VectraFlow: Integrating Vectors into Stream Processing

Duo Lu
Brown University
duo_lu@brown.edu

Siming Feng
Brown University
siming_feng@brown.edu

Jonathan Zhou
Brown University
jonathan_zhou@brown.edu

Franco Solleza
Brown University
franco_solleza@brown.edu

Malte Schwarzkopf
Brown University
malte@cs.brown.edu

Uğur Çetintemel
Brown University
ugur_cetintemel@brown.edu

Abstract

Vectors have quickly become the de facto standard in modern search-oriented applications due to their ability to represent complex data in a structured and effective manner. While existing work has focused mainly on scalable retrieval over vector databases, this paper is the first to tackle unique challenges and opportunities in a *streaming* environment, where continuous and scalable vector processing is required. Streaming settings present distinctive constraints, such as low-latency processing requirements and dynamic data flows, which necessitate new trade-offs and the application of both novel and established techniques.

To this end, we are developing *VectraFlow*, a stream-oriented data flow engine to support scalable monitoring applications involving vector data. We argue that VectraFlow can be effectively used to support a large suite of online applications such as continuous prompts, copyright infringement detection, and video-based surveillance where vector streams need to be processed continuously and with low latency. VectraFlow's design emphasizes efficient handling of data streams, ensuring that latency is minimized while maintaining accuracy and scalability.

We detail VectraFlow's architecture, focusing on vector-based streaming filtering, top-k, and join operations implemented through efficient clustering and novel indexing structures. We also investigate the use of alternative data representation techniques, such as quantization, which demonstrate substantial performance improvements over current methods while maintaining high-quality results. We conclude with ongoing and future research directions in this area.

1 INTRODUCTION

Vectors carry semantic richness: they can encapsulate intricate relationships and attributes of data, enhancing the effectiveness of similarity searches. For text-based search, vectors enable nuanced comparisons beyond simple keyword matching, capturing the underlying meaning and context of the data. For instance, the words "king" and "queen" can be represented as vectors in a way that their relationships and contexts are captured, allowing a search for "royalty" to return both words, even if the keyword "royalty" is not directly matched in the text. For image-based search, similarly, vectors allow for the encapsulation of multidimensional information, enabling deep similarity comparisons.

Much recent work has been dedicated to supporting scalable vector search and retrieval within traditional database settings, typically to support efficient Retrieval-Augmented Generation (RAG) for Large Language Models (LLMs) [17] via top-k queries. To this end, vector databases [2, 7, 9, 26] have emerged as a specialized solution to store and retrieve vector data, focusing primarily on similarity search. They support indexing techniques to handle large datasets where exact searches are computationally infeasible. Approximate Nearest-Neighbor Search (ANNS) approaches [7, 14, 15, 18, 19, 24–29] that trade-off performance and recall are commonly adopted in this context. They work well because vector embeddings are already intrinsically probabilistic models, and interactive performance is crucial for user-facing LLM loads.

To make the search more effective, approximations based on quantization [15, 16] are also used. For example, FAISS [14] is a vector support library that offers a wide range of methods such as product quantization, inverted files, and indexing (e.g., hierarchical navigable small-world graphs (HNSW) [19]). DiskANN [25] is another effort whose focus is on efficient disk-based indexing for handling large disk-resident datasets. There is also ongoing work to improve existing databases with vector support, notably exemplified by pgvector [22], an open-source extension that integrates vector data handling capabilities into PostgreSQL. As far as we are aware, VectraFlow is the first to provide native support for modern vectors within a traditional stream-oriented data flow architecture.

We are building VectraFlow to provide scalable, high-level vector processing support for monitoring-oriented applications that require immediate and accurate results in current and evolving AI workloads. For example, a *continuous prompt*

 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, 2025, Amsterdam, The Netherlands.

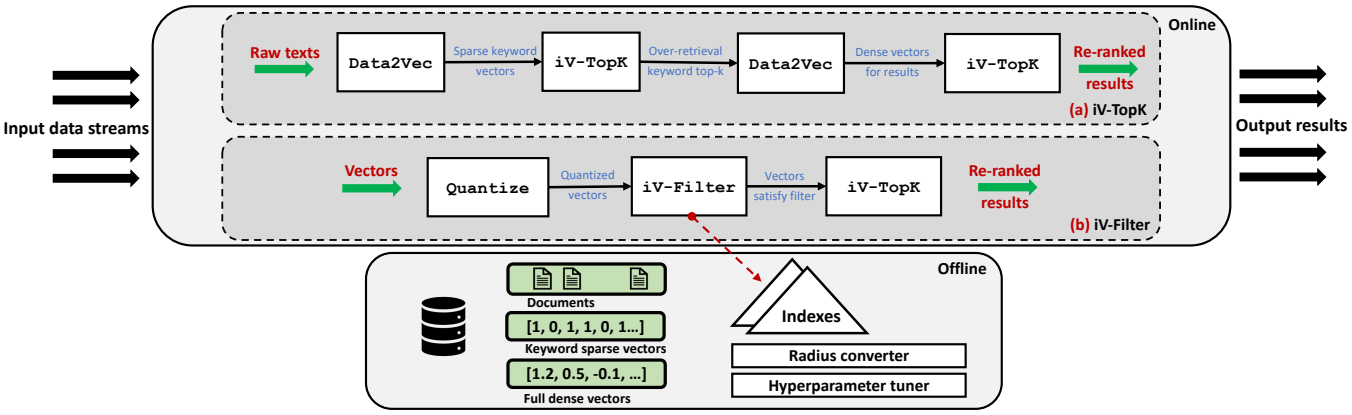

Figure 1. VectraFlow design. The figure demonstrates two data flow examples: (a) the top-k operation processes raw text, such as continuous streams of tweets, and (b) the filter operation processes dense vectors such as those of image streams. The offline component shows storage and indexing for continuous base vectors.

implemented by an LLM-based data agent would require continuous top-k results from the incoming data streams. Similarly, copyright infringement detection over LLM responses would require real-time comparisons of new content against a large database of protected works to identify unauthorized use. Real-world applications require LLM responses to be continuously monitored to ensure adherence to predefined rules and standards in real-time, providing guardrails against hallucinations, bad or biased language, or private data leakage. A video surveillance application would require rapid similarity searches to compare streaming image frames with either each other or an image database.

