VectorMaton: Efficient Vector Search with Pattern Constraints via an Enhanced Suffix Automaton

Haoxuan Xie
haoxuan001@e.ntu.edu.sg
Nanyang Technological University
Singapore

Siqiang Luo
siqiang.luo@ntu.edu.sg
Nanyang Technological University
Singapore

ABSTRACT

Approximate nearest neighbor search (ANNS) has become a cornerstone in modern vector database systems. Given a query vector, ANNS retrieves the closest vectors from a set of base vectors. In real-world applications, vectors are often accompanied by additional information, such as sequences or structured attributes, motivating the need for fine-grained vector search with constraints on this auxiliary data. Existing methods support attribute-based filtering or range-based filtering on categorical and numerical attributes, but they do not support pattern predicates over sequence attributes. In relational databases, predicates such as `LIKE` and `CONTAINS` are fundamental operators for filtering records based on substring patterns. As vector databases increasingly adopt SQL-style query interfaces, enabling pattern predicates over sequence attributes (e.g., texts and biological sequences) alongside vector similarity search becomes essential. In this paper, we formulate a novel problem: given a set of vectors each associated with a sequence, retrieve the nearest vectors whose sequences contain a given query pattern. To address this challenge, we propose VectorMaton, an automaton-based index that integrates pattern filtering with efficient vector search, while maintaining an index size comparable to the dataset size. Extensive experiments on real-world datasets demonstrate that VectorMaton consistently outperforms all baselines, achieving up to 10× higher query throughput at the same accuracy and up to 18× reduction in index size.

KEYWORDS

Vector database, constrained approximate nearest neighbor search

PVLDB Reference Format:
Haoxuan Xie and Siqiang Luo. VectorMaton: Efficient Vector Search with Pattern Constraints via an Enhanced Suffix Automaton. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ForwardStar/VectorMaton.

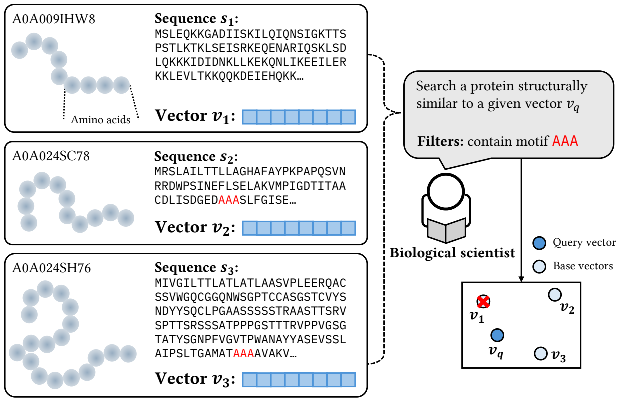

A0A009IHW8
Sequence s_1:
MSLEQKKGADIISKILQIQNSIGKTTS
PSTLKTKLSEISRKEQENARIQSKLSD
LQKKKIDIDNKLLKEKQNLIKEEILER
KKLEVLTKKQQKDEIEHQKK…
Amino acids
Vector v_1:
A0A024SC78
Sequence s_2:
MRSLAILTTLLAGHAFAYPKPAPQSVN
RRDWPSINEFLSELAKVMPIGDTITAA
CDLISDGEDAAASLFGISE…
Vector v_2:
A0A024SH76
Sequence s_3:
MIVGILTTLATLATLAASVPLEERQAC
SSVWGQCGGQNWSGPTCCASGSTCVYS
NDYYSQCLPGAASSSSSTRAASTTSRV
SPTTSRSSSATPPPGSTTTRVPPVGSG
TATYSGNPFVGVTPWANAYYASEVSSL
AIPSLTGAMATAAAAVAKV…
Vector v_3:
Search a protein structurally similar to a given vector v_q
Filters: contain motif AAA
Biological scientist
Query vector
Base vectors
v_1
v_2
v_q
v_3

Figure 1: An example of pattern-constrained ANNS in biological database.

1 INTRODUCTION

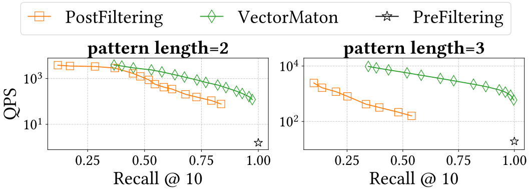
The rapid advancement of deep learning has enabled the transformation of unstructured data (e.g., texts, images and graphs) into high-dimensional vector representations [8, 21, 30, 37], giving rise to vector databases as a new data management paradigm. Unlike traditional database queries that rely on exact matching or relational predicates, vector search focuses on retrieving the most similar vectors according to a distance or similarity metric. However, exact nearest neighbor search in high-dimensional spaces is computationally prohibitive due to the curse of dimensionality [22]. As a result, approximate nearest neighbor search (ANNS) has emerged as the dominant approach for scalable vector retrieval and has attracted significant attention from both academia and industry [5, 16, 19, 24, 26, 33, 34, 43].

Motivation. In many real-world applications, vector embeddings are accompanied by associated sequences or other structured attributes. For example, in scientific document retrieval, the content of a paper can be embedded into a vector representation, while the paper title is naturally modeled as a sequence associated with the vector [40]. While recent studies have explored constrained approximate nearest neighbor search through *attribute-in* filtering [20, 31, 32, 36] and range-based filtering [25, 41, 46–48, 50], these techniques primarily focus on numeric or categorical constraints. In contrast, **pattern constraints over sequences**, such as substring or motif matching, remain largely unexplored in the context of vector search. Given a query vector v and a pattern p, *ANNS with pattern constraints* aims to retrieve the nearest vectors whose associated sequences contain the pattern p. Such queries naturally arise in a wide range of real-world applications, including but not limited to the following examples:

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Methods	Short pattern		Long pattern		Index Size
	Efficiency	Quality	Efficiency	Quality	
PreFiltering	Low	Optimal	Moderate	Optimal	Small
PostFiltering	Moderate	Moderate	Moderate	Low	Small
OptQuery	High	High	High	High	$O(m^2)$
VectorMaton	High	High	High	High	$O(m^{1.5})$

(a) Comparison of different methods (m is the total sequence length).



(b) Impact of pattern length.

Figure 2: Summary of challenges and methods.

- **Supporting LIKE and CONTAINS predicates in vector SQL.** In relational databases, predicates such as LIKE and CONTAINS are fundamental building blocks for SQL queries. These predicates largely involve substring queries, i.e., retrieving data that contains a specific pattern string. As vector databases increasingly adopt SQL-like query interfaces, supporting such pattern predicates alongside vector similarity search becomes essential. For example, in a document database, users may not only want to retrieve documents with similar embeddings, but also require the documents to contain a specific keyword. While systems such as ElasticSearch [2] and PostgreSQL [1] have added support for hybrid vector similarity and keyword or substring queries, they generally rely on either a filter-then-search or a search-then-filter execution strategy. These approaches treat vector search and pattern filtering as separate components and do not provide a unified index structure, leading to low query efficiency and quality. Efficiently integrating pattern constraints with approximate nearest neighbor search remains an open challenge.
- **Similar protein retrieval with motif matching.** In biological databases, proteins and genomic sequences are increasingly represented by learned vector embeddings that capture functional or structural similarity (e.g., AlphaFold [27], ProtBERT [13]). For instance, the SwissProt [28] dataset contains proteins with a total sequence length reaching billions of symbols. These vector embeddings are inherently tied to their underlying sequences, where patterns such as conserved motifs or binding sites play a critical role. For instance, a biologist may wish to retrieve proteins that are functionally similar to a query protein (i.e., vector similarity) while containing a specific amino-acid motif (e.g., $\texttt{Cys}-\texttt{X}_2-\texttt{Cys}$) that is essential for metal binding [7]. Such queries require jointly enforcing vector similarity and pattern constraints over sequences. Figure 1 shows a toy example of this scenario.

Challenge: Pattern constraints exhibit various selectivity and demand specialized index designs. In constrained vector search, different filtering conditions induce varying selectivity. Existing approaches to filtered approximate nearest neighbor search can be classified into three categories: *pre-filtering*, *post-filtering*, and *joint filtering* [31]. Pre-filtering first identifies qualifying vectors using an external index on vector-associated attributes and then performs a brute-force search over the filtered subset. However, when filter selectivity is high (i.e., when a large fraction of vectors qualify), the filtered subset becomes large, leading to substantial query overhead and poor scalability. Post-filtering, in contrast, first performs ANNS on the full vector index and subsequently filters the retrieved candidates. This approach is effective when selectivity is high enough that most nearest neighbors satisfy the constraint. However, under low-selectivity conditions, a large portion of retrieved candidates are discarded, resulting in degraded recall or increased query cost. Unfortunately, **pattern constraints over sequences exhibit various selectivity**. Complex or long patterns typically match only a small number of sequences, leading to low selectivity, whereas short or simple patterns may match a large fraction of the dataset. As a result, neither pre-filtering nor post-filtering can robustly handle pattern-constrained queries across different selectivity. Joint filtering aims to integrate filtering conditions directly into the ANNS process, but typically requires specialized index designs tailored to specific types of constraints. Existing joint-filtering indices primarily target *attribute-in* and range-based predicates over categorical or numerical attributes. These methods are designed to support operators such as equality, subset inclusion, and numerical comparison. However, such techniques cannot be directly extended to pattern-based predicates over sequence attributes. Efficiently supporting *pattern constraints over sequences* within ANNS remains largely unexplored and poses a significant open challenge.

Table 2(a) shows the undesirable query trade-offs of PreFiltering and PostFiltering approaches. For instance, as illustrated in Figure 2(b), on SIFT dataset [24] with synthetic sequences, as the query pattern length grows, the recall of PostFiltering shrinks significantly, and PreFiltering remains relatively low query efficiency.

Our solution. Our goal is to design an index that supports *ANNS with pattern constraints*, while providing (i) index size comparable to the dataset size and (ii) query efficiency and result quality comparable to unconstrained vector search.

To optimize query efficiency and avoid unnecessary computations, a straightforward approach (OptQuery) is to query on the vector index containing only those vectors satisfying the given pattern constraint. This requires constructing the corresponding index for every possible pattern. However, this strategy is highly space consuming. Let m denote the total length of all sequences. In the worst case, the number of distinct substrings across all sequences is $O(m^2)$, leading to unaffordable space cost. While inverted file indexes [12, 29] reduce space consumption by avoiding materialization of all substrings, its query processing requires merging multiple posting lists that may involve duplicated vectors. The query cost grows as the query pattern size increases since more posting lists would be involved, and finally resulting in query overhead.

