Enhanced Hyperbolic Vector Database: Architecture and Implementation

Technical Documentation

September 6, 2025

Contents

1 Introduction

This document provides a comprehensive technical overview of an enhanced hyperbolic vector database implementation written in Go. The system is designed to efficiently store and search high-dimensional vectors using hyperbolic geometry, specifically the Poincaré ball model, with advanced features including approximate nearest neighbor (ANN) search, persistence, and scalability optimizations.

1.1 Key Features

- **Hyperbolic Geometry**: Utilizes the Poincaré ball model for hierarchical data representation
- **ANN Indexing**: Locality Sensitive Hashing (LSH) adapted for hyperbolic space
- **Persistence**: BoltDB integration for reliable data storage
- **Scalability**: Thread-safe operations with concurrent access support
- **HTTP API**: RESTful interface for easy integration

2 Mathematical Foundation

2.1 Poincaré Ball Model

The Poincaré ball model represents hyperbolic space as the unit ball in Euclidean space, where:

- Points are represented as vectors $\mathbf{v} \in \mathbb{R}^n$ with $|\mathbf{v}| < 1$
- The boundary of the unit ball represents points at infinity
- Geodesics are circular arcs orthogonal to the boundary

2.2 Poincaré Distance Metric

The geodesic distance between two points $\mathbf{u}$ and $\mathbf{v}$ in the Poincaré ball is given by:

$$d_{\text{Poincaré}}(\mathbf{u}, \mathbf{v}) = \operatorname{arccosh}\left(1 + \frac{2|\mathbf{u} - \mathbf{v}|^2}{(1 - |\mathbf{u}|^2)(1 - |\mathbf{v}|^2)}\right) \tag{1}$$

where:

- $|\mathbf{u}|^2 = \sum_{i=1}^n u_i^2$ is the squared Euclidean norm
- $|\mathbf{u} - \mathbf{v}|^2 = \sum_{i=1}^n (u_i - v_i)^2$ is the squared Euclidean distance

This metric has the property that as points approach the boundary ($|\mathbf{v}| \to 1$), distances grow exponentially, naturally representing hierarchical relationships.

3 System Architecture

3.1 Core Components

3.2 Data Structures

3.2.1 Vector and Item Types

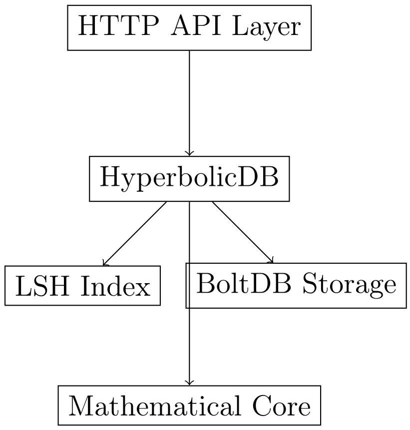

Figure 1: System Architecture Overview

```
type Vector []float64

type Item struct {
    ID     string  `json:"id"`
    Vector Vector  `json:"vector"`
}
```

3.2.2 Search Result

```
type SearchResult struct {
    Item     Item    `json:"item"`
    Distance float64 `json:"distance"`
}
```

4 Locality Sensitive Hashing (LSH) Implementation

4.1 Theoretical Background

LSH is a technique for approximate nearest neighbor search that hashes similar items to the same buckets with high probability. For hyperbolic space, we adapt traditional LSH by:

1. Using random projection vectors normalized to the Poincaré ball
2. Employing hyperbolic-aware hash functions
3. Creating multiple hash families for improved accuracy

4.2 Hash Function Design

Each hash function h_i is defined as:

$$h_i(\mathbf{v}) = \left\lfloor \frac{\langle \mathbf{v}, \mathbf{r}_i \rangle + b_i}{w} \right\rfloor \tag{2}$$

where:

- $\mathbf{r}_i$ is a random vector normalized to the Poincaré ball
- b_i is a random offset in $[0, 2\pi]$
- w is the bucket width parameter

4.3 LSH Index Structure

```
type LSHIndex struct {
    NumHashFunctions int
    NumBuckets       int
    HashFunctions    []HashFunction
    Buckets          map[uint64]*LSHBucket
    mu               sync.RWMutex
}

type HashFunction struct {
    RandomVector Vector
    Offset       float64
}

type LSHBucket struct {
    Items []string // Item IDs in this bucket
}
```

5 Database Operations

5.1 Vector Normalization

Before insertion or search, vectors are normalized to ensure they lie within the Poincaré ball:

```
norm := vectorNorm(item.Vector)
if norm >= 1.0 {
    epsilon := 1e-9
    for i := range item.Vector {
        item.Vector[i] /= (norm + epsilon)
    }
}
```

This ensures mathematical validity of the Poincaré distance calculations.

