Vector Databases

MehrCodeLand
Linkedin

Theoretical Foundations

Vector databases represent a convergence of multiple fields in computer science, primarily information retrieval, database systems, and machine learning. They implement specialized data structures and algorithms optimized for high-dimensional vector spaces, addressing the foundational challenge known as the "curse of dimensionality."

The mathematical foundation of vector databases lies in metric spaces and proximity searches. Formally, a vector database operates over a metric space (X, d) where X is the set of all vectors and $d: X \times X \to \mathbb{R}^+$ is a distance function that satisfies the triangle inequality. Common distance metrics include:

- Euclidean distance: $d(x, y) = \sqrt{(\Sigma_i(x_i - y_i)^2)}$
- Cosine distance: $d(x, y) = 1 - (x \cdot y)/(\|x\| \cdot \|y\|)$
- Manhattan distance: $d(x, y) = \Sigma_i|x_i - y_i|$
- Inner product: $d(x, y) = -x \cdot y$ (for maximum inner product search)

Indexing Structures

The core innovation in vector databases is their specialized indexing structures for approximate nearest neighbor (ANN) search, which sacrifice perfect accuracy for dramatic performance improvements:

Graph-Based Approaches

Hierarchical Navigable Small World (HNSW) constructs a multi-layered navigable small world graph where each node connects to both close neighbors and some distant nodes:

- Implements a skip-list-like structure with logarithmic search complexity O(log n)
- Construction time is O(n log n) where n is the number of vectors
- Query complexity is O(log n) with constant approximation factor
- Parameters include M (max connections per node) and efConstruction (search width during construction)

Navigable Small World (NSW) is the conceptual predecessor to HNSW, implementing a single-layer navigable small world graph.

Space-Partitioning Methods

Inverted File Index (IVF) partitions the vector space into clusters:

- Training stage identifies k centroids using k-means clustering
- Each vector is assigned to its nearest centroid
- Search examines only vectors in the closest nprobe clusters
- Complexity is O(k + n/k × nprobe) where n is the total number of vectors

Product Quantization (PQ) compresses vectors by splitting them into subvectors and quantizing each independently:

- Decomposition: Split d-dimensional vectors into m subvectors of dimension d/m
- Subquantization: Apply k-means to each subspace (typically k=256 for 8-bit codes)
- Asymmetric Distance Computation (ADC): Compute distances using lookup tables
- Memory reduction is approximately factor m with 8-bit codes

Tree-Based Methods

Ball Tree recursively partitions the space into nested hyperspheres:

- Each non-leaf node represents a hypersphere containing its descendants
- Query traverses the tree, pruning branches that cannot contain nearest neighbors

- Theoretically efficient for low-dimensional spaces but degrades in high dimensions

KD-Tree partitions the space using hyperplanes aligned with coordinate axes:

- Construction recursively splits on the dimension with highest variance
- Query complexity is O(log n) in low dimensions but approaches O(n) in high dimensions
- Rarely used in modern vector databases due to poor performance in high-dimensional spaces

Compression Techniques

Vector databases employ sophisticated compression techniques to minimize memory footprint and maximize throughput:

Scalar Quantization maps each vector component to a discrete value:

- Reduces precision from 32-bit float to 8-bit integer (4x compression)
- Typically applies a linear transformation: q(x) = round((x - min) / (max - min) × 255)

Product Quantization as described above achieves higher compression ratios:

- Typically 8-16x compression with minimal accuracy loss
- Enables distance calculations directly in compressed space

Binary Compression represents vectors as binary codes:

- Locality-Sensitive Hashing (LSH) produces binary codes where Hamming distance approximates original distance
- Learning-based approaches like ITQ (Iterative Quantization) optimize binary codes to preserve distances

Query Processing Architecture

Modern vector databases implement complex query processing pipelines:

1. **Query Preprocessing**:
    - Vector normalization (for cosine similarity)
    - Query expansion (enhancing recall)
    - Projection to compressed space if applicable
2. **Coarse Quantization**:
    - Identifying candidate clusters (IVF)
    - Pruning search space
3. **Fine-Grained Search**:
    - HNSW traversal within candidate clusters
    - Re-ranking candidates
4. **Post-Processing**:
    - Exact distance computation for top candidates (optional)
    - Filtering by metadata constraints
    - Score normalization and fusion with other signals