To overcome these limitations, we present VectorMaton, an automaton-based vector index for pattern-constrained ANNS. The key insight is that many patterns exhibit identical occurrence behavior across sequences. Such patterns can therefore be grouped and share the same index. For example, consider two sequences "ab", "aab". Then patterns "b" and "ab" are considered equivalent since they both occur at the end of each sequence. To realize this idea, we borrow the ideas from formal language theory, and leverage the *suffix automaton* (SAM) [10, 38]. While SAM has traditionally been used for substring matching, they are not designed for pattern-constrained vector search. In this work, we extend this technique to both index a collection of sequences and integrate it with approximate nearest neighbor search. Since each pattern can still be directly associated with the vectors whose sequences contain it, the query efficiency remains comparable to OptQuery. Our resulting structure achieves efficient query processing while maintaining a worst-case space complexity of $O(m^{1.5})$, where m is the total sequence length. We further propose two space-saving strategies to reuse index across states to reduce redundancy and selectively construct index based on pattern selectivity to adapt to different query conditions. Together, these strategies enable a unified and scalable solution for pattern-constrained ANNS.

Empirically, we observe that the index size of VectorMaton grows near-linear with respect to the total sequence length, while delivering substantial query performance improvements over pre-filtering and post-filtering baselines. This scalability enables VectorMaton to handle datasets with up to billion-scale sequence lengths. Experimental results show VectorMaton achieves superior query efficiency and recall as well as significantly reduced index size when comparing with the proposed baselines and existing vector search engines pgvector [1] and ElasticSearch [2].

Contributions. Our contributions are summarized as follows.

- We formally define the problem of *ANNS with pattern constraints* that, to the best of our knowledge, has not been systematically studied in prior work.
- We propose VectorMaton, a novel index built on an enhanced suffix automaton that enables efficient pattern-constrained ANNS while maintaining empirically near-linear space complexity.
- We conduct extensive experimental evaluations on diverse real-world datasets, demonstrating that VectorMaton consistently outperforms existing baselines, achieving up to 10× higher query throughput at comparable accuracy and up to 18× reduction in index size.

2 PRELIMINARIES

2.1 Problem definition

Let $D = \{(v_1, s_1), (v_2, s_2), \cdots, (v_n, s_n)\}$ be a dataset of size n, where v_i denotes the i-th vector and s_i denotes the sequence associated with the i-th vector. A sequence s can be a string, protein sequence, DNA sequence, etc. Let V_p be the set of vectors whose associated sequences contain a pattern p. Pattern p is also a sequence, and a sequence s contains p if and only if p occurs as a consecutive subsequence in s. Our task is defined as follows.

Definition 1 (ANNS with pattern constraints). *Given a query vector v_q, a query pattern p, and an integer k, find a set of vectors V_o approximating the top-k vectors in V_p such that their distances are closest to v_q (denoted as $V_{k,p}$), and maximize both the answer quality (i.e., $|V_o \cap V_{k,p}|$) and query efficiency.*

Example 1. *In Figure 1, each protein is embedded into a structural vector associated with its underlying sequence. The biological scientist asks which protein is most structurally similar to a given vector v_q and contains motif* AAA. *Then $p =$ AAA, $k = 1$ and $V_p = \{v_2, v_3\}$. Since v_3 is closer to v_p than v_2, the resulting $V_{k,p}$ should be $\{v_3\}$.*

Notation	Description
n	the size of the dataset
m	the total length of all sequences
D	the dataset $\{(v_1, s_1), \cdots, (v_n, s_n)\}$
V	base vectors $\{v_1, \cdots, v_n\}$ in the dataset
S	sequences $\{s_1, \cdots, s_n\}$ associated with the vectors
v_q	query vector
p	query pattern
V_p	base vectors whose associated sequences contain p
$V_{k,p}$	top-k vectors in V_p closest to the query vector
V_o	approximated vector set of $V_{k,p}$
M	the maximum degree in HNSW graph
ef_con	capacity of the candidate list in index construction
ef_search	capacity of the candidate list in query processing
T	threshold of constructing HNSW graph in VectorMaton

Table 1: Summary of notations.

2.2 Hierarchical Navigable Small Worlds

The *Hierarchical Navigable Small Worlds* (HNSW) graph [34] is a widely adopted graph-based index for ANNS in high-dimensional vector spaces. HNSW represents vectors as nodes in a multi-layer graph, where edges are more likely to connect vectors that are close in the embedding space. We integrate HNSW into our automaton structure to achieve strong empirical query efficiency and accuracy. In the following, we provide a brief overview of HNSW.

HNSW structure. HNSW organizes vectors into a multi-layer graph. The bottom layer contains all vectors and provides dense local connections among nearby vectors, while higher layers contain progressively fewer nodes and act as long-range "highways" that enable rapid navigation toward relevant regions of the space. Each vector is assigned a maximum layer according to a geometric distribution, ensuring that the expected number of nodes per layer decreases exponentially with increasing layer height. The maximum degree of a node is limited by the hyperparameter M. Consequently, the overall space complexity of HNSW is linear in the number of indexed vectors.

Query processing in HNSW. To answer a query of top-k nearest vectors, HNSW performs a greedy search starting from an entry point at the top layer, iteratively moving to neighboring nodes that are closer to the query vector until we find the approximated nearest one, and uses it as the entry node of the next layer. After descending to the bottom layer, HNSW executes a bounded best-first search to find a set of nearest vectors and selects the top-k as the results. The trade-off between query latency and recall is controlled by the parameter *ef_search*, which specifies the maximum size of the candidate list maintained during the bottom-layer search and is similar to *ef_con*. Larger *ef_search* increase recall at the cost of higher query latency, while smaller *ef_search* favor faster queries with reduced accuracy.

2.3 Suffix automaton

The *suffix automaton* (SAM) [10, 11] is a compact deterministic finite automaton that represents all suffixes of a given sequence. The

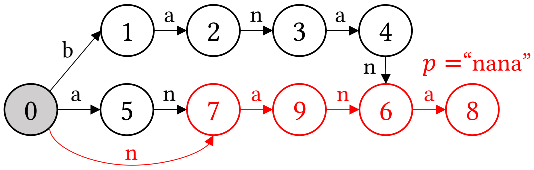


Figure 3: The SAM structure of the sequence "banana".

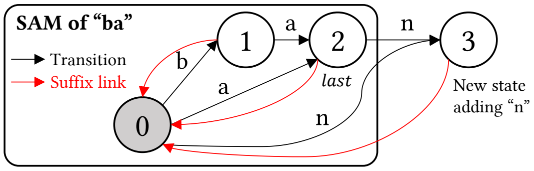


Figure 4: An example of incrementally constructing SAM.

general suffix automaton (GSA) further extends SAM to handle a collection of multiple sequences. Since a `CONTAINS` predicate checks whether a pattern p appears as a substring, it can be equivalently viewed as checking whether p is a prefix of at least one suffix of the sequence. Therefore, both SAM and GSA naturally support efficient verification of whether a pattern is contained in a single sequence or in a sequence collection.

Overview of SAM. A suffix automaton (SAM) is represented as a directed acyclic graph whose nodes, called *states*, compactly represent groups of substrings with shared characteristics. Directed edges, referred to as *transitions*, are labeled with single characters and indicate how substrings can be extended. The automaton has a unique source node called the *initial state*, from which all other states are reachable. There is a one-to-one mapping between substrings of the sequence and transition paths starting from the initial state. While different substrings may share prefixes or terminate at the same state, the automaton compactly merges these overlaps. This design enables SAM to represent all substrings in linear space, while allowing pattern checks to be performed by a single left-to-right traversal of the automaton.

Example 2. *Figure 3 shows the suffix automaton constructed for the sequence "banana", where state 0 is the initial state. Given the pattern p = "nana", we start from state 0 and follow transitions labeled* `n`, `a`, `n`, *and* `a`. *Since the traversal successfully reaches a state after consuming all characters of p, the pattern "nana" is contained in the sequence "banana".*

SAM construction. The suffix automaton of a sequence is built incrementally with one character at a time. Let *last* denote the most recently created state corresponding to the full prefix constructed so far. When a new character c arrives, we create a new state and add a transition labeled c from *last*. This extends the suffixes in *last* to end at the new position.

We then follow a small chain of *suffix links* [10] from *last* to update earlier states that should also gain a transition labeled c. These suffix links connect states representing substrings with shared suffixes and allow the automaton to remain minimal. This incremental procedure constructs the SAM in $O(|s|)$ time for a sequence s.

Example 3. *Figure 4 illustrates a partial construction of the suffix automaton for the sequence "banana". After processing the prefix "ba", state 2 is the current last state and represents the suffixes "ba" and "a". When the next character 'n' is added, these suffixes are extended to "ban" and "an", which are captured by a new state 3. In addition, the single-character suffix "n" is also introduced. To account for this new suffix, the construction follows the suffix link of state 2 and adds a transition labeled 'n' from the initial state 0 to state 3. This step ensures that all new suffixes ending with 'n' are properly represented in the automaton.*

3 PROPOSED BASELINES

3.1 Optimal query baseline for HNSW

We begin by addressing a fundamental question in solving the pattern-constrained ANNS problem in Definition 1: *under the HNSW query framework, what is the best achievable trade-off between query efficiency and query accuracy?*

An intuitive solution is obtained by decoupling pattern filtering from vector search. Specifically, let $V_p \subseteq V$ denote the subset of items whose associated sequences satisfy the pattern constraint p. We first identify V_p, and then build and query an HNSW index over *only* these qualified items. Running a standard HNSW search on this filtered graph avoids any exploration of disqualified items while preserving the same search procedure as unconstrained HNSW on V_p.

Algorithm 1: OptQuery

```
Function Build($n, V, S$):
    $H \leftarrow$ a hashmap that maps patterns to HNSW graphs;
    for $i \leftarrow 1, 2, \cdots n$ do
        for $j \leftarrow 1, 2, \cdots, |s_i|$ do
            for $k \leftarrow j, j+1, \cdots, |s_i|$ do
                Extract substring $s_i[j..k]$ as a pattern $p$;
                Insert $v_i$ to the graph of $H(p)$;

Function Query($v_q, p, k, ef\_search$):
    if $H(p)$ does not exist then return $\emptyset$ ;
    Search in the HNSW graph $H(p)$ with parameters
     $v_q, k, ef\_search$ and return $k$ results;
```

Algorithm 1 presents a straightforward baseline that achieves this optimal query behavior under the HNSW framework. To construct the index, we enumerate all patterns that appear in the dataset and build an independent HNSW graph for each pattern p over the filtered vector set V_p. Specifically, for each sequence s_i, we extract all of its substrings and insert the associated vector v_i into the HNSW graph corresponding to each substring pattern. At query time, we first check whether the queried pattern exists in the index. If so, we directly perform an HNSW search on the graph associated with that pattern; otherwise, no feasible result exists and the query terminates immediately.

Theorem 1. *Algorithm 1 requires $O(m^2)$ index space.*

3.2 PreFiltering and PostFiltering

Algorithm 2 shows the query processes of *PreFiltering* and *PostFiltering*. Given the query vector v_q, query pattern p and integer

k, PreFiltering first identifies candidate vectors V_p and then performs a brute-force search over this filtered subset to find k nearest neighbors to v_q. This approach guarantees exact results but often incurs significantly higher query time due to the exhaustive search. PostFiltering, in contrast, performs a standard ANN search over the full HNSW index and subsequently filters the results to retain only those satisfying the pattern constraint. While PostFiltering achieves query efficiency comparable to HNSW-based methods such as the optimal baseline, it may sacrifice recall because some relevant vectors can be missed during the initial ANN search.