These and other similar applications necessitate a specialized approach beyond what existing batch-oriented systems offer for vector processing:

1. Continuous queries over vectors: Existing systems lack vector-based operators that can effectively express continuous query logic, alongside relevant data structures and optimization strategies necessary for their efficient implementation.
2. High-throughput, low-latency vector processing: Manipulating high-dimensional vectors requires new strategies for complex computations, in particular for expensive vector-to-vector distance calculations. By taking advantage of the streaming nature of the data, these computations can be optimized using batch and incremental processing techniques, along with caching methods that are not applicable in batch processing systems.
3. Balancing trade-offs between result quality and performance: There is a need to balance processing efficiency with result accuracy to maintain low-latency processing in the presence of fluctuating streaming data rates and characteristics. Batch systems, optimized for fixed workloads and tune accuracy globally, lack the flexibility to adapt in real-time, where streaming must adapt accuracy per query based on current load and query importance.
4. Supporting multi-stage dataflows: Modern AI workflows often involve multiple stages that are often chained together in a *pipelined* fashion, such as vector extraction, similarity-based filtering, clustering, and routing. Ensuring that each stage operates efficiently and in concert with others requires a tight integration of specialized operators.

To address these needs, VectraFlow (i) supports complex pipelines of vector operations, including novel vector-based versions of standard relational operations such as filtering, top-k, and joins over streaming data; (ii) employs a number of techniques, including clustering, batch and incremental processing, as well as new main-memory indexing structures that significantly reduce computational overhead; and (iii) introduces adaptive processing techniques to trade off efficiency and result quality, ensuring the system remains scalable while maintaining low latency.

In the rest of the paper, we first describe the general architecture of VectraFlow (Section 2), highlighting its data flow and vector models. We then focus on efficient support for filtering and top-k, two fundamental vector-based search operations, using clustering, indexing, and optimized vector computations (Section 3). We also discuss how to leverage quantization and other alternative data representations to optimize continuous vector-based queries using *over-retrieval and re-ranking* (Section 4). Experiments with an initial VectraFlow prototype quantify the performance of these techniques: they show that streaming-specific solutions can have significant performance gains and allow the system to trade performance and result in quality as needed (Section 5). We end the paper with a discussion of ongoing and future work (Section 6). Related work is discussed throughout the text, where appropriate.

Table 1. VectraFlow Operators

Category	Description
Manipulation	
`Data2Vec()`	Converts various data formats such as text, images, or video into sparse or dense vector representations.
`Vec2Data()`	Recovers raw data from vector representations.
`Chunk()`	Divides data into manageable segments, allowing for parallel processing or batch operations.
`V-Quantize()`	Reduces the precision of vector elements, creating a compact representation.
`V-Cluster()`	Groups vectors (k) into clusters by similarity.
`V-Partition()`	Maps vectors into clusters by similarity.
`V-Sort()`	Sorts vectors according to a specified criterion.
Vector-based Query Operators	
`V-Filter ()`	Each input vector defines a similarity query specified by a similarity threshold. The operation selects base vectors that are similar to the input vector, effectively implementing a similarity-based filter on the base vectors.
`iV-Filter ()`	Each base vector defines a similarity query specified by a similarity threshold. The operation selects input vectors that are similar to the base vector, effectively implementing a continuous (*inverse*) filter operation on the input vectors.
`V-TopK ()`	Each input vector defines a top-k query. The operation identifies the top-k base vectors most similar to the input vector, effectively implementing a top-k selection over the base vectors.
`iV-TopK ()`	Each base vector defines a continuous top-k query. The operation identifies the top-k most relevant input vectors to each base vector within a window, effectively implementing a continuous (*inverse*) top-k selection.
`V-Join ()`	Input vectors on different streams are joined if they fall within the same window and their similarity is higher than a specified join threshold. This is a continuous similarity-based join operation.
Model-based Query Operators	
`P-Filter(), P-Aggregate(), P-TopK(), P-Map(), ...`	Prompts an LLM to execute the operation. Predicate evaluation, ranking conditions, and transformations specified by the prompt.
`M-Filter(), M-Aggregate(), M-TopK(), M-Map(), ...`	Invokes a ML model to execute the operation. Predicate evaluation, ranking conditions, and transformations specified by the model.

2 VECTRAFLOW

VectraFlow's architecture follows that of traditional stream processing systems (e.g., Aurora [4], Stream [20], Flink [6], Storm [1]), and functions as a data flow system where operators connected with queues process input streams to generate output streams. There is also basic support for in-memory tables. These operators can perform a wide range of tasks, including filtering, transformations, aggregation, inference, and data enrichment. By chaining these operators, VectraFlow constructs complex data processing pipelines that can adapt to various applications and use cases.

Extended Relational Model. VectraFlow is built on the relational model, extending it to work with two new data types: `vector` and `unstructured`. The latter can be any arbitrary unstructured or semi-structured data such as free-form text, images, audio, or video data. We facilitate the conversion of these unstructured data forms into vectors using appropriate transformers (e.g., [8], [12]).

The `vector` type encapsulates data points in a structured format suitable for similarity searches and other analytical operations. Vectors are further categorized into two subtypes: sparse and dense. Sparse vectors are efficient for representing data with many dimensions but few nonzero elements, making them ideal for applications like text analysis where only a small subset of words might be present in a document. Conversely, dense vectors hold values across all dimensions, making them suitable for uniformly distributed datasets, such as image embeddings or general text embeddings, which are commonly utilized by LLMs and other machine ML models.

Data Flow and Query Model. In addition to standard relational operators that operate on structured data, VectraFlow supports custom operators to handle various aspects of vector data transformations, clustering, ranking, and filtering, allowing the system to support a wide range of applications that utilize `vector` data. These vectors may originate from streams, such as real-time cameras or social media feeds, or from tables, such as a database of static documents, and VectraFlow supports operators that create vectors from raw text and image data by invoking existing embedding libraries. These operators are shown in Table 1.

Broadly, VectraFlow's query operators are being designed to extend relational operators with the ability to work on vectors and ML models, including direct support for LLMs. The focus of this paper is the motivation and support for vector-based operators, which makes it possible to utilize the semantic expressiveness of vectors to enable a broad range of relational-like operations on unstructured data. We describe the vector-based query operators in detail in Section 3. We briefly discuss our ongoing work on integration model-based operations in Section 6.

