Integrity-Preserving Vector Search: A Merkle Tree Approach to Hierarchical Navigable Small World Graph Verification

Benedict Presley - 13523067
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: presleybenedict04@gmail.com , 13523067@std.stei.itb.ac.id

***Abstract*—Vector databases are essential for retrieval-augmented generation, semantic search, and recommendation systems, but they lack inherent verification mechanisms when deployed in untrusted environments. Approximate nearest neighbor (ANN) algorithms, such as Hierarchical Navigable Small World (HNSW) graphs, cannot guarantee that query results are authentic or unmodified. This paper introduces a verifiable vector database that integrates an HNSW graph index with a Merkle tree. The system returns both the nearest neighbors and a Merkle inclusion proof for each vector against a public root hash. We detail the system architecture, workflows, and verification procedures, alongside a comprehensive threat model distinguishing data membership from search exhaustiveness. Finally, we discuss our C++ implementation choices and outline an evaluation methodology for future empirical validation.**

Keywords—approximate nearest neighbor search, Hierarchical Navigable Small World graphs, Merkle tree, data integrity

I. INTRODUCTION

Vector databases have rapidly become a foundational component of modern information retrieval pipelines. Retrieval-augmented generation (RAG) systems, semantic search engines, and recommendation systems all rely on the ability to embed unstructured data into high-dimensional vectors and to efficiently retrieve the vectors nearest to a query vector under some distance metric. Because exact nearest neighbor search becomes computationally intractable as dimensionality and dataset size grow, production systems almost universally rely on approximate nearest neighbor (ANN) algorithms, which trade a small amount of recall for gains in query latency and memory efficiency.

A consequence of this is that vector search is increasingly performed by a service that is logically or organizationally separate from the party consuming the results. A RAG application may query a third-party vector index hosted by a cloud provider. A regulated entity may be required to demonstrate that a decision-support system retrieved a specific, untampered set of records. In all of these settings, the querying party has limited or no ability to confirm that the vectors returned by the index are unmodified, rather than vectors that have been silently corrupted.

This is a direct consequence of the fact that ANN data structures, including graph-based methods such as Hierarchical Navigable Small World (HNSW) graphs were designed purely for retrieval efficiency. They carry no cryptographic notion of dataset commitment, and a query response is simply a list of vector identifiers and distances with no accompanying evidence of authenticity.

This paper addresses that gap. We present a system that augments approximate nearest neighbor index usign HNSW approach with a Merkle tree constructed over the inserted vectors. Every vector inserted into the index is simultaneously appended as a leaf of the Merkle tree, so that the tree's root hash constitutes the cryptographic commitment to the entire dataset at any point in time. When a query is executed, each returned vector is accompanied by a Merkle inclusion proof, allowing a relying party who holds only the trusted root hash to verify that a returned vector is indeed a member of the committed dataset.

II. BACKGROUND

A. The Nearest Neighbor Problem

Given a set of points (vectors) embedded in a d-dimensional space and a distance metric, the nearest neighbor problem asks, for a query vector, to find the point in the set that minimizes distance to the query. This is the foundational problem underlying semantic search, recommendation, and retrieval-augmented generation.

B. The Curse of Dimensionality

While the nearest neighbor problem is straightforward to solve exactly in low dimensions using structures such as k-d trees, the effectiveness of all known exact methods degrades sharply as dimensionality increases. This phenomenon is referred to as the curse of dimensionality. In high-dimensional spaces, the relative contrast between the nearest and farthest points in a dataset shrinks, so that exact tree-based pruning strategies lose their effectiveness and degenerate toward the cost of a linear scan. Because the embedding dimensionality

used by modern models is typically in the hundreds to low thousands, this phenomenon directly motivates the move away from exact search in practical systems.

C. Approximate versus Exact Search

Because exact nearest neighbor search is impractical at the scale and dimensionality required by modern embedding models, production systems instead solve the approximate nearest neighbor (ANN) problem. Instead of returning the optimal solution, ANN will return results that are, with high probability, among the true nearest neighbors, in exchange for sublinear query time. The quality of an ANN method is typically characterized by its recall (the fraction of true nearest neighbors actually returned) as a function of query latency.

D. Graph-Based ANN Methods

One of the most effective families of ANN algorithms in practice is graph-based search, in which the dataset is represented as a proximity graph and a query is answered by greedily traversing the graph from some entry point toward vectors closer to the query. This family is attractive because it requires no space partitioning assumptions about the data distribution and tends to achieve strong recall–latency tradeoffs in high dimensions.

E. Navigable Small World Graphs

Navigable Small World (NSW) graphs construct a proximity graph with the small-world property, that is, most nodes can be reached from any other node in a small number of hops. This is done by inserting points incrementally and connecting each new point to a bounded number of its approximate nearest neighbors among the points already inserted. The small-world property is what allows a purely local, greedy traversal (always moving to the visited neighbor closest to the query) to reach a good approximate answer in a small number of steps, without any global view of the graph.