Algorithm 2: Pre/Post-Filtering

```
Function PreFiltering($v_q, p, k$):
    Identify $V_p$ from an index of $S$;
    Return $k$ closest vectors in $V_p$;

Function PostFiltering($v_q, p, k, ef\_search$):
    Search in the HNSW graph $H(p)$ with parameters
      $v_q, k, ef\_search$ and return $ef\_search$ results;
    Filter and return $k$ closest vectors whose associated
      sequences contain $p$;
```

3.3 Open challenges and opportunities

Figure 5 summarizes the three baseline approaches. PreFiltering first identifies all pattern-matched vectors and then performs vector search over the filtered subset. When the pattern selectivity is low, this leads to examining a large number of candidates and consequently low query efficiency. PostFiltering, on the other hand, performs ANN search before applying the pattern constraint. As a result, many retrieved candidates may not satisfy the pattern, which degrades recall or requires expanding the search space to compensate.

OptQuery achieves the best efficiency-recall trade-off by constructing vector indexes for pattern-matched subsets, ensuring that only relevant vectors are examined during search. However, this strategy incurs a quadratic space cost, as it potentially requires building indexes for $O(m^2)$ distinct patterns, making it impractical for large-scale datasets.

The key challenge is therefore how to retain the query performance of OptQuery while substantially reducing its index size. Our central insight is that although there can be $O(m^2)$ possible substrings in a sequence collection, many of them share identical occurrence behavior and can be grouped into $O(m)$ equivalence classes via an automaton structure. By constructing indexes over these equivalence classes rather than individual patterns, we can significantly reduce the index size while preserving efficient query processing.

4 VECTORMATON

We now present VectorMaton, an index designed to support efficient approximate nearest neighbor search under pattern constraints. We first describe the overall index structure in Section 4.1. As illustrated in Figure 6, VectorMaton consists of an enhanced suffix automaton (ESAM) that indexes all substrings of the sequences, with each state augmented by either a HNSW graph or a raw vector ID set. To process a query pattern p, we traverse the automaton along edges labeled with the symbols of p until reaching the corresponding state. From this state, we perform a k-nearest neighbor search over the vectors in its own index as well as the index inherited from one of its successor state. The results from these searches are then merged to produce the final top-k nearest neighbors.

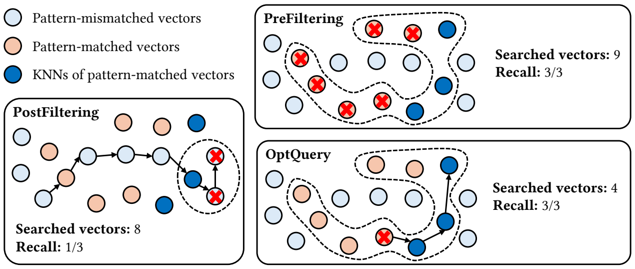


Figure 5: Comparison of different baselines.

We then detail the query processing and index construction procedures in Sections 4.2 and 4.3, respectively. We show that VectorMaton achieves an index size substantially smaller than the optimal query baseline, while providing comparable query efficiency and result quality.

4.1 Index structure

Figure 6 illustrates an overview of the VectorMaton index structure. We propose an enhanced suffix automaton, termed as ESAM, which consists of states and transitions. Each state is associated with a corresponding state index either as an HNSW graph or as a raw vector ID set.

ESAM states. The proposed VectorMaton index constructs an enhanced suffix automaton over all sequences in the collection. Recall that the general suffix automaton (GSA) [38] compactly represents all substrings of a sequence collection and supports efficient *pattern existence* queries (see Section 2.3). However, GSA cannot be directly extended to determine *which sequences* contain a given pattern. This limitation arises because a single automaton state may represent multiple substrings whose occurrences span different subsets of sequences (see Example 4).

Example 4. *Figure 7 illustrates a GSA built over the sequences* ac, acab, *and* acba. *Consider two patterns,* p_1 = cba *and* p_2 = cab. *Although the GSA correctly determines the existence of both patterns by ending in the same state (state 4), their occurrence sets differ:* p_1 *occurs only in sequence* acba, *whereas* p_2 *occurs only in sequence* acab. *Consequently, the GSA cannot distinguish which sequences contain a given pattern.*

To support pattern-constrained queries, ESAM redefines the semantics of automaton states: each state corresponds to a set of *equivalent patterns* and is associated with the set of vector IDs whose underlying sequences contain all patterns represented by that state. This refinement ensures the correctness of answering our problem. In the following, we introduce several definitions and lemmas to show that the number of states in VectorMaton is bounded by $O(m)$, and that the total size of all associated ID sets

ID	Sequence	Vector
0	banana	(1.0, 2.0)
1	nana	(3.0, 4.0)
2	na	(5.0, 6.0)
3	a	(7.0, 8.0)

Query info:
p =“na”, k = 1, v_q = (4.5, 5.0)

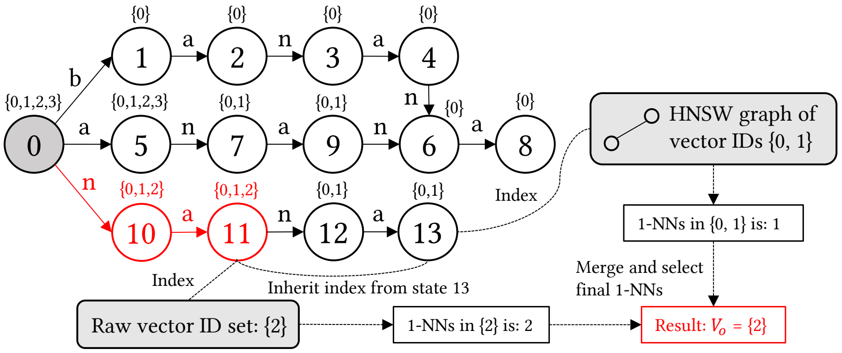


Figure 6: An example VectorMaton index and its query process.

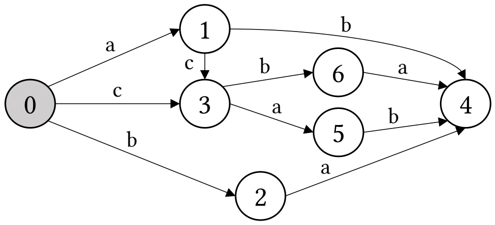


Figure 7: The general suffix automaton [38] of sequences ac, acab and acba.

is bounded by $O(m^{1.5})$. **For space limits, we defer all proofs to the appendix of our extended version.**

Definition 2 (Position list (poslist)). *Given a pattern p that appears in the sequence collection, we define its* position list, *denoted as poslist(p), as the set of pairs (id, pos), where id is the identifier of a sequence containing p, and pos is the end position of an occurrence of p in that sequence.*

Example 5. *Consider the pattern p =* anan, *which corresponds to state 6 in Figure 6. The pattern occurs only once, in the sequence* banana *with sequence ID 1, ending at position 4 of the sequence. Therefore, poslist$(p) = \{(1, 4)\}$.*

Definition 3 (Equivalence class). *Two patterns p_1 and p_2 are considered equivalent if and only if poslist(p_1) = poslist(p_2).*

Definition 4 (Maximal pattern). *Given a pattern p that appears in the sequence collection, we say p is maximal if and only if p is longest within its equivalence class.*

Example 6. *In Figure 6, state 6 corresponds to two equivalent patterns* anan, banan, *as their poslists are equal. The maximal pattern in state 6 is* banan.

As a result, the patterns of each equivalence class corresponds to only one set of vector IDs, which is the union of IDs in its position list. Moreover, it is straightforward to observe that each equivalence class contains exactly one maximal pattern. In particular, if two maximal patterns have the same length and are equivalent, their occurrences in the sequences are thus identical, and extracting the pattern from their occurrences yields the same resulting pattern.

Since each state in ESAM represents a set of equivalent patterns, the total number of states is equal to the total number of maximal patterns in the sequence collection. Consequently, bounding the number of maximal patterns directly yields a bound on the number of states.

Lemma 1. *Let S be a collection of sequences with total length m. The number of maximal patterns in S is bounded by $O(m)$. Therefore, the number of states in ESAM is also bounded by $O(m)$.*

The linear bound on the number of states is a fundamental property of suffix automata. Lemma 1 demonstrates that our ESAM preserves this property, avoiding the potential for substantial space blowup. In the following, we further establish an upper bound on the total size of the ID sets associated with all states.

Lemma 2. *The total size of all associated ID sets in ESAM is bounded by $O(m^{1.5})$, where m is the total length of all sequences.*

ESAM transitions. As we have defined the states of ESAM, we next describe how transitions between states are established. As discussed in Section 2.3, transitions in classical suffix automata correspond to extending a pattern by one character. This principle naturally generalizes to our multi-sequence setting. Specifically, the following lemma formalizes the correctness of the resulting transition construction.

Lemma 3. *Let A and B be two equivalence classes, corresponding to states A and B. If there exist patterns $p_A \in A$ and $p_B \in B$ such that $p_B = p_A \cdot c$ for some character c, then for every other pattern $p'_A \in A$, the extended pattern $p'_A \cdot c$ also belongs to B. Consequently, a transition labeled c can be established from state A to state B.*

Since each transition strictly increases all position values *pos* by one, the states and transitions in ESAM form a directed acyclic graph (DAG). Moreover, for any transition $i \longrightarrow j$, the associated vector ID sets satisfy $V_j \subseteq V_i$. This monotonicity property enables effective reuse of index information between a state and its descendant states.

State index. Although each state in ESAM is associated with a vector ID set, as illustrated in Figure 6, these sets exhibit substantial overlap across states. Moreover, different states correspond to patterns with varying selectivity. Consequently, constructing an independent HNSW index for every state would be highly space-inefficient. To address this issue, we propose two space-optimization strategies: **(1) an index reuse strategy** and **(2) a skip-build strategy**.

For index reuse strategy, let V_j be the associated ID set over state j, and I_j be the ID set to index associated with state j that contains $|I_j|$ vectors. Among all descendants of j (i.e., states reachable from j via a directed path), let k be the descendant with the largest indexed ID set. Instead of building I_j over the entire set V_j, we construct I_j only over the difference set $V_j \setminus I_k$. During query processing on state j, we independently retrieve the k nearest neighbors from both I_j and I_k, and then merge the two result sets. This reuse strategy achieves the same query accuracy as an index built directly over V_j while reduces the total space consumption.

For skip-build strategy, as states may have widely varying ID set sizes due to differences in pattern selectivity, constructing a full HNSW graph for every state is often unnecessary. Under the skip-build strategy, when the size of a state's ID set is below a threshold T, we store only the raw vector ID set and perform brute-force search when querying. This approach is generally more efficient than graph-based search for very small datasets. For example, when the search parameter ef_search exceeds the graph size, HNSW search may degenerate into an exhaustive traversal of all vertices, which is typically slower than direct brute-force evaluation.

4.2 Query processing

To answer a query, we first leverage the ESAM to locate the state corresponding to the query pattern, which is associated with the filtered vector set. We then perform nearest neighbor search over the index associated with this state and, if applicable, its inherited index.