Table 2. VectraFlow Operator Use Cases

Query operator	Example Use
`V-Filter ()`	**Retail product matching**: In an internet retail context, V-Filter can be used to identify product listings with similar images across different retail platforms. Each input vector represents a new product image, and V-Filter selects base vectors from a large product database that are similar to the input. This can help online retailers identify duplicate or nearly identical products being sold by different vendors, ensuring consistency in pricing and detecting unauthorized listings.
`iV-Filter ()`	**Real-time copyright detection**: iV-Filter can be used to continuously monitor new content being generated by LLMs, such as articles, blog posts, or even multimedia content. Each base vector represents a (chunk of) copyrighted work, and the iV-Filter operation selects input vectors (representing new LLM-generated content) that are similar to these base vectors. This ensures that infringing content is identified in real time, preventing it from being presented to users or downstream applications. This proactive detection is essential for public-facing platforms or automated workflows where adherence to copyright laws is critical, helping to prevent unauthorized use and mitigate legal risks.
`V-TopK ()`	**Retrieval-augmented generation (RAG)**: V-TopK can be used to find the top-k relevant documents or text passages to augment LLM-generated responses. Given a user query represented as an input vector, V-TopK identifies the top-k base vectors that are most similar to the query. These base vectors represent sections of a knowledge base containing relevant information. By retrieving the top-k relevant documents, VectraFlow can provide the LLM with rich and pertinent context, enabling it to generate more informed and accurate responses.
`iV-TopK ()`	**Continuous prompts:** A prompt to be continually executed is registered and converted into a base vector, representing its intent and focus. VectraFlow then continuously monitors incoming tweets, represented as input vectors, for the most relevant information that aligns with the prompt's base vector. Relevant tweets are identified and included in the prompt's context, allowing the prompt to be reevaluated and refined based on the new data. Such a continuous prompt can adapt dynamically in response to new information, ensuring that the output is always up-to-date and contextually enriched. Such an approach is particularly useful in scenarios like live event tracking or trending topic analysis, where real-time updates and adaptability are crucial to maintaining relevance.
`V-Join ()`	**Traffic monitoring:** V-Join can be used to track cars across multiple camera feeds by joining similar vectors representing visual features (e.g., color, shape, and speed) from different streams. For instance, if a car is moving through different areas covered by separate cameras, V-Join can join the vector representations of the car from different feeds, enabling continuous tracking across the entire monitored space.

Figure 1 illustrates the design of VectraFlow, for both online and offline components. The data flow in (a) receives raw texts (e.g., tweet streams), transforms them into sparse keyword vectors (`Data2Vec`), and retrieves the top matches using an over-retrieval factor (`iV-TopK`), as discussed in Section 4. Then it generates vector embeddings for only these matches (`Data2Vec`), followed by another (`iV-TopK`) to return the top matches using full vector embeddings.

The data flow in (b) quantizes the input vectors (`Quantize`), then filters them according to user-defined radii (`iV-Filter`), ensuring that only vectors meeting specific similarity criteria progress to the final ranking stage where full vector embeddings can be used to identify matches (`iV-TopK`).

Table-based Storage and Indexing. As shown in the offline component of Figure 1, VectraFlow also provides memory-resident storage for base vectors in terms of in-memory tables. VectraFlow also supports custom indexes to speed up similarity-based filtering operations (Section 3). VectraFlow's statistics collection and tuning modules are crucial for optimizing online querying: they adjust parameters and thresholds based on historical data and performance metrics (Section 6).

3 VECTOR-BASED QUERY OPERATORS

In this section, we describe VectraFlow's vector-based query operators. These include filtering, top-k, and join operations (shown in Table 1 along with general descriptions). While we discuss all the operators for completeness, our main focus is on the operations that are unique to the streaming setting and continuous processing, in particular iV-Filter, iV-TopK, and V-Join. We emphasize that, in this context, the stored *base vectors* serve as a set of *continuous query vectors*, that is, they are used to continuously search over incoming vector streams. This *inverse* search approach (hence the "*i*"s in the name of the respective operators) flips traditional vector searches on its head, necessitating new techniques. We discuss how VectraFlow supports these operators efficiently using clustering, indexing, batch processing, and incremental evaluation techniques to maintain and update results efficiently. Table 2 describes sample use cases for each vector-based operator.

3.1 iV-Filter

The *inverse vector filter* operation returns all input vectors that fall within a specified distance (i.e., radius) of each base vector (Figure 3(b)). More specifically, iV-Filter acts as a filter on the input stream vectors, returning the indices or unique IDs of the input vectors that match each base vector within each batch. Here, the radius of a base vector serves as a similarity threshold: A smaller radius retrieves the most relevant input vectors, while a larger radius captures a wider set of matches. Users may directly define the radii for base vectors,

or they can be adjusted by the system based on statistical values or offline learning processes to match a target frequency over, for example, a time period. In general, adjustable radii allow the system to meet specific requirements, either prioritizing high precision (i.e., low false positives) or high recall (i.e., low false negatives). iV-Filter resembles a range query in vector databases, but it works inversely for each base vector.

The brute-force way of performing iV-Filter requires comparing every input vector with all base vectors and then filtering out those that fall outside the similarity thresholds. Alternatively, one can also adapt an existing ANNS method to implement iV-Filter: first, we can index base vectors, and then, for each input vector, we can search for the top-k most similar vectors and finally apply filtering. However, there are a couple of challenges with this approach: (i) It is difficult to determine the optimal value of k. If k is too low, some matches will go unnoticed; if it is too high, redundant vectors will be retrieved and filtered out. (ii) ANNS indexes are typically designed for very large vector databases. Indexing a smaller set of base vectors might introduce more overhead than even the brute-force approach.

OPList. As another alternative, we introduce a new index structure to support iV-Filter: the *Overlapping Partition List* (OPList). We define the overlap of two base vectors b_x and b_y with given radii as all base vectors that match:

$$\text{dist}(b_x, b_y) \leq \text{radius}(b_x) + \text{radius}(b_y) \tag{1}$$

This inequality implies that two vectors b_x and b_y are considered to *overlap* if the sum of their radii is greater than or equal to the vector distance between them. The idea is to record the indices of base vectors that overlap with each other, leveraging the fact that the base vectors that encompass incoming data within a specific radius *must* inherently overlap with each other. VectraFlow then performs the search by first identifying an initial base vector that contains (or "matches") the input data vector, followed by a scan of the corresponding OPList to identify the base vectors that truly include the incoming data, that is, filter out the false positives.

This naïve OPList approach suffers from two problems: ❶ It is difficult to find base vectors that first match the input vector — in other words, the anchor point to initiate which OPList to scan is hard to establish. ❷ It takes $O(N^2)$, where N is the number of base vectors, time and space to build the vanilla OPList. This does not scale when the number of base vectors is large.