5.2 Add Operation

The add operation performs the following steps:

1. Generate UUID if no ID provided
2. Validate and normalize the vector
3. Add to in-memory store
4. Update LSH index
5. Persist to BoltDB
6. Trigger index rebuild if threshold reached

```
func (db *HyperbolicDB) Add(item Item) (string, error) {
    db.mu.Lock()
    defer db.mu.Unlock()

    // ID generation and validation...

    // Normalize for Poincare ball
    // Add to memory and index
    // Persist to disk
```

```
    // Handle index rebuilding

    return item.ID, nil
}
```

5.3 Search Operation

The search operation implements approximate nearest neighbor search:

1. Normalize query vector
2. Get candidate set from LSH index
3. Calculate exact distances for candidates
4. Sort and return top-k results

Algorithm 1 ANN Search

1: **Input:** Query vector $\mathbf{q}$, number of neighbors k
2: Normalize $\mathbf{q}$ to Poincaré ball
3: $C \leftarrow$ LSH.GetCandidates($\mathbf{q}$, $10k$)
4: **for** each $c \in C$ **do**
5: $\quad d \leftarrow$ PoincaréDistance($\mathbf{q}$, c.vector)
6: $\quad$ Add (c, d) to results
7: **end for**
8: Sort results by distance
9: **return** top k results

6 Persistence Layer

6.1 BoltDB Integration

The system uses BoltDB, an embedded key-value database, for persistence:

- **Atomic Operations**: Each add operation is wrapped in a BoltDB transaction
- **Gob Encoding**: Items are serialized using Go's gob format
- **Bucket Organization**: Separate buckets for items and index metadata

6.2 Data Loading

On startup, the system loads all persisted data:

```
func (db *HyperbolicDB) loadFromDisk() error {
    return db.boltDB.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte("items"))
        c := b.Cursor()
        for k, v := c.First(); k != nil; k, v = c.Next() {
            // Decode and load item
            // Add to LSH index
        }
        return nil
    })
}
```

7 HTTP API Interface

7.1 Endpoint Specifications

7.1.1 POST /add

Adds a new vector to the database.

Request Body:

```
{
    "id": "optional_id",
    "vector": [0.1, 0.2, 0.3, ...]
}
```

Response:

```
{
    "id": "generated_or_provided_id",
    "status": "added"
}
```

7.1.2 POST /search

Searches for nearest neighbors.

Request Body:

```
{
    "vector": [0.1, 0.2, 0.3, ...],
    "k": 10,
    "exact": false
}
```

Response:

```
{
    "results": [
        {
            "item": {"id": "item1", "vector": [...]},
            "distance": 0.123
        }
    ],
    "search_time": 15,
    "method": "ann"
}
```

7.1.3 GET /stats

Returns database statistics.

Response:

```
{
    "total_items": 1000,
    "lsh_buckets": 150,
    "hash_functions": 16
}
```

8 Performance Analysis

8.1 Time Complexity

- **Brute Force Search**: $O(n \cdot d)$ where n is number of items, d is dimensionality
- **LSH Search**: $O(k' \cdot d)$ where $k' \ll n$ is the number of candidates
- **Add Operation**: $O(H \cdot d)$ where H is number of hash functions

8.2 Space Complexity

- **Vector Storage**: $O(n \cdot d)$
- **LSH Index**: $O(n + B)$ where B is number of buckets
- **Total**: $O(n \cdot d + B)$

8.3 Configuration Trade-offs

- **More Hash Functions**: Higher accuracy, slower indexing
- **More Buckets**: Better distribution, higher memory usage
- **Larger Candidate Set**: Higher recall, slower search

9 Scalability Considerations

9.1 Current Limitations

- Single-machine deployment
- In-memory index storage
- Sequential candidate evaluation

9.2 Scaling Strategies

9.2.1 Horizontal Scaling

- Partition vectors across multiple nodes
- Implement consistent hashing for data distribution
- Use distributed consensus for metadata management

9.2.2 Vertical Scaling

- Implement disk-based index storage
- Add memory-mapped file support
- Optimize cache locality for better performance

9.2.3 Advanced Optimizations

- GPU acceleration for distance calculations
- SIMD vectorization for parallel operations
- Adaptive index parameters based on data distribution

10 Configuration and Deployment

10.1 Configuration Parameters

```
type Config struct {
    DataPath              string   // "./hyperbolic_db"
    NumHashFunctions      int      // 16
    NumBuckets           int      // 1000
    IndexRebuildThreshold int      // 1000
}
```

10.2 Deployment Requirements

- **Dependencies**: BoltDB, UUID generator
- **Disk Space**: Proportional to dataset size
- **Memory**: RAM for in-memory index and active dataset
- **Network**: HTTP server capabilities

10.3 Monitoring and Maintenance

- Monitor search latency through /stats endpoint
- Track index rebuild frequency
- Monitor disk space usage for BoltDB files
- Periodically backup database files

11 Future Enhancements

11.1 Advanced Indexing

- Implement hyperbolic-native tree structures
- Add support for dynamic index updates
- Develop learned indices for query-specific optimization

11.2 Query Optimization

- Range queries in hyperbolic space
- Batch search operations
- Query result caching

11.3 Distribution and Replication

- Multi-master replication
- Automatic failover mechanisms
- Load balancing across search replicas

12 Conclusion

The enhanced hyperbolic vector database provides a robust foundation for applications requiring hierarchical data representation and efficient similarity search. The combination of hyperbolic geometry, LSH indexing, and persistent storage creates a scalable solution suitable for modern machine learning and data science applications.

The implementation demonstrates how mathematical concepts from hyperbolic geometry can be effectively applied to practical database systems, offering unique advantages for hierarchical and tree-like data structures that are common in natural language processing, computer vision, and knowledge representation domains.

12.1 Key Achievements

- Successful adaptation of LSH for hyperbolic space
- Robust persistence layer with atomic operations
- Thread-safe concurrent access patterns
- Comprehensive HTTP API with performance monitoring
- Scalable architecture ready for future enhancements