Algorithm 3 presents the details. The algorithm begins at the initial state 0. It sequentially scans the query pattern and follows the corresponding transitions in the automaton until either the entire pattern is consumed or no valid transition exists (lines 24-26). Let *cur* denote the resulting state. The algorithm then retrieves the index associated with state *cur* as well as the index of its inherited state, if such a state exists (lines 27-28). Finally, it independently performs k-nearest neighbor searches on the two indexes and merges the results to obtain the final answer set (lines 29-30).

As VectorMaton adopts a skip-build strategy that adaptively constructs each index either as an HNSW graph or as a raw vector ID set, the query processing is different on the two types of index. If an index is implemented as an HNSW graph, we apply the standard HNSW search procedure with parameter ef_search. Otherwise, we perform a brute-force scan over all associated vectors and select the top-k nearest neighbors.

As the indexes of a state and its inherited state together form an exact cover of the state's associated vector set, merging the top-k results returned from these two indexes is **lossless**. Consequently, the resulting recall is comparable to that obtained by querying over the entire vector set.

4.3 Index construction

In this section, we introduce how to construct the ESAM and state index, respectively. Additionally, we provide a parallel construction approach that accelerates the index construction process.

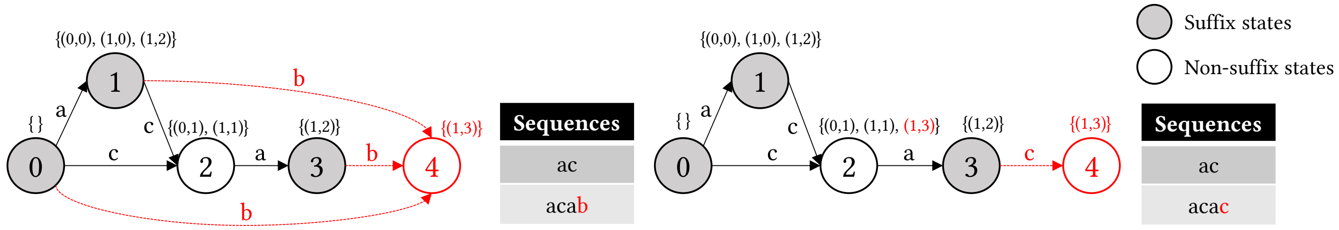
Algorithm 3: VectorMaton

```
1  Function Build($V, S, T$):
       // Build automaton via suffix-link extension
2      Initialize automaton with root 0;
3      foreach sequence $s_i \in S$ do
4          $last \leftarrow 0$;
5          foreach symbol $c$ in $s_i$ do
6              Create new state $cur$;
7              Follow suffix links from $last$ and: (i) connect
                 missing $c$-transitions to $cur$; (ii) stop at first
                 state $A$ with $c$-transition;
8              if no such state exists then link($cur$) $\leftarrow$ 0 ;
9              else
10                 $B \leftarrow$ the $c$-successor of $A$;
11                 if $|B| = |A| + 1$ then link($cur$) $\leftarrow B$ ;
12                 else
13                     Clone $B$ as $B'$ and redirect transitions;
14                     link($cur$) $\leftarrow B'$;
15             $last \leftarrow cur$;
               // ID propagation
16             Follow suffix link from $last$ and (1) add ID $i$ to
                 their ID set; (2) stop at first state that already
                 contains $i$;
       // Build state indexes
17     foreach state $u$ in reverse topological order do
18         $v \leftarrow$ descendant with largest indexed set;
19         $base(u) \leftarrow ID(u)$ excluding the indexed set of $v$;
20         if $|base(u)| < T$ then Store $I(u) \leftarrow$ raw ID set of
             $base(u)$ ;
21         else Build index $I(u) \leftarrow$ HNSW over $base(u)$ ;

22 Function Query($v_q, p, k, ef\_search$):
23     $cur \leftarrow$ the state 0;
24     for $i \leftarrow 0, 1, \cdots, |p| - 1$ do
25         if no transition $p_i$ in $cur$ then return $\emptyset$ ;
26         $cur \leftarrow$ next state of transition $p_i$;
27     $next \leftarrow$ the state that $cur$ inherits index from;
28     $I(cur), I(next) \leftarrow$ the index of $cur, next$;
29     $res_1, res_2 \leftarrow$ the $k$-NN results of $v_q$ from $I(cur), I(next)$;
30     return the top-$k$ closest vector IDs in $res_1 \cup res_2$;
```

ESAM construction. For ESAM construction, we propose a generalized variant of the SAM construction described in Section 2.3. We process the sequence collection incrementally, and for each sequence, we process its symbols in order. Next, we aim to identify new states and transitions upon processing each symbol.

The key question of is **which old states may give rise to new states**. Since symbols are processed incrementally, each newly read symbol extends all suffixes of the currently processed sequence by one character, potentially forming new equivalence classes. Consequently, the construction must maintain the states that correspond to suffixes of the current sequence.



(a) Case 1: no transition in suffix states

(b) Case 2: has transitions in suffix states

Figure 8: An example of suffix states and their extension process.

Definition 5 (Suffix state). *Consider the sequence s_i currently being processed and suppose the algorithm has processed position j. Upon reading the next symbol (at p osition $j + 1$), a state (i.e., an equivalence class) is called a* suffix state *if and only if (1) its position list is empty, or (2) its position list contains (i, j).*

Example 7. *Figure 8(a) shows an example of suffix states. Before extending symbol b, states 0, 1, 3 are suffix states as their position list is either empty or contains $(1, 2)$.*

The next question is **how to extend these suffix states to new states**. By Lemma 3, for a suffix state A with position list $poslist(A) = \{(x_k, y_k)\}$, extending A by the next symbol $c = s_i[j + 1]$ yields a state whose position list is $\left\{(x_k, y_k + 1)|s_{x_k}[y_k + 1] = c\right\}$. Consequently, if a suffix state A has no outgoing transition labeled c prior to the extension, none of its existing occurrences can be extended by c, and the extension introduces only the new occurrence ending at $(i, j + 1)$. Therefore, all such suffix states point to the same newly created state whose position list is exactly $\{(i, j + 1)\}$.

Lemma 4. *Let $c = s_i[j + 1]$ be the next symbol to be processed. For all suffix states A that do not have an outgoing transition labeled c, extending A by c leads to the same state.*

Example 8. *Figure 8(a) shows an example when suffix states have no corresponding transition 'b'. We simply create a new state and connect these suffix states to the new state with position list $\{(1, 3)\}$.*

Conversely, suppose a suffix state A already has an outgoing transition labeled c prior to the extension, and let B denote its target state. We examine whether all occurrences represented by B remain valid after introducing the new occurrence $(i, j+1)$. If adding $(i, j + 1)$ preserves the equivalence class represented by B, then no structural change is required: B simply augments its position list with $(i, j + 1)$ and continues to represent the updated class. Otherwise, the extension violates the equivalence of occurrences in B. In this case, we split the class by creating a copy B'. The original state B retains the occurrences that cannot be extended to $(i, j + 1)$, while B' represents the refined equivalence class that includes $(i, j + 1)$, and then replace connection $A \longrightarrow B$ by $A \longrightarrow B'$.

Example 9. *Figure 8(b) illustrates the case where suffix states 0 and 1 both have an outgoing transition labeled 'c'. Since all occurrences represented by state 2 remain consistent after adding the new occurrence $(1, 3)$, no splitting is required and state 2 simply updates its position list.*

However, directly scanning all suffix states would be inefficient. Thus we utilize the *suffix link* mechanism [10] to provides an efficient way to process them. While the concept is primarily designed for single-sequence scenario (see Section 2.3), it can be naturally generalized to our multi-sequence automaton. For space reasons, the detailed correctness proof is deferred to the appendix of the extended version. In the following, we outline the automaton construction process via suffix links. The detailed procedure is illustrated in Algorithm 3.

Specifically, the suffix states form a chain ordered by decreasing maximal pattern length via suffix links, which splits into at most two contiguous parts: states without an outgoing transition labeled c, and states with such a transition. For states without transition c, we create a new state and connect all of them to it via transitions labeled c (lines 6-7). For states with transition c, let A be the first such state and let B be its c-successor (line 10). If the maximal pattern length of B equals that of A plus one, then B already represents the correct equivalence class after extension and no structural change is required (line 11). Otherwise, we create a copy B' of B so that B' represents position list extendable by the new symbol (i.e., $poslist(B') = poslist(B) \cup \{(i, j + 1)\}$), while B preserves the original position list (lines 13-14). We then update the remaining suffix states on the chain by redirecting their c-transitions from B to B'. Consequently, each extension step introduces at most two new states and updates transitions only along the suffix-link chain, yielding amortized constant processing time. The suffix links themselves can also be maintained in constant time per symbol. After processing a sequence, we need to reset suffix state to 0 before processing the next sequence. This guarantees that only suffixes of the current sequence are extended, hence no pattern spanning multiple sequences is ever constructed.

Vector ID propagation. Vector IDs can be propagated in an online manner during ESAM construction. Since each extension step processes all suffix states along the suffix-link chain, these states correspond exactly to patterns that end at the current position and should therefore include the current sequence ID. Thus, we propagate the vector ID by traversing the suffix-link chain starting from the newly created state and adding the ID until reaching a state that already contains it. This procedure is shown in line 16 of Algorithm 3.

State index construction. To address the significant overlap among ID sets and the presence of small ID sets, we propose two strategies: an **index reuse strategy** and a **skip-build strategy**. Specifically, we construct state indexes in reverse topological order of the automaton. In this order, when processing a state u, all its descendant states have already been processed. We identify the descendant of u with the largest indexed ID set by examining its outgoing neighbors. We then define the *base set* of u as the difference set between $ID(u)$ and the largest indexed set of its descendants. If $|base(u)| < T$, we directly store $base(u)$ as a raw ID set. Otherwise, we build an HNSW index over the vectors corresponding to $base(u)$. This process ensures effective index reuse while avoiding unnecessary graph construction for small sets. The detailed procedure is shown in lines 17–21 of Algorithm 3.

We remark that all vectors are stored in a global array. Each HNSW graph maintains only the IDs of vectors rather than storing local copies of the vectors themselves. Therefore, the total space usage of VectorMaton is dominated by the storage of ID sets and ESAM structures, and is linear in the total size of all ID sets.

Theorem 2. *Let m denote the total length of all sequences. With a constant size of symbol set, the overall space complexity of VectorMaton is bounded by $O(m^{1.5})$.*

Parallel construction. The primary bottleneck in index construction is building the HNSW graphs. Although the construction of a single HNSW graph is inherently sequential and difficult to parallelize efficiently, different state indexes are independent once their descendant states have been processed. To exploit this property, in parallel settings, we maintain a concurrent queue that stores states whose descendant states have all been processed (i.e., ready states in reverse topological order). Worker threads repeatedly dequeue ready states, construct their corresponding indexes (either raw sets or HNSW graphs), and update the readiness of their predecessor states. Newly ready states are then inserted into the queue. This design enables parallel construction of multiple HNSW graphs to improve construction efficiency while preserving correctness. Our experimental results show that with 16 threads, the index can be constructed within 2 hours on billion-scale datasets.

