimizer achieves higher QPS by selecting the more efficient post-filter strategy, compared to the case where

CBO is disabled, which defaults to using pre-filter strategy.

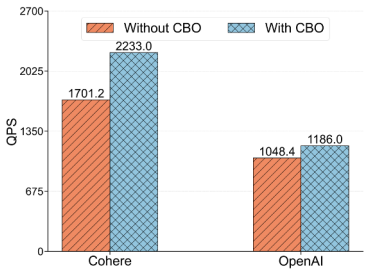


Fig. 15: QPS of different CBO settings at recall@99

7) Evaluation of Data Partition Strategy: We evaluate partitioning strategies using synthetic workloads generated from the LAION dataset. The semantic partitioning strategy clusters images using the k-means algorithm, processing only segments whose centroids are nearest to the query vector. The scalar partitioning strategy splits data based on caption-image similarity scores, selecting only segments that satisfy the filter conditions on similarity scores. As demonstrated in Figure 16, both scalar and semantic partitioning outperform the baseline random partitioning strategy, while their combination achieves the best results. This improvement stems from BlendHouse's ability to efficiently prune segments during query execution through intelligent partitioning.

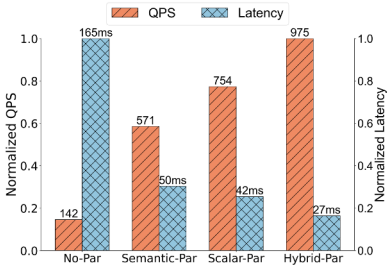


Fig. 16: Performance of different partition strategies

8) Evaluation of Workload-Aware Optimizations: To evaluate the effectiveness of our workload-aware optimizations, we performed a detailed analysis of how each optimization contributes to the overall performance improvement in hybrid query scenarios. Figure 17 illustrates the impact of BlendHouse's read optimizations and query plan optimizations. Through read optimizations (denoted as READ_Opt) using adaptive column caching and fine-grained sparse indexing, we achieve a QPS improvement of 124.2% compared to the baseline by reducing the I/O cost. Further optimization at the query plan level (denoted as READ_Opt+Query_Opt) using short-circuit query processing and hybrid query plan caching resulted in a total QPS improvement of 206.5% compared to the baseline by reducing the plan generation and optimization overhead.

C. Production Workload Evaluation

1) Performance of Production Workload: We compare BlendHouse with Milvus and pgvector on production workloads, retrieving the top-1000 most similar images, focusing on search latency and recall. Our experiments use 1,000 queries, measuring latency at 99% target recall. Both systems were tested with and without partitioning. Table VII shows that BlendHouse-with-Partition outperforms Milvus, achieving a 4.21x speedup, and 2.38x over Milvus-with-Partition. BlendHouse also delivered better recall and similar latency to Milvus-with-Partition, even without partitioning. This demonstrates BlendHouse's segment pruning and efficient multi-column filtering, powered by vectorized query execution [35]. pgvector's performance is omitted due to significantly lower recall.

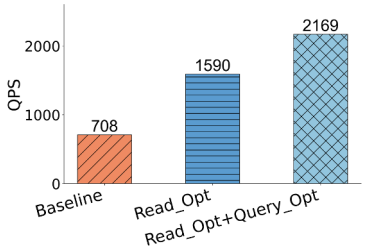


Fig. 17: Performance breakdown of workload-aware optimizations

TABLE VII: Search Latency (Seconds)

System	Recall	Latency	Speedup
Milvus	0.99221	0.181	1x
Milvus-Partition	0.99109	0.076	2.38x
ByteHouse	0.99417	0.078	2.32x
ByteHouse-Partition	0.99665	0.043	4.21x
pgvector	< 0.35	-	-
pgvector-partition	< 0.35	-	-

2) Evaluation of Elasticity: We evaluate the elasticity of BlendHouse by running a hybrid query workload with 1% selectivity on a Kubernetes cluster. We scale up the VW and measure the real-time QPS before the vector index cache loads into memory for new workers. As shown in Figure 18, the QPS increases almost linearly with scaling, demonstrating BlendHouse's high elasticity. Unlike traditional approaches where elasticity is limited by vector index loading times [2], newly added workers can immediately begin serving queries even without having local vector indexes. Vector serving effectively leverages the resources of newly added workers by enabling minimal additional resource consumption for vector index search operations in the existing nodes, resulting in overall QPS improvement.