Centroid OPList. This baseline approach requires an OPList per base vector, so VectraFlow further optimizes it using clustering-based techniques. Specifically, VectraFlow clusters the base vectors and generates OPLists for each cluster centroid. It then assigns radii to centroids, using statistics about the member base vectors, and for each cluster records the base vectors that overlap with the radius of the centroid. In the search phase, VectraFlow assigns each input vector to the nearest centroid and scans the corresponding OPList for matches. As multiple input vectors can be assigned to the same cluster and scan the same OPList, VectraFlow batches these input vectors to enable simultaneous scanning. Consequently, VectraFlow *approximates* each input vector using the centroid it is associated with, using the centroid's radius to adjust the degree of approximation. Since all scan operations are in memory (making retrieval fast), the actual distance computation is the bottleneck in VectraFlow. Rather than performing pairwise distance computations, VectraFlow uses Equation 2 where squared norms ($\|\mathbf{v}_i\|^2$ and $\|\mathbf{v}_j\|^2$) can be computed independently and only once for each vector.

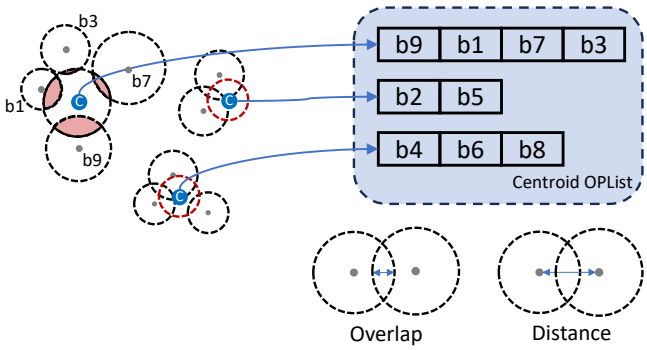

Figure 2. An example Centroid OPList sorted based on the overlap metric.

$$\text{dist}(v_i, v_j) = \|v_i\|^2 + \|v_j\|^2 - 2 \cdot v_i \cdot v_j \tag{2}$$

VectraFlow further leverages Intel MKL's `sgemm()` [13] to compute the dot product ($2 \cdot \mathbf{v}_i \cdot \mathbf{v}_j$) efficiently. This batch approach works well for our setting, as the data is already segmented into windows.

Ordered Centroid OPList. For each Centroid OPList, we can further sort and bucketize the elements in the list so that the most promising elements (that is, base vectors that are more likely to "match") are visited earlier in the scan. In addition to *distance*, which is the distance between the centroid and the base vector, we also define two additional sorting metrics:

$$\text{overlap}(b_i, c_j) = r_{b_i} + r_{c_j} - \text{dist}(b_i, c_j) \tag{3}$$

$$\exp(b_i, c_j) = e^{-\left(\frac{\text{dist}(b_i, c_j)}{\text{overlap}(b_i, c_j)}\right)} \tag{4}$$

The *overlap* metric (3) captures the geometric relationship between the radii. The *exponential decay* metric (4) is designed to capture the intuition that, as the distance between two points increases, their similarity should decrease rapidly. The objects that are closer in some feature space (e.g., semantic similarity) have much stronger relationships than those that are far apart. For vectors that have the same level of overlap, this metric can give a large penalty to those distances that are larger. Figure 2 illustrates the Centroid OPList organization, where base vectors (labeled b1-b9) are clustered. Each centroid maintains an ordered list of its associated overlapping base vectors, as described in Eq. (1),

with the ordering determined by metrics like overlap (Eq. (3)), exponential decay (Eq. (4)), or vanilla distance (Eq. (2)).
Bucketing. VectraFlow implements a bucket-based *early termination strategy* for Centroid OPList scanning. Each Centroid OPList can be divided into buckets using either equal depth partitioning (where each bucket contains the same number of vectors) or equal weight partitioning (where vectors are grouped to maintain a similar metric distribution within buckets). During the search phase, we scan these buckets sequentially and monitor the match productivity of each bucket. If we observe that subsequent buckets yield significantly fewer matches compared to previous ones (below a configurable threshold), we can terminate the scan early. This adaptive stopping mechanism provides an effective runtime trade-off between recall and performance, particularly valuable when the system faces high-rate streaming inputs. The intuition behind this approach is that since vectors in the Centroid OPList are ordered by their matching likelihood, a substantial drop in the match rate suggests that continuing to scan the remaining buckets will provide diminishing returns and may not justify the additional computational cost. By detecting significant drops in match productivity, we can avoid unnecessary computation on less promising buckets while maintaining competitive recall rates.

3.2 V-Filter

V-Filter (vector filter) performs a *similarity-based range search* for each input vector, identifying the matches within the stored vectors (Figure 3(a)). It is a special case of iV-Filter where each base vector has an identical radius; as such the indexing approach presented for iV-Filter should readily apply. This contrasts with a common implementation of V-Filter that involves a series of top-k queries on the base vectors until all matches in the target range are identified. Our solution supports range searches more directly.

3.3 iV-TopK

This *inverse vector* top-k operation continuously returns the top-k matching input vectors for each base vector and each window (Figure 3(d)). Specifically, input vectors accumulate within a window, which slides after either a certain number of input vectors are seen or some time period has passed. The unique IDs of the matched top-k input vectors for each base vector are then returned for each window. As in iV-Filter, VectraFlow uses clustering over base vectors to reduce search scopes by recording the top-k results for cluster centroids instead of for every base vector. Each base vector approximates the results to its cluster centroid, reducing the need for brute-force searches.

Simply using k to approximate the results of base vectors is often insufficient. We therefore employ a *over-retrieval factor*, m, to increase the number of k results that need to be retrieved for centroids. For each base vector, the method examines the $k * m$ closest vectors to its centroid to re-rank the top-k most similar vectors. We implemented both an incremental and a non-incremental version of the min-heap structure to keep track of the top-k results for each base vector.

There has been much work on supporting top-k queries (e.g., SAP [30]) over streams, but no prior work addressed vector data and the optimizations that VectraFlow employs.

3.4 V-TopK

V-TopK (vector top-k search) is the streaming version of the standard, widely supported, and heavily optimized vector retrieval in vector databases: For each input vector, V-TopK identifies the top-k most similar to base vectors (Figure 3(c)). Our streaming setting allows us to perform batch distance computations for a set of input vectors at the same time, thus optimizing the distance computations.

3.5 V-Join

The streaming vector join operator processes two streams of vectors using a window. For each vector on the first stream, it compares vectors on the other stream that are within the same window (Figure 3(e)). A brute-force approach to V-Join is straightforward; however, optimizations are possible by learning the distribution of input vectors and performing clustering. For example, each vector can be assigned to a centroid, allowing the join operation to be computed within each cluster in a way similar to traditional hash-based joins.