CRUD Operations

Vector databases extend traditional CRUD operations to high-dimensional spaces:

Create/Insert operations typically involve:

- Vector normalization
- Indexing structure updates (adding nodes to HNSW graph)
- Metadata indexing for filtering
- Performance is O(log n) for graph-based indexes

Read/Search operations support various similarity search types:

- k-nearest neighbors (k-NN): Finding k most similar vectors
- Range search: Finding all vectors within a distance threshold
- Hybrid search: Combining vector similarity with metadata filters

- Performance is O(log n) for approximate search

Update operations typically require:

- Removal of old vector representations
- Insertion of new vectors
- Re-optimization of index structures
- Most implementations treat updates as delete + insert

Delete operations involve:

- Removing nodes from index structures
- Updating connecting edges in graph-based indexes
- Handling "tombstone" markers to maintain consistency

Distributed Architecture

Enterprise-grade vector databases implement sophisticated distributed architectures:

Sharding Strategies:

- Random sharding: Vectors distributed randomly across nodes
- Cluster-based sharding: Similar vectors assigned to same shard
- Hybrid approaches: Two-level sharding with coarse clustering

Replication Models:

- Leader-follower replication for fault tolerance
- Multi-leader replication for write scalability
- Read replicas for query scalability

Distributed Query Execution:

- Scatter-gather pattern: Query sent to all shards, results merged
- Routing-based: Query directed to relevant shards based on cluster membership
- Hierarchical: Two-level search with global coarse index and local fine-grained indexes

Consistency Models:

- Strong consistency: All replicas in sync before acknowledging writes
- Eventual consistency: Replicas converge over time
- Read-after-write consistency: Client sees own writes immediately

Performance Optimizations

Advanced vector databases implement numerous performance optimizations:

Data Locality:

- Cache-conscious data layouts for dense vectors
- SIMD instructions for distance calculations
- Memory-mapped files for persistence with direct access

Algorithmic Optimizations:

- Early termination in search traversal
- Beam search with dynamic width
- Incremental distance calculations

Hardware Acceleration:

- GPU-accelerated indexing and search
- FPGA implementations for specialized workloads
- Tensor processing units (TPUs) for neural network operations

Query Planning:

- Cost-based optimization for hybrid queries
- Dynamic parameter selection based on workload
- Adaptive execution based on intermediate results

Transaction Processing

Enterprise vector databases implement ACID transaction semantics:

Atomicity: Ensuring multi-vector operations succeed or fail as a unit
Consistency: Maintaining index validity during concurrent operations

Isolation: Preventing interference between concurrent operations (using MVCC) **Durability**: Persisting vectors and index structures to non-volatile storage

Implementation techniques include:

- Write-ahead logging for recovery
- Multi-version concurrency control for isolation
- Lock-free data structures for concurrency
- Snapshot isolation for consistent reads

Practical Implementation Challenges

Several challenging problems arise in practical vector database implementations:

Cold-Start Problem:

- Initial index construction requires O(n log n) time
- Incremental construction techniques with quality guarantees

Index Maintenance:

- Degradation of index quality over time with insertions/deletions
- Periodic reindexing vs. incremental optimization

Curse of Dimensionality:

- Performance degradation in extremely high dimensions (>1000)
- Dimensionality reduction techniques as preprocessing

Data Skew:

- Handling non-uniform vector distributions
- Adaptive indexing based on data characteristics

Quality Drift:

- Semantic shift in embedding spaces over time
- Version control for embedding models and vectors

Research Frontiers

Current research in vector databases focuses on several frontiers:

Learned Index Structures:

- Using machine learning to optimize index structures
- Neural network-based ANN search algorithms

Multi-Modal Embeddings:

- Unified storage for text, image, audio embeddings
- Cross-modal similarity search

Streaming Vector Databases:

- Real-time indexing of continuous vector streams
- Time-aware similarity search with concept drift

Explainable Vector Search:

- Attribute-based explanations for similarity
- Counterfactual explanations for search results

Privacy-Preserving Vector Search:

- Homomorphic encryption for searching encrypted vectors
- Differential privacy guarantees for query results

Hybrid Search and BM25 in vector databases

MehrCodeLand

1. History of the Subjects

Hybrid search emerged as an evolution in information retrieval, combining traditional keyword-based search methods with modern vector embeddings. The concept developed as a response to the limitations of both approaches when used individually.