5 DISCUSSIONS

In this section, we discuss the maintenance of VectorMaton index.

Insertion. Since the automaton is constructed incrementally, a newly arriving vector–sequence pair can be processed by extending the automaton with the new sequence and updating the corresponding ID sets. To avoid rebuilding state indexes via a reverse topological traversal after each insertion, we update indexes online. During ID propagation along the suffix-link chain, we directly insert the new vector ID into each affected state's index. If the state maintains a raw ID set, we simply append the ID; if it maintains an HNSW index, we insert the vector into the corresponding graph. This incremental update strategy preserves correctness while avoiding expensive global index reconstruction.

Deletion. Deletion is more challenging, as both the automaton structure and the HNSW index do not naturally support efficient structural removal. Instead of modifying the structure, we adopt a lazy deletion strategy. Specifically, given a vector–sequence pair to be removed, we repeat the insertion-style traversal over the sequence to locate all affected suffix states. For each such state, we mark the corresponding vector ID as deleted in its index. During query processing, these marked IDs are filtered out from the returned results. This approach avoids expensive structural updates while preserving correctness, at the cost of a lightweight filtering step at query time. We remark that designing an efficient garbage collection mechanism to reclaim space from deleted entries remains challenging and is a direction for future work.

Datasets	No. vectors	Total seq. len.	Dim.
spam [42]	489	13,643	384
words [15]	8,000	56,209	3,072
mtg [44]	21,550	1,504,633	1,152
arxiv [35]	157,605	9,851,413	768
prot [28]	455,692	116,326,099	1,024
code [23]	1,838,414	40,940,167	768

Table 2: Datasets used in our experiments.

6 EXPERIMENTS

6.1 Setup

System environment. All experiments were conducted on a machine equipped with an AMD Ryzen Threadripper PRO 7975WX 32-core processor and 755 GB of RAM. The code was compiled using GCC 11.4.0 on Ubuntu Linux with the `-O3` optimization flag enabled.

Datasets. We evaluate our method on a diverse set of datasets collected from Hugging Face and the SpamAssassin corpus, as summarized in Table 2. The detailed descriptions are provided below.

- **Spam** [42]: a dataset of spam emails from the SpamAssassin corpus. Each record consists of an email title used as the sequence, and a pretrained embedding of the email content (generated by `all-MiniLM-L6-v2`) used as its vector.
- **Words** [15]: a dataset of word embeddings. Each record consists of a word (a sequence of letters) as the sequence and its pretrained word embedding as the vector.
- **MTG** [44]: a dataset of image embeddings. Each record consists of an image description as the sequence and a pretrained image embedding vector as its representation.
- **ArXiv** [35]: a dataset of paper titles and embeddings. Each record consists of a paper title as the sequence and a text embedding vector (generated by `all-mpnet-base-v2`) as its representation.
- **SwissProt** [28]: a manually curated subset of the UniProt protein sequence database. Each record consists of a protein sequence as the sequence and a structural embedding vector (generated by ProtBERT) as its representation.
- **CodeSearchNet** [23]: a dataset of code snippets. Each record consists of a function name as the sequence and a code embedding vector (generated by CodeBERT) as its vector representation.

Baselines. We compare VectorMaton against four baselines:

- **OptQuery**: an index-based optimal query baseline described in Algorithm 1.
- **PreFiltering**: a filter-then-search approach (Algorithm 2), where we first use the enhanced suffix automaton as the filtering index S to retrieve candidate vectors, and then perform vector search over the filtered subset.

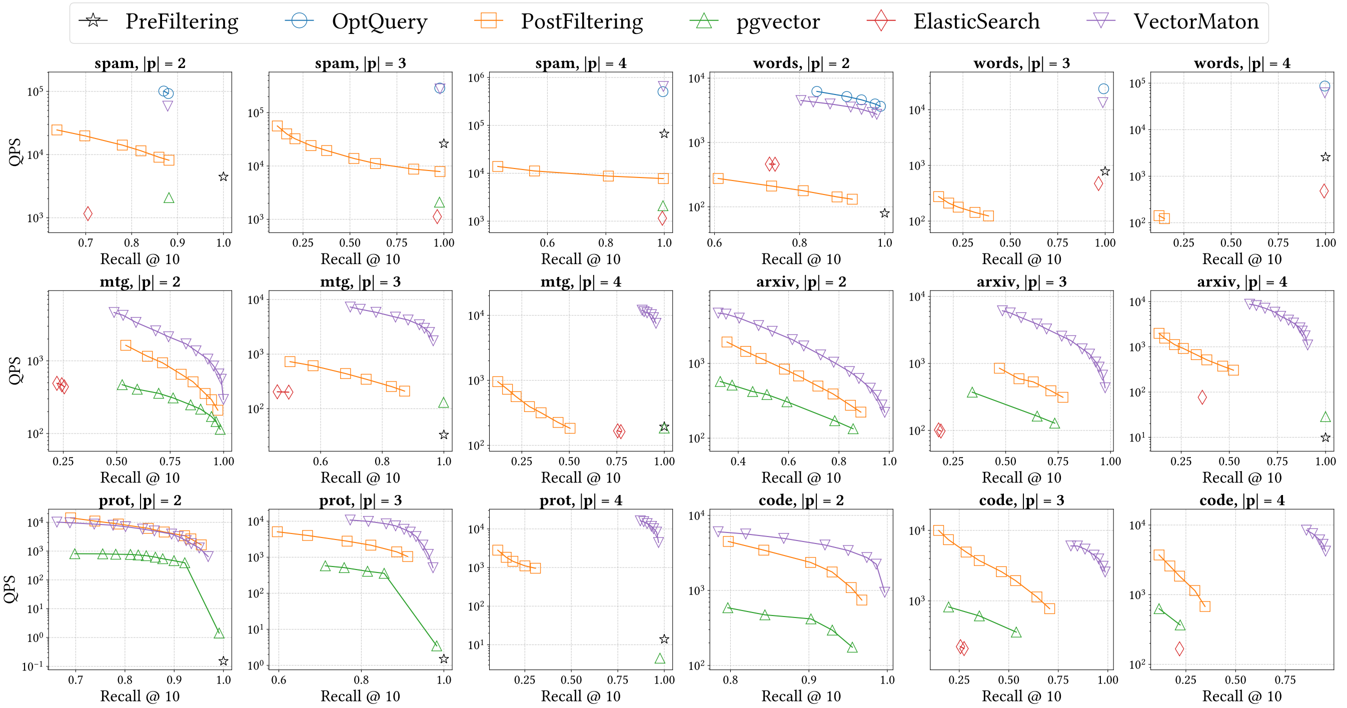
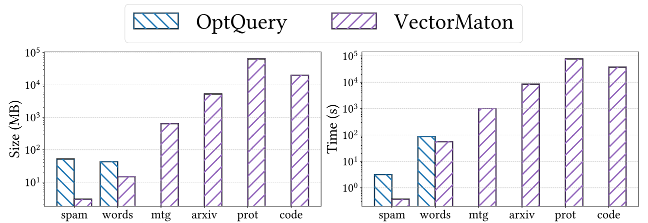


Figure 9: QPS vs. recall (PreFiltering is omitted if its QPS is significantly lower).



(a) Index size

(b) Index time

Figure 10: Index size and construction time.

- **PostFiltering**: a search-then-filter approach (Algorithm 2), which first performs ANN search over the full dataset and subsequently filters results based on the pattern constraint.
- **Pgvector** [1]: a vector extension of PostgreSQL, which supports hybrid SQL queries with vector similarity based on HNSW index and `LIKE` predicates. To minimize disk I/O overhead and ensure fair comparison, we store all data in temporary tables and configure `temp_buffers` to 128 GB so that query processing is performed entirely in memory.
- **ElasticSearch** [2]: an engine that supports hybrid queries by combining vector similarity based on HNSW index with keyword-based filtering (e.g., `match` or `wildcard` predicates). To minimize disk I/O overhead and ensure fair comparison, we allocate sufficient JVM heap space to keep all vectors and indices in memory.

As we are the first to formally study pattern-constrained ANNS, there are no directly comparable existing baselines beyond pgvector [1] and ElasticSearch [2]. We note that ElasticSearch is implemented in Java, whereas the other evaluated methods, including ours, are implemented in C or C++. Such implementation differences may lead to variations in runtime overhead and latency. In addition, pgvector operates within a SQL-based framework, which requires SQL parsing and query planning prior to execution. This architectural design may introduce additional overhead compared to our native implementation over the queries.

Parameter settings. PreFiltering does not involve any ANN-specific tuning parameters. For the remaining methods, including VectorMaton, we vary the HNSW search parameter ef_search in the range $[8, 1024]$ and sample representative points to plot the QPS–recall curves. ElasticSearch does not support adjusting ef_search, and we adjust its parameter "num_candidates" alternatively. Unless otherwise specified, the maximum degree of the HNSW graph is set to $M = 16$, and the construction parameter ef_con is set to 200. For VectorMaton, the threshold T of constructing a HNSW graph is set to 200.

Queries. We generate 1,000 queries for each pattern length $|p| \in 2, 3, 4$ and set $k = 10$ (i.e., querying 10-NNs). For each query, the vector is generated using a random number generator, while the pattern is randomly sampled from substrings of the specified length that appear in the sequence collection.

6.2 Query performance

Overall performance. Figure 9 reports the query performance of VectorMaton and all baselines. OptQuery encounters out-of-memory (OOM) errors on the MTG, ArXiv, SwissProt, and CodeSearchNet datasets, and is therefore omitted from the corresponding plots. Pgvector does not support constructing HNSW indexes for

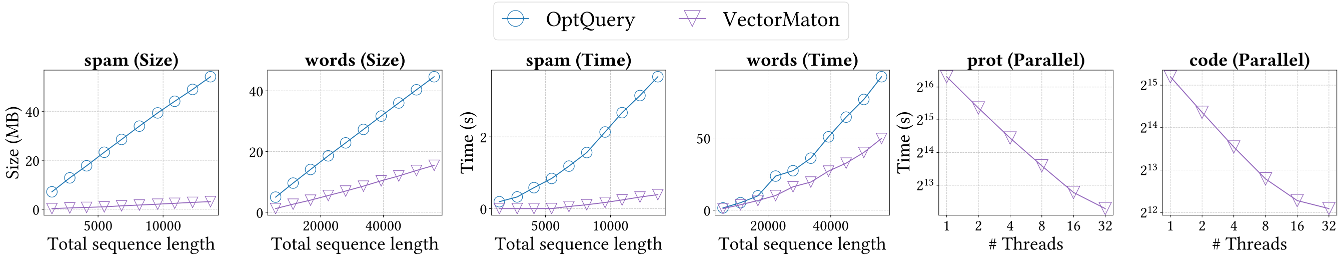


Figure 11: Index scalability test and parallelized index construction time.