We note that a stream-to-table join on vectors was introduced in VBASE [29], where the use case was to assign a very large number of tags to documents based on their semantic similarity threshold. We support this use case with V-Filter and use V-Join to perform stream-to-stream joins.

4 LEVERAGING ALTERNATIVE VECTOR REPRESENTATIONS

We now study leveraging more efficient but lower-precision data representations to improve processing speed while still achieving high recall rates. Although this general idea has been successfully applied in vector databases (e.g., [10]), its effectiveness in a streaming environment remains uncertain, as these alternative representations must be generated in real time during the query processing phase. We investigate two scenarios: a dense binary quantization method and a sparse keyword vector technique, showcasing their advantages. Notably, we observe that the keyword-based methodology is a particularly good fit for streaming settings, when it is applicable, since it can offer substantial performance improvements by obviating the creation of embeddings for all input data when they do not already exist.
Over-retrieval & Re-ranking. The basic approach is to use a two-stage process, starting with an efficient and approximate data representation and then refining with the full

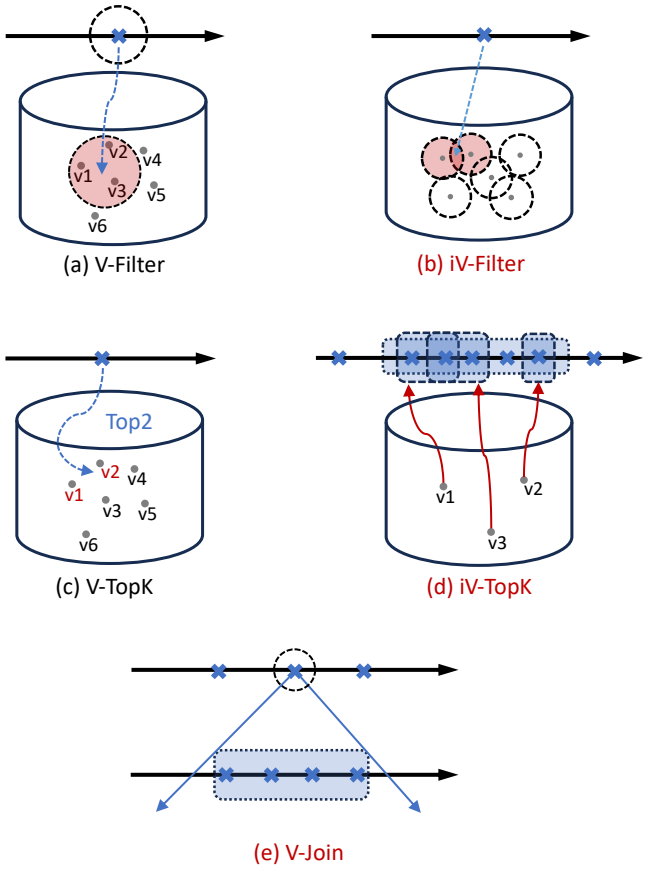

Figure 3. Vector-based query operators supported by VectraFlow. We primarily focus on (b) iV-Filter (d) iV-TopK, and (e) V-Join, continuous query operations not well-supported by vector databases. The input vector(s) are shown on the arrows, and the base vectors are shown below them in storage.

vector representation [10]. VectraFlow initially uses an efficient data representation to perform rapid similarity computations, specifically distance computations, which are crucial to narrow down candidate results to a manageable subset. At this first stage, we over-retrieve results, meaning that the resultant set size should be larger than what is ultimately needed. The rationale behind this is to account for the inherent approximation of the efficient representation, which may lead to some inaccuracies in the selection of similar vectors. By over-retrieving, we ensure that the subset includes all potentially relevant vectors, despite the approximate nature of the initial computations. We then transition to stage two, employing the full, high-fidelity vector representation. In this stage, VectraFlow performs precise computations on this smaller set of candidates. This two-stage process strikes a balance between computational efficiency and accuracy, and has the potential to significantly improve the overall performance of similarity searches over vector streams. Below, we present two examples of alternative representations.

1. **Dense: Binary Quantization (BQ).** In BQ [16], each dimension of a vector is converted into a binary value using a threshold value and then stored in a singular bit. In the quantization phase, values above a chosen threshold are assigned a 1, and values below it are assigned a 0. BQ allows us to simplify the vectors, reducing the amount of memory required to store the vectors. BQ also speeds up the time it takes to do distance computations because using less memory can lead to less cache misses and allows us to use an XOR to compute the distance between two quantized vectors, which is much faster than doing distance computations in the full vector space.
2. **Sparse: Keyword-based Representation.** Consider the case where we have text data, such as tweets, as an input stream. Here, we can use a traditional keyword-based representation where each tweet is represented as a sparse vector, with dimensions corresponding to the presence or absence of specific keywords. This approach allows for rapid and efficient filtering that quickly identifies a subset of tweets that contain relevant terms. We note that we need to have full vector representations in the second stage *only* for those items in the candidate result set identified in stage one, *not for all data items.* This optimization greatly reduces the computational burden.

Filter Cascades. We also explore the use of a *cascade* approach with alternative representations. Consider a series of binary quantizers to improve query execution. Adding an extra quantizer introduces extra overhead as it requires an additional quantization of input vectors in real-time. However, a cascading approach becomes advantageous if the new quantizer eliminates enough vectors where its added processing time is offset by the reduction in the time spent on the slower, higher-precision filter.

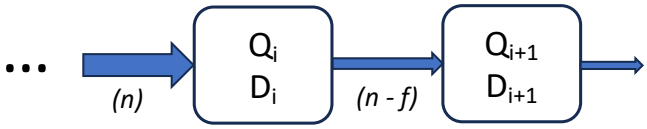

Figure 4. Adding a quantizer into the system

Equation 5 describes the number of vectors an additional quantizer must filter out, in order to maintain the same throughput:

$$f = n\left(\frac{Q_i + (D_i \cdot nb)}{Q_{i+1} + (D_{i+1} \cdot nb)}\right) \tag{5}$$

Here, f is the number of vectors filtered out; n is the number of incoming vectors; Q_i is the time it takes for quantizer i to quantize one vector; D_i is the time it takes for quantizer i to compute one vector-to-vector distance computation, and nb is the number of base vectors.

The equation reveals that the number of vectors that need to be filtered out in order to make an additional quantizer

beneficial is directly proportional to the ratio between the current quantizer and subsequent quantizer's costs. Note that the next filter could be another quantizer, entailing both quantization and distance computation costs, or it could be the full vector representation filter, in which case the quantization time is negligible, and the distance computation time contributes solely to the total cost.