3) Evaluation of Vector Index Compaction: We perform a detailed study on the impact of the number of segments on hybrid query performance with a production workload with extremely high write frequency. We randomly sample the system state that consists of the current number of segments and the query QPS at different times, normalize the segment number to several bins, and calculate the average QPS belonging to different bins. As shown in Figure 19, through efficient compaction, BlendHouse is able to control the number of segments to converge within a certain range, where the query

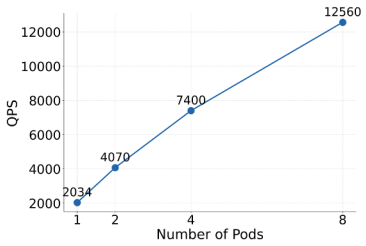


Fig. 18: The immediate query QPS in response to scaling.

QPS per worker decreases with the increase in the number of segments. The disaggregated architecture allows BlendHouse to isolate the segment compaction tasks in a dedicated VW and scale it independently according to the demands.

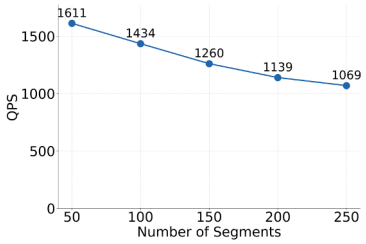


Fig. 19: Impact of the number of segments on performance.

VI. Related Work

First, we introduce ByteHouse, the database behind BlendHouse, then review related work on vector databases.

A. ByteHouse

ByteHouse [16], [37] is a multi-tenant, transactional, secure, and scalable system with full SQL support for structured data and built-in extensions for semi-structured queries and full-text search. It has three layers: cloud service, virtual warehouse, and data storage. The cloud service manages user requests and clusters, the virtual warehouse provides dynamically adjusted computation resources, and data storage uses distributed systems like AWS S3 for persistence. ByteHouse separates storage and compute, decoupling metadata and data from computation, allowing on-demand scaling without data movement overhead. It uses an LSM-based storage engine where tables consist of sorted, immutable segments that are periodically compacted for better query efficiency.

B. Specialized Vector Databases

The first stage of vector data management solutions primarily consisted of libraries designed for efficient indexing and searching [21], [38]–[46]. Optimizations for the case of vector search with attributed filter is considered in many libraries [21], [47]–[49]. These emerging libraries serve as the foundation for building high-performance vector databases.

The second stage of development saw the emergence of specialized vector databases, which are generally built upon the vector index libraries [1]–[6], [50]. While they achieved excellent vector search performance, their use of shared-nothing architecture hindered their ability to achieve better resource isolation and high elasticity. Some specialized vector databases began to redesign their systems to natively support storage and compute disaggregated architecture, achieving relatively better scalability and resource isolation [2], [3]. However, during scaling, due to changes in segment assignment relationships, node changes must wait for segments to be reassigned and loaded before taking effect, impacting their elasticity. Besides, in Manu's scaling process, the reassignment of segments is non-atomic, leading to additional computation and I/O overhead [2].

C. Generalized Vector Database

The increasing demand for unstructured data analytics has led many relational databases to incorporate hybrid query [7], [8], [10]–[12], [51], [52], following a one-size-fits-all design philosophy [53], [54]. PASE [11], pgvector [7] and VBASE [51] are generalized vector databases focused on extending the PostgreSQL ecosystem. They are known for their efficient vector similarity search and optimized indexing structures. However, due to their standalone design, they cannot achieve distributed horizontal scalability. Some distributed generalized databases integrate vector search into their systems to support hybrid query by effectively interoperating with existing features such as filtering, text search, and joins [9], [12], [55], [56]. Their tightly coupled data storage and computation limit their resource isolation and elasticity. AnalyticDB-V (ADB-V) [8] stands out as a pioneering work, being the first to support hybrid query in a generalized database that adopts a disaggregated storage and computation architecture. While it supports custom index types, its integration approach lacks extensibility and generality in design and consideration, hindering the integration of new open-source index libraries and the adaptation of its methodology to other systems.