Datasets	spam	words	mtg	arxiv	prot	code
Reduction (size)	71.3%	58.6%	66.3%	64.7%	/	57.9%
Datasets	spam	words	mtg	arxiv	prot	code
Reduction (time)	43.6%	42.1%	43.8%	59.7%	/	38%

Table 3: Reduction of index size and construction time using proposed index strategies.

vectors with dimensionality larger than 2000 and is thus excluded from the Words dataset. ElasticSearch exhibits a recall less than 0.1 on SwissProt dataset and CodeSearchNet dataset ($|p| = 2$ case), and is thus omitted. PreFiltering is omitted in cases where its QPS is less than 10% of the lowest QPS achieved by the other methods.

Overall, VectorMaton achieves a performance trade-off comparable to OptQuery (when OptQuery is feasible) and outperforms the remaining baselines across most datasets and parameter settings. For example, on the CodeSearchNet dataset with pattern length $|p| = 3$, the lowest recall achieved by VectorMaton exceeds the highest recall achieved by PostFiltering, while delivering approximately 3× higher QPS. On the SwissProt dataset with $|p| = 2$, PostFiltering performs comparably to VectorMaton, as short patterns filter out very few sequences and thus have limited impact on the recall of PostFiltering. Pgvector adopts a hybrid strategy combining pre-filtering and post-filtering, and may occasionally reach recall = 1 under certain parameter settings. While ElasticSearch provides a tunable "num_candidates" parameter, we observe that different parameter values do not affect its recall and QPS much, and thus it shows only few points on all datasets.

Effects of varying $|p|$. To evaluate the impact of pattern selectivity, we vary the pattern length $|p| \in \{2, 3, 4\}$ for each dataset. The results are shown in Figure 9. As $|p|$ increases, the number of vectors satisfying the pattern constraint generally decreases, leading to lower selectivity. Under this setting, both PostFiltering and pgvector experience performance degradation, as they must expand the search space (e.g., increasing ef_search or scanning more candidates) to compensate for the higher fraction of filtered-out results and maintain recall. In contrast, VectorMaton and PreFiltering benefit from larger $|p|$. A longer pattern reduces the size of the filtered vector set V_p, thereby shrinking the effective search space of VectorMaton and PreFiltering. As a result, VectorMaton achieves improved query efficiency while maintaining high recall, and the query efficiency of PreFiltering also increases as $|p|$ increases.

6.3 Index construction

Index size. Figure 10(a) compares the index size of OptQuery and VectorMaton on each dataset. Overall, VectorMaton consistently consumes significantly less space than OptQuery. Specifically, OptQuery requires approximately 18× and 3× more space than VectorMaton on the Spam and Words datasets, respectively. Moreover, OptQuery encounters OOM errors on the MTG, ArXiv, SwissProt, and CodeSearchNet datasets, while VectorMaton successfully constructs indexes on all of them. The reason is that OptQuery requires $O(m^2)$ space whereas VectorMaton requires $O(m^{1.5})$ space in the worst case, where m denotes the total sequence length. As m grows, the gap between the two approaches becomes increasingly significant, and this leads to the OOM of OptQuery. On the Words dataset, however, each sequence is relatively short, which prevents OptQuery from exhibiting substantial space blowup.

Index construction time. Figure 10(b) compares the index construction time of OptQuery and VectorMaton across all datasets. VectorMaton consistently outperforms OptQuery, achieving approximately 9× and 1.5× speedup on the Spam and Words datasets, respectively. For OptQuery and VectorMaton, the index construction time is in proportional to index size. Therefore, the substantial reduction in index size achieved by VectorMaton directly translates into faster construction time.

Index scalability test. We evaluate the scalability of index size and construction time on Spam and Words datasets. For each dataset, we construct the OptQuery and VectorMaton index over progressively larger subsets $\{10\%, 20\%, \cdots, 100\%\}$ of the data and plot the resulting size-length and time-length curves. Figure 11 (spam, words) illustrates the results. Although the theoretical worst-case space complexity is $O(m^2)$ for OptQuery and $O(m^{1.5})$ for VectorMaton, the empirical space costs exhibit near-linear growth in practice for both methods. This behavior arises because real sequences typically contain repeated substrings, resulting in fewer distinct patterns to index than the worst-case bound. In addition, we observe the growth rate of VectorMaton is consistently and significantly lower than that of OptQuery. As the dataset size increases, the gap between the two methods becomes larger. These results demonstrate that VectorMaton scales gracefully with increasing data size and can effectively handle large-scale sequence collections.

Parallelized index construction. We evaluate the scalability of parallel index construction on two large-scale datasets, SwissProt and CodeSearchNet. For each dataset, we construct the index using

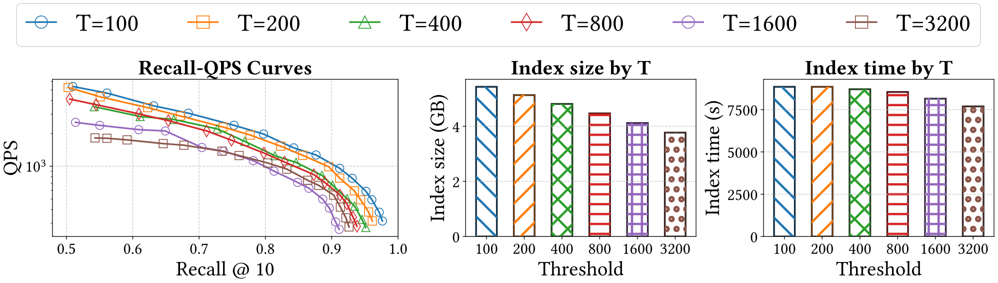


Figure 12: Parameter study of build threshold T.

1, 2, 4, 8, 16, and 32 threads, respectively. Figure 11 (prot, code) reports the results. The results show that VectorMaton scales effectively in a parallel setting, achieving near-linear speedup when the number of threads does not exceed 16. Beyond this point, the speedup decreases. This is because our parallelization strategy distributes the construction of different state indexes across threads, while the construction of each individual index (e.g., an HNSW graph) remains sequential. As the number of threads increases, the overall runtime becomes dominated by a few large state indexes whose construction cannot be further parallelized, leading to reduced scalability gains. Nevertheless, when the number of threads is not less than 16, the index construction time is reduced to less than two hours on both datasets, which is efficient for large-scale datasets.

Ablation study of index strategies. Recall that VectorMaton incorporates two design strategies: index reuse across states and selective index construction (skip-build). We conduct an ablation study to evaluate their contributions, and the results are summarized in Table 3. We do not report results on SwissProt dataset as the index construction process without these two strategies fail to finish within 48 hours. The results show that they effectively reduce index size and construction time, with an up to 71.3% reduction in index size and 59.7% reduction in index time. The index reuse strategy minimizes redundancy of overlapping vector subsets across related states, while the skip-build strategy reduces the number of HNSW graphs constructed by avoiding index construction for small ID sets. These optimizations improve construction efficiency without compromising query performance.

Effect of parameter T. In the skip-build strategy, we introduce a threshold parameter T to determine whether an HNSW index should be constructed for a state. If the size of the associated vector set of a state is smaller than T, we store the raw ID set instead of building an HNSW graph. Figure 12 illustrates the impact of varying T, where the queries are generated with mixed pattern lengths $|p| \in \{2, 3, 4\}$. As T increases, both index size and construction time decrease, since more states skip graph construction. Meanwhile, query performance remains stable within a moderate range of T, indicating that brute-force search over small vector sets is sufficiently efficient. However, when T becomes too large, more states rely on brute-force search, which may degrade query performance. This suggests a trade-off between index construction cost and query efficiency when selecting T, which can be further investigated in future work.

7 RELATED WORKS

ANN approaches. Approximate nearest neighbor search (ANNS) methods can be generally classified into four categories: (1) tree-based methods [9, 22, 39]; (2) hashing-based methods [3, 14, 45, 49]; (3) quantization-based methods [4, 6, 17, 18]; and (4) graph-based methods [16, 34, 43]. Tree-based methods partition the data space hierarchically and organize points into recursive structures, including KD-tree [9], ball tree [39], and ring-cover tree [22]. Hashing-based methods include locality-sensitive hashing (LSH) [3, 45, 49] and SimHash [14]. LSH ensures that nearby points collide in the same bucket with high probability. SimHash [14] implements random hyperplane hashing for angular similarity. Quantization-based methods [4, 6, 17, 18] encode high-dimensional vectors into compact representations using learned codebooks and approximate distances in the compressed space. Graph-based methods construct a proximity graph over the dataset and perform query processing via greedy or best-first graph traversal. Representative methods include HNSW [34], NSG [16], and DiskANN [43].

Filtered ANN approaches. Existing filtered ANN approaches mainly focus on attribute-based filtering [20, 31, 32, 36] and range-based filtering [25, 41, 46–48, 50]. Attribute-based filtering includes selecting vectors whose categorical attribute equals a queried value [20], or whose attribute set contains the queried attribute set [32]. Range-based filtering considers numerical constraints associated with each vector [25, 41, 46–48, 50]. The objective is to answer ANNS queries while restricting results to vectors whose associated numeric value falls within a specified query interval.

8 CONCLUSION

In this paper, for the first time, we introduce the problem of ANNS with pattern constraints. To address this problem, we propose VectorMaton, an index that jointly supports similarity search and pattern filter. Experimental results show that VectorMaton achieves a favorable trade-off among query latency, recall, and index space consumption compared with baseline methods. Moreover, the index size grows empirically linearly with respect to the dataset size. In future work, we plan to extend VectorMaton to support dynamic updates, enabling efficient insertions and deletions for real-time ANN workloads. Another potential future direction is to design a protocol to tune parameters like T, M and ef_con on different states and datasets, which may further improve the trade-off between query efficiency and index size.

REFERENCES

[1] 2025. pgvector. https://github.com/pgvector/pgvector.

[2] 2025. Vector search in Elasticsearch. https://www.elastic.co/docs/solutions/search/vector.

[3] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51, 1 (Jan. 2008), 117–122. https://doi.org/10.1145/1327452.1327494

[4] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: high-performance nearest neighbor search with product quantization fast scan. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 288–299. https://doi.org/10.14778/2856318.2856324

[5] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-Based Vector Search: An Experimental Evaluation of the State-of-the-Art. *Proc. ACM Manag. Data* 3, 1, Article 43 (Feb. 2025), 31 pages. https://doi.org/10.1145/3709693

[6] Artem Babenko and Victor Lempitsky. 2015. Tree quantization for large-scale similarity search and classification. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4240–4248. https://doi.org/10.1109/CVPR.2015.7299052

[7] Timothy L Bailey, Mikael Boden, Fabian A Buske, Martin Frith, Charles E Grant, Luca Clementi, Jingyuan Ren, Wilfred W Li, and William S Noble. 2009. MEME SUITE: tools for motif discovery and searching. *Nucleic Acids Res* 37, Web Server issue (May 2009), W202–8.

[8] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation Learning: A Review and New Perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* 35, 8 (Aug. 2013), 1798–1828. https://doi.org/10.1109/TPAMI.2013.50

[9] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517. https://doi.org/10.1145/361002.361007

[10] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnell. 1984. Building the minimal DFA for the set of all subwords of a word on-line in linear time. In *Automata, Languages and Programming*, Jan Paredaens (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 109–118.