5 PRELIMINARY RESULTS

We conducted a preliminary evaluation of a VectraFlow prototype that implements the search operations and optimizations discussed. We report throughput and latency figures, as well as recall values to quantify the quality of the results, as is common in vector-based processing.

Experimental Setup. We conduct our experiments on a dual-socket system equipped with two Intel(R) Xeon(R) Gold 6150 CPUs operating at 2.70 GHz with 18 physical cores and 377GB of DRAM.

Datasets: for iV-Filter, iV-TopK, V-Join, and Binary Quantization evaluation, we use the SIFT1M [11] dataset, a standard benchmark for evaluating the performance of ANNS algorithms. SIFT1M comprises 1M 128-dimensional floating-point vectors, supplemented by an additional set of 10K vectors used as base vectors. Since keyword-based representation requires raw documents for experimentation, we utilize a popular corpus from Hugging Face [5]. We shuffled and sampled portions of the documents to serve as both query and input data. When we used radii value, we generated reasonable values using a normal distribution with a mean of 300 and a standard deviation of 20 based on the average Euclidean distance between vectors. V-Join operates on two streams of shuffled SIFT1M dataset.

Streaming framework: We streamed 1M input vectors using a socket interface, utilizing 18 threads pinned to physical cores on the same CPU to prevent any hyperthreading or NUMA effects across the evaluated methods. We used independent sender and receiver sockets to simulate a realistic setting. For V-Join, we used two sender sockets to stream the input vectors, two receiver sockets to receive the vectors, and one extra receiver to listen to the join results.

Methods Settings: (i) *Centroid OPList* is trained on the entire set of training vectors to generate centroids. The number of centroids is 64, with 20 iterations for training. (ii) *Clustering iV-TopK* is similar to Centroid OPList: we train this method on the full set of training vectors, producing 128 centroids over 20 iterations. (iii) For *ANNS*, we opted to use FAISS [14], a widely-used library for in-memory vector similarity searches. We utilized FAISS's HNSW index and tuned its parameters for best performance.

iV-Filter. Figure 5 shows iV-Filter performance for the Brute Force, ANNS (HNSW), and Centroid OPList (Section 3) approaches across various batch sizes in terms of throughput (1K vectors/sec) and latency (milliseconds). For all batch sizes, Centroid OPList achieves substantially better performance than the alternatives in terms of throughput (1.1–2.5x over Brute Force and 3x–9x over HNSW) while maintaining favorable latencies. This demonstrates the benefits of Centroid OPList's task-specific index structure. The average recall values for the approaches (Centroid OPList: 0.946, Brute Force: 1, HNSW: 0.804) show that the resulting drop in recall for Centroid OPList is acceptable relative to HNSW.

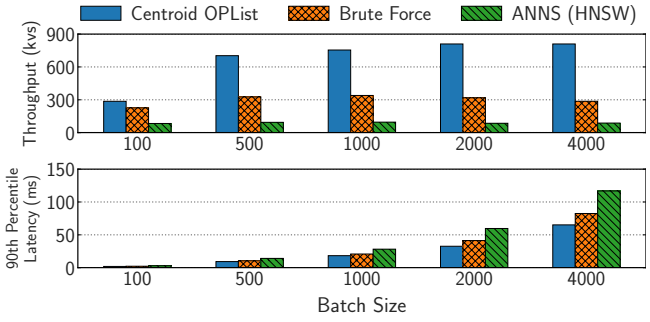

Figure 5. iV-Filter performance for Brute Force, HNSW, Centroid OPList.

We note that the Brute Force approach performs well due to its use of highly optimized batch distance computations through the `sgemm()` library [13]. We also note that HNSW, which is widely used within traditional vector databases, underperforms even compared to the Brute Force approach. One major reason is that the HNSW maintains additional data structures, most importantly min-heaps, to manage top k candidates, which is a relatively expensive operation in a main-memory setting. Using an optimal top-k knob is also challenging, as it requires robust tuning to achieve good performance. Ideally, HNSW should be built over each batch so that a top-k operation can be executed for each base vector, but building HNSW in real-time is infeasible in general.

Figure 6 demonstrates the relationship between bucket scanning depth and the number of matches using equal-depth partitioning across 64 buckets for Centroid OPList. We analyze three randomly sampled input vectors to demonstrate how matching patterns evolve during run-time, with each line marked at points where the match-growth plateaus for two, three, and four consecutive iterations (denoted as *i*). The results indicate that recall (R) stabilizes well before all buckets are exhausted. Vector 1 reaches R = 0.83 at i = 2 (bucket 24), improving to R = 0.98 at i = 4 (bucket 53). Vector 2 shows a similar progression, achieving R = 0.79, 0.86, and 0.97 at the respective plateau points. In particular, Vector 3, although it has the highest absolute match count (220), reaches R = 0.76 on its first plateau (i = 2), eventually reaching R = 0.96. The evident early curve steepness followed by distinct plateaus across all vectors validates our adaptive stopping strategy, indicating that the scanning depth can be dynamically adjusted based on consecutive plateau detection without a significant impact on the quality of the result. This

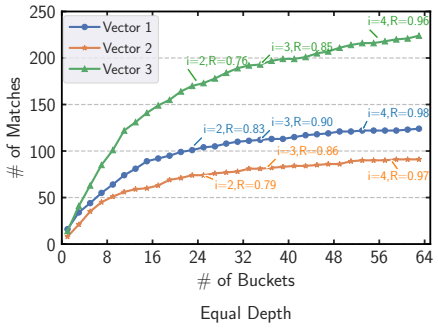

Figure 6. Early stopping during scans.

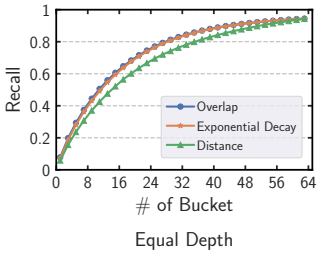

Figure 7. Sorting heuristics for Centroid OPList.

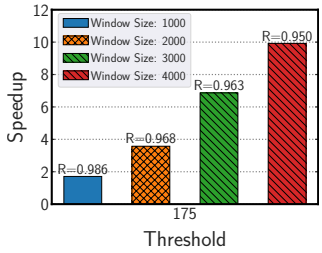

Figure 8. The streaming V-Join operator speedup over Brute Force (recall values shown above bars).

can allow the system to perform effective load shedding in real-time under high load.