VII. Conclusion

We have presented BlendHouse, a cloud-native vector database designed to achieve high performance, high elasticity, and resource isolation simultaneously by employing a storage and compute disaggregated architecture. The extensibility and generality of our integration approach have enabled us to leverage a wide range of algorithms and libraries while providing valuable reference for other systems. Moreover, we have implemented diverse performance optimizations for hybrid query workload. Experimental results have demonstrated that BlendHouse outperforms both Milvus and pgvector in terms of read and write performance.

Future work of BlendHouse includes two directions: (1) exploring the on-disk vector index more for better cold read performance; and (2) utilize the AI-powered method applied in autonomous database systems for more efficient hyper parameter auto tuning on vector index [57], [58].

Acknowledgment

We sincerely thank Minjie Zhu, DevOps of ByteHouse, for setting up the experiment environment, and Siyuan Wang and Changheng Cai for their early contributions to the project.

REFERENCES

[1] 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.

[2] 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," *Proceedings of the VLDB Endowment*, 2022.

[3] Pinecone Team, "Pinecone: The vector database for machine learning applications," 2021. [Online]. Available: https://www.pinecone.io/

[4] Weaviate Team, "Weaviate: A cloud-native, modular, real-time vector db," 2021. [Online]. Available: https://weaviate.io/

[5] Vespa Team, "Vespa: The open big data serving engine," 2021. [Online]. Available: https://vespa.ai/

[6] Qdrant Team, "Qdrant: Vector similarity search engine," 2021. [Online]. Available: https://qdrant.tech/

[7] A. Kane, "pgvector," 2021. [Online]. Available: https://github.com/pgvector/pgvector

[8] 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," *Proceedings of the VLDB Endowment*, 2020.

[9] C. Chen, C. Jin, Y. Zhang, S. Podolsky, C. Wu, S.-P. Wang, E. Hanson, Z. Sun, R. Walzer, and J. Wang, "Singlestore-v: An integrated vector database system in singlestore," *Proceedings of the VLDB Endowment*, 2024.

[10] Y. Zhang, S. Liu, and J. Wang, "Are there fundamental limitations in supporting vector data management in relational databases? a case study of postgresql," in *International Conference on Data Engineering (ICDE)*, 2024.

[11] W. Yang, T. Li, G. Fang, and H. Wei, "Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020.

[12] "elasticsearch." [Online]. Available: https://github.com/elastic/elasticsearch

[13] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[14] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[15] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig *et al.*, "Amazon redshift re-invented," in *Proceedings of the 2022 International Conference on Management of Data*, 2022.

[16] ByteDance, "Unified data platform for big data analytics," 2021. [Online]. Available: https://bytehouse.cloud/

[17] B. Appleton and M. O'Reilly, "Multi-probe consistent hashing," *arXiv preprint arXiv:1505.00062*, 2015.

[18] V. Mirrokni, M. Thorup, and M. Zadimoghaddam, "Consistent hashing with bounded loads," in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2018.

[19] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi *et al.*, "Service fabric: a distributed platform for building microservices in the cloud," in *Proceedings of the thirteenth EuroSys conference*, 2018.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM computer communication review*, 2001.

[21] M. Douze and et al., "The faiss library," *arXiv preprint arXiv:2401.08281*, 2024.

[22] "hnswlib." [Online]. Available: https://github.com/nmslib/hnswlib

[23] "Knowhere." [Online]. Available: https://milvus.io/docs/knowhere.md

[24] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011.

[25] "Guidelines to choose an index in faiss." [Online]. Available: https://github.com/facebookres earch/faiss/wiki/Guidelines-to-choose-an-index

[26] "An automatic k-nearest-neighbor indexing library at scale." [Online]. Available: https://github.com/criteo/autofaiss

[27] "Faiss autotune." [Online]. Available: https://github.com/facebookresearch/faiss/wiki/Index-IO,-cloning-and-hyper-parameter-tuning

[28] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996.

[29] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, 2010.

[30] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE transactions on pattern analysis and machine intelligence*, 2014.