[11] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. 1985. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science* 40 (1985), 31–55. https://doi.org/10.1016/0304-3975(85)90157-4 Eleventh International Colloquium on Automata, Languages and Programming.

[12] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. 1987. Complete inverted files for efficient text retrieval and analysis. *J. ACM* 34, 3 (July 1987), 578–595. https://doi.org/10.1145/28869.28873

[13] Nadav Brandes, Dan Ofer, Yam Peleg, Nadav Rappoport, and Michal Linial. 2022. ProteinBERT: a universal deep-learning model of protein sequence and function. *Bioinformatics* 38, 8 (02 2022), 2102–2110. https://doi.org/10.1093/bioinformatics/btac020 arXiv:https://academic.oup.com/bioinformatics/article-pdf/38/8/2102/49009610/btac020.pdf

[14] Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) *(STOC '02)*. Association for Computing Machinery, New York, NY, USA, 380–388. https://doi.org/10.1145/509907.509965

[15] Efarrall. 2026. Word Embeddings. https://huggingface.co/datasets/efarrall/word_embeddings. Hugging Face dataset.

[16] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 461–474. https://doi.org/10.14778/3303753.3303754

[17] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2025. Practical and Asymptotically Optimal Quantization of High-Dimensional Vectors in Euclidean Space for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 3, Article 202 (June 2025), 26 pages. https://doi.org/10.1145/3725413

[18] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3, Article 167 (May 2024), 27 pages. https://doi.org/10.1145/3654970

[19] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 518–529.

[20] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 3406–3416. https://doi.org/10.1145/3543507.3583552

[21] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 855–864. https://doi.org/10.1145/2939672.2939754

[22] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (Dallas, Texas, USA) *(STOC '98)*. Association for Computing Machinery, New York, NY, USA, 604–613. https://doi.org/10.1145/276698.276876

[23] Irds. 2026. Code search net. https://huggingface.co/datasets/irds/codesearchnet. Hugging Face dataset.

[24] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (Jan. 2011), 117–128. https://doi.org/10.1109/TPAMI.2010.57

[25] Mengxu Jiang, Zhi Yang, Fangyuan Zhang, Guanhao Hou, Jieming Shi, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. *Proc. ACM Manag. Data* 3, 3, Article 148 (June 2025), 26 pages. https://doi.org/10.1145/3725399

[26] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

[27] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (01 Aug 2021), 583–589. https://doi.org/10.1038/s41586-021-03819-2

[28] Khairi. 2026. Uniprot swissprot. https://huggingface.co/datasets/khairi/uniprot-swissprot. Hugging Face dataset.

[29] Donald E. Knuth. 1998. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., USA.

[30] Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Eric P. Xing and Tony Jebara (Eds.), Vol. 32. PMLR, Bejing, China, 1188–1196. https://proceedings.mlr.press/v32/le14.html

[31] Mocheng Li, Xiao Yan, Baotong Lu, Yue Zhang, James Cheng, and Chenhao Ma. 2025. Attribute Filtering in Approximate Nearest Neighbor Search: An In-depth Experimental Study. *Proc. ACM Manag. Data* 3, 6, Article 298 (Dec. 2025), 26 pages. https://doi.org/10.1145/3769763

[32] Anqi Liang, Pengcheng Zhang, Bin Yao, Zhongpu Chen, Yitong Song, and Guangxu Cheng. 2024. UNIFY: Unified Index for Range Filtered Approximate Nearest Neighbors Search. *Proc. VLDB Endow.* 18, 4 (Dec. 2024), 1118–1130. https://doi.org/10.14778/3717755.3717770

[33] Shangqi Lu and Yufei Tao. 2025. Proximity Graphs for Similarity Search: Fast Construction, Lower Bounds, and Euclidean Separation. *Proc. ACM Manag. Data* 3, 5, Article 280 (Nov. 2025), 25 pages. https://doi.org/10.1145/3767716

[34] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (April 2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[35] Malteos. 2026. Aspect Paper Embeddings. https://huggingface.co/datasets/Qdrant/arxiv-titles-instructorxl-embeddings. Hugging Face dataset.

[36] Yusuke Matsui, Ryota Hinami, and Shin'ichi Satoh. 2018. Reconfigurable Inverted Index. In *Proceedings of the 26th ACM International Conference on Multimedia* (Seoul, Republic of Korea) *(MM '18)*. Association for Computing Machinery, New York, NY, USA, 1715–1723. https://doi.org/10.1145/3240508.3240630

[37] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL] https://arxiv.org/abs/1301.3781

[38] Mehryar Mohri, Pedro Moreno, and Eugene Weinstein. 2009. General suffix automaton construction algorithm and space bounds. *Theoretical Computer Science* 410, 37 (2009), 3553–3562. https://doi.org/10.1016/j.tcs.2009.03.034 Implementation and Application of Automata (CIAA 2007).

[39] Stephen M. Omohundro. 2009. Five Balltree Construction Algorithms. https://api.semanticscholar.org/CorpusID:61067117

[40] Malte Ostendorff, Till Blume, Terry Ruas, Bela Gipp, and Georg Rehm. 2022. Specialized document embeddings for aspect-based similarity of research papers. In *Proceedings of the 22nd ACM/IEEE Joint Conference on Digital Libraries* (Cologne, Germany) *(JCDL '22)*. Association for Computing Machinery, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/3529372.3530912

[41] Zhencan Peng, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2025. Dynamic Range-Filtering Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 18, 10 (June 2025), 3256–3268. https://doi.org/10.14778/3748191.3748193

[42] SpamAssassin. 2026. Spam emails. https://spamassassin.apache.org/old/publiccorpus/. Hugging Face dataset.

[43] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. *DiskANN: fast accurate billion-point nearest neighbor search on a single node.* Curran Associates Inc., Red Hook, NY, USA.

[44] TrevorJS. 2026. Mtg Scryfall Cropped Art Embeddings. https://huggingface.co/datasets/TrevorJS/mtg-scryfall-cropped-art-embeddings-siglip-so400m-patch14-384. Hugging Face dataset.

[45] Jiuqi Wei, Botao Peng, Xiaodong Lee, and Themis Palpanas. 2024. DET-LSH: A Locality-Sensitive Hashing Scheme with Dynamic Encoding Tree for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 17, 9 (May 2024), 2241–2254. https://doi.org/10.14778/3665844.3665854

[46] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6, Article 239 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698814

[47] Fangyuan Zhang, Mengxu Jiang, Guanhao Hou, Jieming Shi, Hua Fan, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 3, Article 152 (June 2025), 26 pages. https://doi.org/10.1145/3725401

[48] Fangyuan Zhang, Mengxu Jiang, Guanhao Hou, Jieming Shi, Hua Fan, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 3, Article 152 (June 2025), 26 pages. https://doi.org/10.1145/3725401

[49] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 643–655. https://doi.org/10.14778/3377369.3377374

[50] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 1, Article 69 (March 2024), 26 pages. https://doi.org/10.1145/3639324

A PROOF OF THEOREM 1

For each sequence s_i, the number of distinct substrings is at most $\frac{|s_i|(|s_i|+1)}{2}$ by lines 4-5 in Algorithm 1. Therefore, across all n sequences, the total number of vector insertions is bounded by:

$$\sum_{i=1}^{n} \frac{|s_i|(|s_i|+1)}{2} \le \sum_{i=1}^{n} |s_i|^2 + \sum_{i=1}^{n} |s_i| \le m^2 + m = O(m^2)$$

Each insertion corresponds to a vector stored in a pattern-specific HNSW graph. Since the HNSW construction bounds the number of neighbors per node by a fixed constant, the total space required for all graphs is $O(m^2)$.

B PROOF OF LEMMA 1

Consider a maximal pattern whose position list contains a pair (id, pos). Such a pattern must be a substring of the form $s_{id}[i : pos]$, representing the substring from position i to pos in sequence s_{id}. For $0 \le i < pos$, suppose that $s_{id}[i : pos]$ and $s_{id}[i+1 : pos]$ are not equivalent. Then, by the definition of equivalence classes, $s_{id}[i+1 : pos]$ must occur in some other pair (id', pos'), while $s_{id}[i : pos]$ does not. In other words, for x maximal patterns whose position lists contain (id, pos), there must exist at least $x - 1$ occurrences of these patterns associated with other (id', pos') pairs.

Now, to count the total number of distinct maximal patterns, we sum over all (id, pos) pairs. For each pair, the contribution to the total number of distinct maximal patterns is at most $x_{id,pos} - (x_{id,pos} - 1) = 1$. Therefore, summing over all such pairs, the total number of distinct maximal patterns is at most $\sum_{(id,pos)} 1 = m$, where m is the total length of all sequences.

C PROOF OF LEMMA 2

Consider a sequence s of length $|s|$. It has at most $\frac{|s|(|s|+1)}{2}$ distinct substrings, and thus can contribute its sequence ID to at most that many maximal patterns. By Lemma 1, the total number of states in VectorMaton is bounded by $O(m)$; denote this bound by cm for some constant c. Therefore, the ID i of sequence s_i can appear in at most: $\min\left\{\frac{|s_i|(|s_i|-1)}{2}, cm\right\}$ states. Let S denote the sequence collection. The total size of all associated ID sets is thus bounded by:

$$\sum_{i=1}^{n} \min\left\{\frac{|s_i|(|s_i|-1)}{2}, cm\right\}$$

Denote $t_i = \min\left\{\frac{|s_i|^2}{2}, cm\right\} \ge \min\left\{\frac{|s_i|(|s_i|-1)}{2}, cm\right\}$. We consider two cases.

- If $\frac{|s_i|^2}{2} \le cm$, then $|s_i| \le \sqrt{2cm}$ and $t_i = \frac{|s_i|^2}{2} \le |s_i| \cdot \frac{\sqrt{2cm}}{2}$.
- If $\frac{|s_i|^2}{2} > cm$, then $\sqrt{m} < \frac{|s_i|\sqrt{2c}}{2c}$ and $t_i = cm < |s_i| \cdot \frac{\sqrt{2cm}}{2}$.

Therefore, the total size of all associated ID sets is bounded by:

$$\sum_{i=1}^{n} t_i \le \sum_{i=1}^{n} |s_i| \frac{\sqrt{2cm}}{2} = \frac{\sqrt{2c}}{2} m^{1.5} = O(m^{1.5})$$

D PROOF OF LEMMA 3

Let $poslist(p_A) = \left\{(id_j, pos_j)\right\}_{j=1}^{k}$. Since $p_B = p_A \cdot c$, an occurrence of p_A at position (id_j, pos_j) can be extended by c if and only if $pos_j < |s_{id_j}| - 1$ and the next character is c. Therefore, $poslist(p_B) = \left\{(id_j, pos_j + 1) \mid 1 \le j \le k, s_{id_j}[pos_j + 1] = c\right\}$. For any other pattern $p'_A \in A$, by definition of equivalence classes we have $poslist(p'_A) = poslist(p_A)$. Hence, extending p'_A by the same character c yields the same set of occurrences and thus the same position list as p_B. Consequently, $p'_A \cdot c$ belongs to the same equivalence class B, which establishes a well-defined transition from state A to state B labeled by c.