Figure 7 compares different sorting metrics: Overlap (Eq. 3), Exponential Decay (Eq. 4), and Distance (Eq. 2) when Centroid OPList is built with equal-depth bucketing. The experiment measures recall performance as buckets increase from 0 to 64. The Overlap metric and Exponential Decay demonstrate very similar performance characteristics, with both achieving higher recall rates more quickly than the Distance metric as the number of buckets increases. This suggests that both Overlap and Exponential Decay metrics are more effective at prioritizing relevant vectors in earlier buckets. The naïve Distance metric's lower performance can be attributed to its simpler nature — it only considers raw distances without accounting for the geometric relationships or gradual similarity decay that the other metrics incorporate.

iV-TopK. Figure 9 shows the performance of iV-TopK, both in its incremental and non-incremental versions, compared to a Brute Force approach. The over-retrieval factor m is set to make recall above and close to 70% with window size 10,000 and slide size 1000. The results show that the incremental method consistently achieves higher throughput across all k values compared to other methods. This performance improvement is due to the use of incremental distance computations (i.e., enabled by a distance cache) and the incremental updating of the underlying min heaps that record the top-k results for each base vector across vector streams.

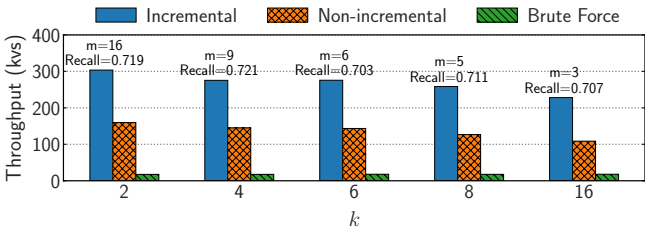

Figure 9. iV-TopK performance for three methods: Incremental, Non-incremental, and Brute Force. The recall and over-retrieval factor (m) values for each method are annotated directly above the bars.

Table 3. Binary Quantization (BQ) optimization for iV-Filter for varying numbers of base vectors (NB).

NB	10^0	10^1	10^2	10^3	10^4
Speedup	0.91x	1.06x	2.46x	3.37x	4.04x
Recall	1.00	0.99	0.93	0.87	0.83

V-Join. Figure 8 evaluates the performance of a streaming join algorithm under different window sizes (1k, 2k, 3k, and 4k vectors) at a fixed join similarity threshold value of 175. The speedup is achieved through the clustering optimization described in Section 3.5 over the brute force method. As the window size increases from 1000 to 4000, the speedup increases from approximately 2x to 10x compared to the Brute Force approach, with a slight decrease in the recall values. This significant increase in speedup with larger windows is primarily because the Brute Force method complexity grows quadratically with window size, while the clustering approach better manages this growth by limiting comparisons to within clusters.

Alternative Representations. Table 3 shows the benefits of using Binary Quantization (BQ) with the Over-Retrieval and Re-ranking technique (Section 4) to optimize iV-Filter for varying base vectors. We observe throughput increasing up to 4x using BQ with a moderate reduction in recall.

Figure 11 shows the speedup in throughput and corresponding recall values obtained when using the keyword-based representation (with TF-IDF sparse vectors) with over-retrieval (with SentenceTransformer [3] dense vectors) and re-ranking for iV-TopK evaluation for varying k values. The figure shows that we can effectively trade off performance with recall and that higher k values are more costly to support compared to the dense vector representation of all data items for each batch.

The preliminary results with the filter cascade approach using two binary quantizers showed promising performance improvements over the baseline, performing competitively with the single binary quantizer. The efficacy of the cascade approach is heavily dependent on the relative filtering characteristics of each quantizer in the cascade, as well as the

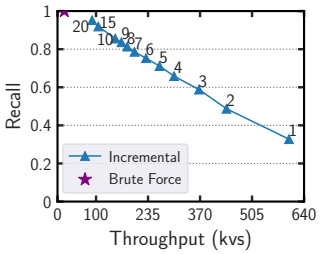

Figure 10. BQ optimization for iV-TopK for varying over-retrieval factors (k = 8).

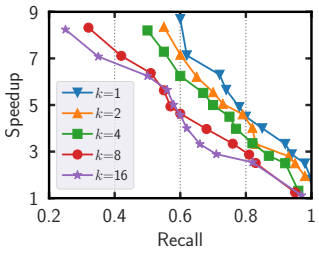

Figure 11. Keyword-based representation for iV-TopK for varied k values.

underlying data distribution. Exploring the utility of this approach comprehensively is an ongoing research direction.

6 RESEARCH DIRECTIONS

In this section, we describe our ongoing and future research directions.

Integration with LLMs and other ML Models. The combination of vector-based and model-based semantic operations offers rich opportunities for augmenting data processing with general AI capabilities. To this end, we are integrating VectraFlow with LLMs and other ML models using *model*-based operators. We have already extended the system with *prompt*-based operators, such as P-Filter(), P-Aggregate(), P-TopK(), P-Map(), and P-Join(), which are LLM counterparts of the vector-based semantic operations we discussed earlier. They invoke LLMs with text prompts that are used to evaluate selection predicates, perform ranking, and apply structured transformations, such as aggregations or maps on multimodal, unstructured data. These draw inspiration from and share similarities with other recent proposals that also describe similar structured semantic operators based on LLMs (e.g., [21], [23]).

We are also extending our set of operators further to invoke general ML models, allowing VectraFlow to incorporate classifiers, regressors, and clustering algorithms at various stages of data flow. For instance, M-Filter(), a filter operator invoking an ML model, could apply a trained classifier to filter data based on learned categories, while M-Aggregate() could use regressors to generate aggregated metrics such as trend predictions. Additionally, clustering models can be used in an M-Cluster() operation to identify inherent groupings in the data, enabling segmentation and analysis. These ML-based operations would seamlessly integrate into the data flow, similar to how our vector- and prompt-based operators are used, enhancing the ability to perform advanced, model-driven data analysis and transformation.

Continuous and Agentic Prompts. Another key area of interest is supporting continuous prompts and agentic LLM workloads that operate persistently, iterating over responses and updating them in response to new incoming data. For example, a continuous prompt implemented by an LLM-based data agent would need to continually retrieve top-k results from the incoming data stream, ensuring that the generated responses are always up-to-date and contextually relevant. This type of workload requires efficient handling of real-time data, incremental prompt evaluation, and intelligent caching mechanisms to minimize latency and resource utilization. To support such workloads effectively, we are incorporating an iteration operator that executes a loop over an operator pipeline, repeatedly applying transformations and refining data until a certain condition is met. The termination condition can be a fixed number of iterations or a custom convergence criterion that can also be evaluated by a prompt.