[31] "Cohere dataset," 2022. [Online]. Available: https://huggingface.co/datasets/Cohere/wikipedia-22-12

[32] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, 2020.

[33] "laion dataset," 2021. [Online]. Available: https://laion.ai/blog/laion-400-open-dataset/

[34] Zilliz, "Vectordbbench," 2023. [Online]. Available: https://github.com/zilliztech/VectorDBBench

[35] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *Proceedings of the VLDB Endowment*, 2018.

[36] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proceedings of the VLDB Endowment*, 2011.

[37] Y. Han, H. Wang, L. Chen, Y. Dong, X. Chen, B. Yu, C. Yang, and W. Qian, "Bytecard: Enhancing bytedance's data warehouse with learned cardinality estimation," in *Companion of the 2024 International Conference on Management of Data*, 2024.

[38] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[39] 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*, 2018.

[40] M. Muja and D. Lowe, "Flann-fast library for approximate nearest neighbors user manual," *Computer Science Department, University of British Columbia, Vancouver, BC, Canada*, 2009.

[41] Q. Chen, H. Wang, M. Li, G. Ren, S. Li, J. Zhu, J. Li, C. Liu, L. Zhang, and J. Wang, *SPTAG: A library for fast approximate nearest neighbor search*, 2018. [Online]. Available: https://github.com/Microsoft/SPTAG

[42] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *Advances in Neural Information Processing Systems*, 2019.

[43] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "Spann: Highly-efficient billion-scale approximate nearest neighborhood search," *Advances in Neural Information Processing Systems*, 2021.

[44] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar, "Accelerating large-scale inference with anisotropic vector quantization," in *International Conference on Machine Learning*. PMLR, 2020.

[45] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *Proceedings of the VLDB Endowment*, 2019.

[46] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, 2014.

[47] J. Mohoney, A. Pacaci, S. R. Chowdhury, A. Mousavi, I. F. Ilyas, U. F. Minhas, J. Pound, and T. Rekatsinas, "High-throughput vector similarity search in knowledge graphs," *Proceedings of the ACM on Management of Data*, 2023.

[48] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia, "Acorn: Performant and predicate-agnostic search over vector embeddings and structured data," *Proceedings of the ACM on Management of Data*, 2024.
[49] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan, A. Singh, and H. V. Simhadri, "Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters," in *Proceedings of the ACM Web Conference 2023*, 2023.
[50] Vald Team, "Vald: A highly scalable distributed vector search engine," 2021. [Online]. Available: https://vald.vdaas.org/
[51] Q. Zhang, S. Xu, Q. Chen, G. Sui, J. Xie, Z. Cai, Y. Chen, Y. He, Y. Yang, F. Yang, M. Yang, and L. Zhou, "Vbase: Unifying online vector similarity search and relational queries via relaxed monotonicity," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 2023.
[52] Y. Chen, R. Zheng, Q. Chen, S. Xu, Q. Zhang, X. Wu, W. Han, H. Yuan, M. Li, Y. Wang, J. Li, F. Yang, H. Sun, W. Deng, F. Sun, Q. Zhang, and M. Yang, "Onesparse: A unified system for multi-index vector search," in *Companion Proceedings of the ACM on Web Conference*, 2024.
[53] M. Stonebraker and A. Pavlo, "What goes around comes around... and around..." *ACM Sigmod Record*, 2024.
[54] J. Dittrich and A. Jindal, "Towards a one size fits all database architecture." in *CIDR*, 2011.
[55] Rockset, "Introducing vector search on rockset: How to run semantic search with openai and rockset." [Online]. Available: https://rockset.com/blog/introducing-vector-search-onrockset/
[56] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang, "Tidb: a raft-based htap database," *Proceedings of the VLDB Endowment*, 2020.
[57] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li, "opengauss: An autonomous database system," *Proceedings of the VLDB Endowment*, 2021.
[58] B. C. Ooi, S. Cai, G. Chen, Y. Shen, K.-L. Tan, Y. Wu, X. Xiao, N. Xing, C. Yue, L. Zeng *et al.*, "Neurdb: an ai-powered autonomous data system," *Science China Information Sciences*, 2024.