F. Hierarchical Navigable Small World (HNSW) Graphs

HNSW extends NSW with a layered structure. Each inserted point is probabilistically assigned a maximum layer at which it will appear, with the probability of being assigned to higher layers decaying exponentially (typically governed by a parameter related to $1/ln(M)$, where M is the maximum number of connections per node per layer). Layers above the base layer contain a small fraction of the dataset and provide long-range "highways" that allow a greedy search to quickly approach the query's neighborhood before descending to denser, lower layers for fine-grained refinement.

G. Distance Metrics in Vector Search

Vector search systems commonly use Euclidean distance, squared Euclidean distance (which preserves the relative ordering of neighbors without the cost of a square root), cosine similarity, or inner product, depending on how the embedding model normalizes its output space. The choice of metric does not affect the structural properties of the graph-based index itself, only the comparison function used during traversal and neighbor selection.

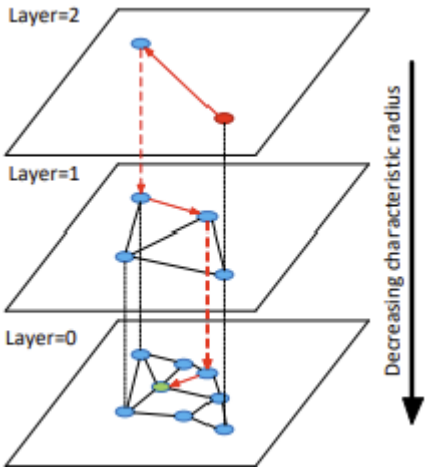

Illustration of HNSW idea. Source: https://arxiv.org/pdf/1603.09320

H. Cryptographic Hash Functions

A cryptographic hash function maps an input of arbitrary length to a fixed-length digest and is expected to satisfy three properties: preimage resistance (given a digest, it should be computationally infeasible to find any input that hashes to it), second-preimage resistance (given an input, it should be infeasible to find a different input producing the same digest), and collision resistance (it should be infeasible to find any two distinct inputs that produce the same digest. These properties are what allow a hash digest to serve as a compact, tamper-evident "fingerprint" of arbitrarily large data.

I. SHA-256

SHA-256 is a member of the SHA-2 family of cryptographic hash functions standardized by NIST in FIPS 180-4, producing a 256-bit (32-byte) digest. It remains widely used in integrity-verification and commitment schemes, including the Merkle trees used in many blockchain and certificate-transparency systems.

J. Merkle Trees as Data Commitments

Merkle tree is a binary tree in which every leaf node holds the hash of a data block and every internal node holds the hash of the concatenation of its two children's hashes. The hash stored at the root of the tree is a single, fixed-size value that constitutes a cryptographic commitment to the entire set of leaves. Under the collision-resistance assumption, it is computationally infeasible to find a different set of leaves that hashes to the same root. This property is what allows a single, short root value to stand in for an entire dataset for the purposes of integrity checks.

K. Merkle Proofs and Authentication Paths

Because each internal node's hash depends only on its two children, proving that a specific leaf is part of the tree that produced a given root does not require revealing the entire tree. Instead, a Merkle proof (or authentication path) consists of the sibling hash at each level along the path from the leaf to the root, a total of $O(log\ n)$ hashes for a tree with n leaves. A verifier who holds the trusted root hash, the leaf's data (or its

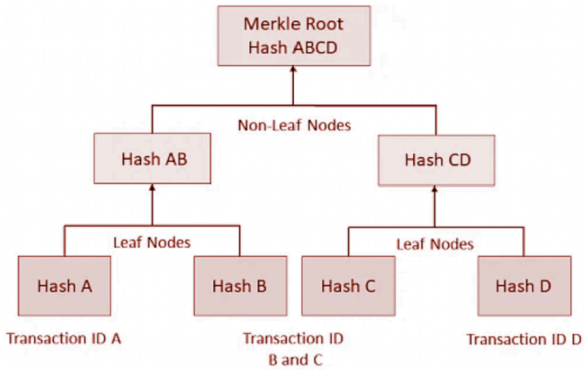

Illustration of Merkle Tree. Source: https://cryptorank.io/news/feed/7e9f5-merkle-tree-efficiency-in-blockchain-explained

hash), the leaf's position, and the proof can recompute the path of hashes from the leaf up to the root and compare the result against the trusted root, achieving $O(log\ n)$ verification cost regardless of the size of the underlying dataset.

III. METHODOLOGY

A. Overall System Design

The system under study is structured as a thin integration layer that owns two independently maintained data structures: an approximate nearest neighbor index (an HNSW-family graph) and a Merkle tree. The integration layer does not modify the internal logic of either structure; instead, it is responsible for ensuring that every vector inserted into the dataset is inserted into both structures, in the same order, so that the identifier assigned to a vector by the ANN index can be used directly as the corresponding leaf index in the Merkle tree.