BM25 (Best Matching 25) has deeper historical roots, being developed in the 1970s and 1980s by Stephen Robertson and Karen Spärck Jones as part of the Probabilistic Retrieval Model framework. It was initially introduced as part of the OKAPI system at City University London and has since become one of the most successful and widely implemented ranking functions in information retrieval.

Vector databases, meanwhile, gained prominence in the late 2010s as machine learning models became more sophisticated at generating semantic embeddings - numerical representations of text that capture meaning beyond just keywords. The integration of these technologies to form hybrid search approaches began to take shape around 2020, as organizations recognized the need to combine the precision of keyword matching with the semantic understanding of vector embeddings.

2. Easy Explanation

Hybrid search is like having two detectives work on your case at the same time:

Detective 1 (Keyword Search): Looks for exact word matches in documents. For example, if you search for "apple pie recipe," it finds documents containing those exact words.

Detective 2 (Vector Search): Understands the meaning behind your search. So even if a document says "how to bake a delicious apple tart" without using the word "recipe," it still recognizes this as relevant.

BM25 is a special technique that Detective 1 uses. Instead of just counting how many times your search words appear in a document, BM25 is smarter:

- It gives higher scores to rare words (finding "pomegranate" is more meaningful than finding "the")
- It considers document length (so short documents with your keywords aren't unfairly advantaged)
- It accounts for word saturation (a document that mentions "apple" 50 times isn't necessarily 50 times more relevant than one that mentions it once)

Easy example: When you search "best running shoes for beginners" in a hybrid system:

- Keyword search finds articles with those exact terms
- Vector search understands you want entry-level athletic footwear
- The system combines these results to show you the most relevant information

Another example: A library using hybrid search could help you find books about "overcoming challenges" even if some relevant books use phrases like "conquering obstacles" or "facing adversity" instead.

3. Advanced Explanation

From a technical perspective, hybrid search combines lexical matching (like BM25) with semantic matching using dense vector representations. The BM25 ranking function is expressed as:

For a query Q containing terms t, the BM25 score for document D is:

```
score(D,Q) = Σ(t in Q) IDF(t) · (f(t,D) · (k1 + 1)) / (f(t,D) + k1 · (1 - b + b · |D|/avgdl))
```

Where:

- f(t,D) is the term frequency of t in document D
- |D| is the document length
- avgdl is the average document length
- k1 and b are free parameters (typically k1 ∈ [1.2, 2.0] and b = 0.75)
- IDF(t) is the inverse document frequency of term t

The vector component of hybrid search involves transformer-based models (like BERT, RoBERTa, or domain-specific models) that encode text into dense vectors, typically of 768-1536 dimensions. These vectors capture semantic relationships in a high-dimensional space where proximity indicates semantic similarity. Approximate Nearest Neighbor (ANN) algorithms like HNSW, IVF, or FAISS are then employed to efficiently search this vector space.

The integration of these approaches can be implemented through various architectures:

1. **Sequential Integration**: Applying one method as a filter and the other for ranking
2. **Parallel Integration**: Running both methods independently and combining results

3. **Cross-Attention Integration**: Using attention mechanisms to dynamically weight the importance of lexical and semantic signals

The ranking fusion strategies include:

- **Linear Combination**: score = α·BM25_score + (1-α)·vector_score
- **Reciprocal Rank Fusion**: score = Σ(1/(k + rank_i)) across all retrieval methods
- **Learning-to-Rank**: Using ML models to optimize the fusion based on relevance feedback

Advanced hybrid search systems also incorporate query understanding techniques, query expansion, and contextualization to refine search intent before execution. Modern implementations often utilize inverted indices for BM25 alongside specialized vector indices like HNSW graphs or quantized vector databases to optimize for both storage and retrieval efficiency.