AI Guardrails. VectraFlow aims to mitigate AI issues such as errors, hallucinations, leakage of sensitive information and inappropriate language by embedding predefined rules and validation checks within its data flow. Such *integrity constraints* can ensure that medication doses stay within clinically safe boundaries, verify patient information accuracy across multiple data sources, and cross-check treatment recommendations against established clinical guidelines. Transparently incorporating these integrity validations into the prompt execution dataflow and applying them on the streaming outputs of language models in real-time allows VectraFlow to block inappropriate results from reaching users or applications further down the line. This approach mitigates the chance of clinical mistakes, ensuring compliance with medical best practices and regulations. To react to constraint violations, these rules also specify corrective actions, such as using a revised prompt or utilizing inference masking.

Auto Tuning and Optimizations. As we have discussed, VectraFlow offers numerous "knobs" which can dynamically determine the suitable representation and fidelity level based on a task's specific needs and its required recall rate. These knobs enable temporary adaptations to lighter-weight representations for the purpose of efficiently managing the incoming workload. Our initial experiments demonstrate that conventional optimization strategies, such as caching and prefetching, can be advantageously and seamlessly applied within vector-based data streams, especially for continuous queries. Duplicate or similar instances, such as re-tweets or images of the same vehicle or product, are prevalent in practical applications; hence, caching, reusing vector representations, and vector-to-vector distance computations prove to be effective. Efficiently merging vector-based and raw data filtering in a streaming environment remains an unresolved challenge.

More broadly, an open research direction involves developing an optimization framework for a system that supports dataflows consisting of vector-based and model-based operators. Alternative implementations for the same operator (e.g., V-Filter vs. P-Filter), as well as different implementations of

the same operator (e.g., different regression models for M-Aggregate) offer rich optimization opportunities involving trade-offs between response speed and quality.

7 CONCLUSIONS

VectraFlow aims to support a new class of vector-heavy AI workloads that involve monitoring continuous streams. By integrating techniques from both streaming and vector database domains and supporting vector-based filtering, top-k, and join operations with streaming-specific optimizations, VectraFlow addresses the unique challenges of low-latency query processing over vector streams. Experiments with our initial prototype have shown that our design and optimizations have the potential to scale well, and their behavior can be tuned to serve the needs of modern vector-based workloads.

8 ACKNOWLEDGEMENTS

We thank Shu Chen, Alex Lee, Evan Li, Deepti Raghavan, and Weili Shi for their feedback and contributions to the project.

References

[1] [n. d.]. Apache Storm. https://storm.apache.org/ Accessed: 2024-08-02.
[2] [n. d.]. Pinecone. https://www.pinecone.io/ Accessed: 2024-07-30.
[3] [n. d.]. Sentence Transformers. https://huggingface.co/sentence-transformers Accessed: 2024-08-02.
[4] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12, 2 (August 2003), 120–139. https://doi.org/10.1007/s00778-003-0095-z
[5] L. Ben Allal, A. Lozhkov, G. Penedo, T. Wolf, and L. von Werra. 2024. *SmolLM-Corpus*. https://huggingface.co/datasets/HuggingFaceTB/smollm-corpus
[6] P. Carbone, A. Katsifodimos, Kth, Sics Sweden, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38 (2015).
[7] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. arXiv:2111.08566 [cs.DB] https://arxiv.org/abs/2111.08566
[8] J. Devlin, M. Chang, K. Lee, and K. Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805
[9] E. Dilocker, B. van Luijt, B. Voorbach, M. S. Hasan, A. Rodriguez, D. A. Kulawiak, and P. Duckworth M. Antas. 2023. Weaviate. https://github.com/weaviate/weaviate
[10] M. Glass, G. Rossiello, M. F. M. Chowdhury, A. R. Naik, P. Cai, and A. Gliozzo. 2022. Re2G: Retrieve, Rerank, Generate. arXiv:2207.06300 https://arxiv.org/abs/2207.06300
[11] M. Douze H. Jégou and C. Schmid. n.d.. SIFT1M dataset. http://corpus-texmex.irisa.fr/. Accessed: March 29, 2024.
[12] K. He, X. Zhang, S. Ren, and J. Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV] https://arxiv.org/abs/1512.03385
[13] Intel Corporation. 2024. oneAPI Math Kernel Library (oneMKL) Interface. https://github.com/oneapi-src/oneMKL. Accessed: 2024-07-31.
[14] J. Johnson, M. Douze, , and H. Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.
[15] H. Jégou, M. Douze, and C. Schmid. 2010. Product quantization for nearest neighbor search. *PAMI* 33, 1 (2010), 117–128.
[16] N. Kasliwal. 2023. Binary Quantization. https://qdrant.tech/articles/binary-quantization/. Accessed on: July 31, 2024.
[17] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS* (2021).
[18] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.
[19] Yu A. Malkov and D. A. Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *PAMI* 42, 4 (2018), 824–836.
[20] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. 2003. Query Processing, Approximation, and Resource Management in a Data Stream Management System.
[21] L. Patel, S. Jha, C. Guestrin, and M. Zaharia. 2024. LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data. arXiv:2407.11418 [cs.DB] https://arxiv.org/abs/2407.11418
[22] Postgres. 2024. pgvector. https://github.com/pgvector/pgvector.
[23] S. Shankar, A. G. Parameswaran, and E. Wu. 2024. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. arXiv:2410.12189 [cs.DB] https://arxiv.org/abs/2410.12189
[24] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *arXiv preprint arXiv:2105.09613* (2021).
[25] S. J. Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *NeurIPS* 32 (2019).
[26] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, and et al. 2021. Milvus: A purpose-built vector data management system. *SIGMOD* (2021), 2614–2627.
[27] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai. 2020. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020).
[28] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang, and et al. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. *SOSP* (2023), 545–561.
[29] Q. Zhang, S. Xu, Q. Chen, G. Sui, J. Xie, Z. Cai, Y. Chen, Y. He, Y. Yang, F. Yang, and et al. 2023. {VBASE}: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. *OSDI* (2023).
[30] R. Zhu, B. Wang, X. Yang, B. Zheng, and G. Wang. 2017. SAP: Improving Continuous Top-K Queries Over Streaming Data. *IEEE Trans. on Knowl. and Data Eng.* 29, 6 (jun 2017), 1310–1328. https://doi.org/10.1109/TKDE.2017.2662236