E PROOF OF LEMMA 4

Let A be a suffix state with position list $poslist(A) = (x_k, y_k)$. If there is no outgoing transition from A labeled c, then for every $(x_k, y_k) \in poslist(A)$, either $y_k + 1$ exceeds the length of s_{x_k} or $s_{x_k}[y_k + 1] \ne c$. Hence, none of the existing occurrences of patterns in A can be extended by c. Since we are currently processing position (i, j) and the next symbol is $c = s_i[j + 1]$, appending c introduces a new occurrence ending at $(i, j + 1)$. Therefore, extending A by c yields a new equivalence class with position list $\{(i, j + 1)\}$.

F SUFFIX LINK AND ANALYSIS

F.1 Correctness

In this section, we will prove the correctness of our automaton construction algorithm in an on-line faction (i.e., the correctness of the incremental process). We first present the definition of the *suffix link* [10]. While it is originally introduced for the single-sequence setting, the concept naturally extends to our multi-sequence scenario.

Definition 6 (Suffix link). *Given a state A whose maximal pattern is s, the suffix link* link(*A*) *points to the state whose maximal pattern is the longest proper suffix of s that is not equivalent to s.*

We next show that all suffix states can be organized into a chain of strictly decreasing maximal pattern lengths via suffix links.

Lemma 5. *Consider sequence s_i currently being processed, and suppose the algorithm has processed position j. Let A be the state corresponding to pattern $s_i[0..j]$. Then the set of all suffix states is precisely $\{A, \text{link}(A), \text{link}(\text{link}(A)), \cdots, 0\}$, where 0 denotes the initial state.*

Proof. By definition of suffix states, a suffix state is a state whose position list is empty or contains (i, j). Hence, every suffix state represents at least a pattern that is a suffix of $s_i[0..j]$. Moreover, the maximal pattern in a suffix state must be also a suffix of $s_i[0..j]$, which is not hard to see.

Let A be the state corresponding to $s_i[0..j]$. Then A is a suffix state with maximal pattern $s_i[0..j]$. By construction of suffix links, $\text{link}(A)$ points to the state representing the longest proper suffix of the maximal pattern of A that belongs to a different equivalence class. Repeatedly following suffix links therefore enumerates states whose maximal patterns are exactly the suffixes of $s_i[0..j]$ in strictly decreasing order of length, until reaching the initial state 0, which represents the empty pattern. Each of these states contains (i, j) in its position list (or is the initial state), and is thus a suffix state.

Conversely, let C be any suffix state. Then C represents a pattern that is a suffix of $s_i[0..j]$. By the defining property of suffix links in the (generalized) suffix automaton, every suffix of $s_i[0..j]$ is represented by a state on the suffix-link chain starting from A. Therefore, C must belong to the set $\{A, \text{link}(A), \text{link}(\text{link}(A)), \cdots, 0\}$. □

Next, we show that the suffix-link chain of suffix states can be partitioned into at most two contiguous segments: (1) states without an outgoing transition labeled $c = s_i[j+1]$, and (2) states with such a transition.

Lemma 6. *Let A be the state corresponding to pattern $s_i[0..j]$, and let $c = s_i[j+1]$ be the next symbol to be processed. Then the suffix-link chain $\{A, \text{link}(A), \text{link}^2(A), \cdots, 0\}$ can be divided into at most two contiguous segments such that:*

- *In the first segment (if it exists), no state has an outgoing transition labeled c.*
- *In the second segment (if it exists), every state has an outgoing transition labeled c.*

Proof. Let A be the state corresponding to the pattern $s_i[0..j]$, and consider the suffix-link chain $\{A, \text{link}(A), \text{link}^2(A), \dots, 0\}$.

If A has an outgoing transition labeled c, then the extended pattern $s_i[0..j+1]$ is already represented in the successor state. By the construction of suffix links, all states reachable via link from A correspond to shorter suffixes of $s_i[0..j]$. Since $s_i[0..j+1]$ contains these suffixes, each of these states must also have an outgoing transition labeled c.

Otherwise, if A has no transition labeled c, we follow the suffix links until we reach the first state B that does have a c-transition (if any). All states before B lack such a transition, while all states from B onward have it. If no state along the chain has a c-transition, then the entire chain forms a single segment without a c-transition. □

For the first segment, by Lemma 4, we create a new state and connect all states in that segment to it via transitions labeled c. For the second segment, we only need to examine the first state in the segment.

Lemma 7. *Let B be the first state in the second segment of the suffix-link chain, and let C be the successor of B via the transition labeled c. Let $s(B)$ and $s(C)$ denote the maximal patterns of states B and C, respectively. If $|s(C)| = |s(B)| + 1$, then extending by c does not introduce any structural change in the second segment.*

Proof. We first establish the following claim.

Claim. Let X be a state with maximal pattern $s(X)$. Then every pattern y in the equivalence class of X is a suffix of $s(X)$.

Proof of claim. Let (i, j) be any occurrence in the position list of $s(X)$. Since all patterns in state X share the same position list, the pattern $y \in X$ must also end at position j in sequence s_i. Hence, $s(X) = s_i[j - |s(X)| + 1..j]$ and $y = s_i[j - |y| + 1..j]$, which shows that y is a suffix of $s(X)$.

Now consider the main statement. Since $|s(C)| = |s(B)| + 1$, the maximal pattern of C is exactly the extension of $s(B)$ by c, i.e., $s(C)$ is a suffix of $s_i[0..j+1]$. By the claim, all patterns represented in state C are suffixes of $s(C)$, and they are thereby suffixes of $s_i[0..j+1]$. Thus, when the currently processed sequence is extended by c at position $(i, j+1)$, all patterns in C remain consistent with this extension and belong to the same equivalence class. Consequently, no refinement of state C is required, and no structural change occurs in the second segment. □

We now consider the remaining case where $|s(C)| > |s(B)| + 1$. This implies that state C represents patterns strictly longer than the direct extension of $s(B)$ by c. In this situation, the extension by c introduces a refinement of the equivalence relation represented by C, and the state must be split.

Lemma 8. *Let B be the first state in the second segment of the suffix-link chain, and let C be the successor of B via the transition labeled c. Let $s(B)$ and $s(C)$ denote the maximal patterns of states B and C, respectively. If $|s(C)| > |s(B)| + 1$, then for every state B' in the second segment whose c-transition points to C, the equivalence class obtained by extending B' with c is no longer correctly represented by C.*

Proof. By our previous claim, $s(B) + c$ belongs to C and should be a suffix of $s(C)$. Since $s(B)$ is a suffix of $s_i[0..j]$, either $s(C)$ is a suffix of $s_i[0..j+1]$ or $s_i[0..j+1]$ is a substring of $s(C)$. For the latter case, the conclusion is trivial as $s(C)$ does not occur at $(i, j+1)$, violating the equivalence class of $s(B) + c$. For the former case, this indicates there exists some other state D which extends a sequence $x \in D$ to $s(C)$ via transition c, and x is thereby a suffix of $s_i[0..j]$. Therefore, D should be within the suffix-link chain and x is longer than $s(B)$. However, since B is the first state in the second segment, all previous states with longer suffixes of $s_i[0..j]$ should have no outgoing transition labeled c, which leads to a contradiction. □

After splitting state C into C and its clone C', we maintain the invariant that $\text{poslist}(C') = \text{poslist}(C) \cup \{(i, j+1)\}$, while C retains its original position list. We now determine precisely which transitions must be redirected to C'.

Let X be a suffix state in the second segment whose c-transition previously pointed to C. Then its maximal pattern occurs at position (i, j). Extending this pattern by $c = s_i[j+1]$ therefore produces an occurrence ending at $(i, j+1)$. Consequently, the extended pattern has position list equal to $\text{poslist}(C')$, and must be represented by state C'. Hence the outgoing c-transition of X must be redirected to C'.

Conversely, let Y be any state that is not a suffix state whose c-transition points to C. Extending its maximal pattern by c does not yield an occurrence ending at $(i, j+1)$, and therefore its extended pattern does not include $(i, j+1)$ in its position list. The corresponding equivalence class thus remains $\text{poslist}(C)$, and the transition of Y continues to point to C. No redirection is required.

Therefore, exactly the suffix states in the second segment require their c-transitions to be redirected from C to C'. Combining all above analysis, we conclude that after each extension step, the automaton maintains correct to represent all equivalence classes.

F.2 Maintenance of suffix links

As discussed above, processing each symbol creates at most two new states. We now describe how suffix links are maintained. For the state created by cloning, its suffix link is set as the same suffix link value of original state being cloned. For the newly created state Q that is introduced for the first segment (i.e., states without an outgoing transition labeled c), its suffix link depends on whether the second segment exists:

- If the second segment does not exist (i.e., no suffix state has an outgoing transition labeled c), then $\text{link}(Q)$ is set to the initial state 0.

- Otherwise, let B be the first state in the second segment, and let C be its successor via the transition labeled c after any necessary cloning has been performed. Then we set $\text{link}(Q) = C$.

This maintains the invariant that the suffix link of a state points to the state representing the longest proper suffix of its maximal pattern.

Lemma 9. *The suffix link* $\text{link}(Q)$ *constructed as above is correct.*

Proof. Let $s(Q) = s_i[0..j + 1]$ denote the maximal pattern of the newly created state Q.

Case 1: The second segment does not exist. This means that no suffix state of $s_i[0..j]$ has an outgoing transition labeled c. Hence, for every proper suffix x of $s_i[0..j]$, the extension $x + c$ does not occur in the previously processed prefix. Consequently, among all proper suffixes of $s(Q)$, none except the empty pattern corresponds to an existing state in a different equivalence class. Therefore, the longest proper suffix of $s(Q)$ that belongs to a different equivalence class is the empty pattern, represented by the initial state 0. Thus $\text{link}(Q) = 0$ is correct.

Case 2: The second segment exists. Let B be the first state in the second segment, and let C be the successor of B via transition labeled c after any necessary structural updates. Since B lies on the suffix-link chain of $s_i[0..j]$, $s(B)$ is a suffix of $s_i[0..j]$, and hence $s(B) + c$ is a proper suffix of $s(Q)$. Moreover, all suffix states preceding B (which correspond to strictly longer suffixes of $s_i[0..j]$) do not have a transition labeled c. Therefore, no longer proper suffix of $s(Q)$ corresponds to an existing state in a different equivalence class. Thus $s(B) + c$ is the longest proper suffix of $s(Q)$ that belongs to a distinct equivalence class, and it is represented by state C. Hence, setting $\text{link}(Q) = C$ satisfies the suffix-link invariant. □

G PROOF OF THEOREM 2

By Lemma 1, the number of states in VectorMaton is bounded by $O(m)$, and by Lemma 2, the total size of all ID sets is bounded by $O(m^{1.5})$. If the symbol set size is constant, each state has at most a constant number of outgoing transitions. Therefore, the total number of transitions is also bounded by $O(m)$, and the overall space consumption is dominated by the storage of ID sets and associated index structures, which is bounded by $O(m^{1.5})$.