4. Where and Why

Hybrid search is particularly valuable in scenarios where both precision and recall are critical:

Where to use:

- **Enterprise search systems**: Finding relevant documents across corporate repositories where both exact matches and conceptual similarity matter
- **E-commerce product discovery**: Helping users find products even when they use different terminology than what's in product descriptions
- **Legal and medical document retrieval**: Where finding exact terms (like statutes or medical codes) and semantically similar concepts is essential

- **Academic research databases**: Enabling researchers to discover relevant papers beyond keyword matching
- **Customer support knowledge bases**: Matching customer queries to relevant solutions even when terminology differs

Why use hybrid search:

- Improves recall by capturing semantically relevant documents that don't contain exact query terms
- Maintains precision through lexical matching for technical or specific terminology
- Provides robustness against vocabulary mismatch between queries and documents
- Handles both explicit information needs (exact matches) and implicit information needs (related concepts)
- Delivers better relevance ranking by considering both surface-level and semantic similarities

Where NOT to use:

- Simple lookup systems where exact matching is sufficient (e.g., ID-based retrieval)
- Extremely resource-constrained environments where the computational overhead isn't justified
- Applications where search intent is always explicitly and precisely stated
- Systems where explainability of results is paramount (vector components can be less interpretable)
- When dealing with highly specialized terminology where semantic models haven't been properly fine-tuned

5. Related Subjects

The most closely related subjects to hybrid search and BM25 in vector databases include:

1. **Retrieval Augmented Generation (RAG)**: Using hybrid search to retrieve relevant context for large language models to generate more accurate and factual responses
2. **Neural Information Retrieval**: The broader field of applying neural networks to information retrieval tasks
3. **Semantic Search**: Pure vector-based search approaches that focus exclusively on meaning rather than keywords
4. **Query Understanding and Expansion**: Techniques to interpret user intent and expand queries with related terms
5. **Learning to Rank (LTR)**: Machine learning approaches for optimizing search result rankings
6. **Cross-Encoder Models**: Deep learning models that evaluate query-document pairs jointly rather than independently
7. **Okapi BM25 Variants**: Extensions like BM25F (for fielded data), BM25+ (with lower bounds on term frequency), and BM25L (for long documents)
8. **Approximate Nearest Neighbor Algorithms**: Methods like HNSW, FAISS, and Annoy that enable efficient vector similarity search
9. **Knowledge Graphs**: Structured representations of knowledge that can complement hybrid search systems
10. **Natural Language Processing for Search**: The broader application of NLP techniques to improve search quali

Extra

1. Dense Vectors

History

Dense vectors in the context of machine learning emerged in the early 2010s with the rise of word embeddings. The breakthrough came with Word2Vec, introduced by Mikolay et al. at Google in 2013, which represented words as dense numerical vectors in a continuous space. This approach was further developed with GloVe (2014) and later with contextual embeddings in models like ELMo (2018), BERT (2018), and GPT (2018 onwards).

Easy Explanation

A dense vector is like a list of numbers that represents a word, sentence, image, or any piece of information in a way computers can understand. Think of it as giving everything a specific location in space.

For example, imagine a simple 2D map where "king" might be at position (5, 3) and "queen" at (5, 1). The closeness of their x-coordinates (both 5) shows they're related concepts, while the difference in y-coordinates reflects gender difference.

In real-world applications, these vectors have hundreds or thousands of dimensions, not just 2 or 3, which allows them to capture subtle relationships between concepts.

Easy example: When you ask Spotify for song recommendations, it converts all songs into dense vectors that represent features like

tempo, instruments, vocal style, etc. Songs with similar vectors sound similar, so the system recommends songs whose vectors are close to songs you already like.

Advanced Explanation

Dense vectors are high-dimensional numerical representations where semantic information is distributed across all dimensions. Unlike sparse representations (like one-hot encoding), where a single dimension corresponds to a single feature, dense vectors encode information holistically.

Formally, a dense vector $v \in \mathbb{R}^d$, where d typically ranges from 64 to 1536 dimensions in modern applications. The key property of these embeddings is that semantic similarity corresponds to geometric proximity, typically measured using cosine similarity:

$$\text{sim}(a, b) = (a \cdot b) / (||a|| \times ||b||)$$

Dense vectors are typically learned through neural networks using one of several approaches:

1. **Predictive methods**: Models like Word2Vec predict context words from target words (skip-gram) or vice versa (CBOW).
2. **Autoencoding**: Encoding inputs into lower-dimensional spaces and reconstructing them, as in autoencoders.
3. **Contrastive learning**: Training embeddings to minimize distance between semantically similar items and maximize distance between dissimilar ones, as in CLIP or SimCLR.
4. **Transformer encodings**: Using the hidden states of transformer models like BERT or Sentence-BERT to generate contextual embeddings.

These vectors enable efficient similarity search using techniques like Approximate Nearest Neighbor (ANN) algorithms, which is crucial for information retrieval applications.

Attention Mechanisms

2. Attention Mechanisms

History

Attention mechanisms were introduced around 2014-2015, initially for machine translation tasks. Bahdanau et al.'s 2014 paper "Neural Machine Translation by Jointly Learning to Align and Translate" was groundbreaking in this area. The concept reached its pinnacle with "Attention Is All You Need" by Vaswani et al. in 2017, which introduced the Transformer architecture that relies entirely on attention mechanisms without recurrence or convolution.

Easy Mode

Attention is like a spotlight that helps a model focus on the most important parts of information.

Imagine you're in a crowded room trying to listen to one person speaking. You focus your attention on their voice and somewhat ignore everything else. Similarly, attention mechanisms help AI models decide which parts of the input are most relevant for a particular task.

For example, when translating "The cat sat on the mat" to French, the model might focus strongly on "cat" when generating "chat" and on "mat" when generating "tapis."

3. Ranking Fusion

History

Ranking fusion techniques have roots in information retrieval from the 1970s and 1980s. Early methods like Borda count and Condorcet fusion were borrowed from voting theory. Modern ranking fusion techniques specifically designed for search systems emerged in the late 1990s and early 2000s with methods like CombSUM and CombMNZ. The field gained renewed importance with the rise of hybrid search systems in the late 2010s that needed to combine traditional lexical search with vector-based semantic search.

Easy Mode

Ranking fusion is like having multiple experts give you recommendations, and then combining their advice to make the best decision.

Imagine you're trying to find a good restaurant. You might ask a food critic (expert 1), check online reviews (expert 2), and ask friends (expert 3). Each gives you a list of recommendations. Ranking fusion is the process of combining these different lists into one final recommendation that takes all opinions into account.

In search systems, different algorithms (keyword-based, semantic, etc.) each produce their own rankings of results, and ranking fusion combines them into a single, optimized list.

Time To Code 🙂 - GitHub

1:

Connect to Qdrant-Database (how to run docker)

```
client = QdrantClient("localhost", port=6333)
```

2:

Collection likes "Table" is SQL

```
def create_collection():
   # create a collection
   client.create_collection(
       collection_name="test_collection_1",
       vectors_config=VectorParams(
           size=384,
           distance=Distance.COSINE,
       )
   )
```

3 :

Creating Data(as you can see we have just a vector)

```
def create_products():

   products = [

       {

           "id": 1,

           "vector": [0.1, 0.2] + [0.0] * 380 + [0.3, 0.4],

           "payload": {

               "name": "Blue T-shirt",

               "category": "clothing",

               "price": 25.99

           }

       },

   ]

   return products
```

Everything can be a vector—images, music, sounds, memories, languages, gestures, objects, patterns, thoughts, or even emotions! In terms of grammar, words themselves act as vectors of meaning, each carrying nuances and depth that shape communication.

4:

Time to add into database and then we have search method !

```
def upsert_vectors(products):
   client.upsert(
   collection_name="test_collection_1",
   points=[
       PointStruct(
           id=products["id"],
           vector=products["vector"],
           payload=products["payload"],
       )
       for products in products
   ]
)
def search():
   query_vector = [0.9, 1.0] + [0.0] * 380 + [1.1, 1.2]
   search_result = client.search(
       collection_name = "test_collection_1",
       query_vector=query_vector,
       limit=3,
```

```
    )

    return search_result
```