B. Insertion Workflow

Conceptually, inserting a vector proceeds as follows: the vector is first handed to the HNSW index, which incorporates it into the graph structure and returns an integer identifier corresponding to its position in the index's internal storage. The same vector is then appended as a new leaf of the Merkle tree. Because both structures are populated by appending to the end of an internal list in the same call sequence, the identifier returned by the index and the position of the corresponding leaf in the Merkle tree coincide, provided insertions occur strictly sequentially and no insertion into either structure fails independently of the other. The Merkle tree itself does not recompute internal hashes on every insertion, instead it marks itself "dirty" and defers the full tree reconstruction until a root hash or a proof is actually requested.

C. Query Workflow

A query consists of a query vector and a desired result count, K. The integration layer forwards the query to the ANN index, which performs its internal graph traversal and returns a candidate list of (distance, identifier) pairs representing its best estimate of the K nearest neighbors. For each candidate, the integration layer retrieves the corresponding stored vector data from the index and requests a Merkle inclusion proof for that identifier from the Merkle tree (triggering a tree rebuild first, if the tree is currently marked dirty due to insertions that occurred since the last rebuild). The result returned to the caller for each candidate consists of the identifier, the vector's data, and the authentication path needed to verify that identifier's inclusion in the tree that produced the current root.

D. Verification Procedure

Verification of a single returned vector proceeds as follows:

- Root Acquisition: The client obtains the dataset's authoritative root hash (H_{root}) through an out-of-band, trusted channel.
- Leaf Hashing: The client computes the SHA-256 hash of the returned vector's raw floating-point bytes, mirroring the server's leaf-hashing convention.
- Path Traversal: The client traverses the provided authentication path from leaf to root. Because the proof is a flat list of sibling hashes lacking directional tags, the client determines left/right ordering dynamically based on the parity of the current node index at each level.
- Root Comparison: The final computed hash is compared against H_{root}. A match guarantees that the vector is an authentic, unmodified member of the committed dataset.

E. Design Tradeoffs

The implementation reflects several deliberate engineering and architectural tradeoffs:

- Lazy Tree Rebuilding: To avoid incremental update overhead, the Merkle tree defers internal node reconstruction until a query or root is requested. A dirty flag mitigates redundant work during high-throughput insertion bursts, amortizing costs at the expense of an $O(n)$ rebuild penalty on the first query following a batch insert.
- Sequential Identifier Coupling: Leveraging the ANN index's insertion-order identifier directly as the Merkle leaf index eliminates the memory overhead of a separate mapping table. However, this tightly couples insertion semantics, presenting a bottleneck for future support of deletions or out-of-order insertions.
- Odd-Node Duplication: When a tree level contains an odd number of nodes, the final node is paired with a duplicate of itself. This maintains a balanced binary tree structure at the cost of minor hashing redundancy.
- Decoupled Client-Side Validation: The verification logic independently replicates the SHA-256 leaf- and pair-hashing routines. While this ensures the client does not need to trust or link against the server framework, it introduces a maintenance hazard if the underlying hashing conventions or byte-ordering schemes change.

F. Threat Model

What is protected. Given a dataset root hash obtained through an out-of-band, trusted channel, a relying party can reliably detect if a returned vector has been substituted, truncated, or modified. The Merkle inclusion proof guarantees data integrity and membership, confirming that the returned vector is an authentic element of the dataset committed under that root. This mitigates post-hoc result-tampering by a compromised or adversarial host, assuming the collision resistance of SHA-256 holds.

What is not protected. Merkle inclusion proofs strictly demonstrate membership. They do not validate search completeness or execution correctness. An adversarial or malfunctioning server could execute a degraded search, return arbitrary (but genuine) vectors, or selectively omit the true nearest neighbors. Because every returned vector is an actual member of the dataset, each would successfully pass verification. Proving that an approximate nearest neighbor search was exhaustive or unbiased requires verifying the graph traversal itself, which is out of scope for this architecture. Furthermore, the current design does not support proofs of non-membership, authenticated deletions, or automated root distribution.

IV. IMPLEMENTATION

A. HNSW Index

```
class SimpleHNSW {
public:
    SimpleHNSW(int M, int ef_construct);

    int insert(const std::vector<float>& q);
    std::vector<Candidate> search(const
std::vector<float>& q, int K);
    const std::vector<float>& get_vector(int id)
const;

private:
    struct Node {
        std::vector<float> vec;
        std::vector<std::vector<int>> neighbors;
    };

    int M;
    int ef_construct;
    int entry_point;
    int max_layer;
    std::mt19937 rng;
    double level_mult;
    std::vector<Node> nodes;

    float distance(const std::vector<float>& a,
const std::vector<float>& b) const;
    int sample_layer();
    int search_layer_ef1(const std::vector<float>&
q, int ep, int lc);
    MaxHeap search_layer(const std::vector<float>&
q, int ep, int ef, int lc);
};
```

This module provides the Approximate Nearest Neighbor (ANN) search capabilities. The active implementation, SimpleHNSW, is a functioning Hierarchical Navigable Small World graph that stores vectors individually per node alongside per-layer adjacency lists.

- Insertion and Search: It supports adding vectors and K-nearest-neighbor lookup. It utilizes a multi-layer greedy descent to find entry points, followed by a bounded beam search at the target layers.
- Distance Metric: Distance is calculated using squared Euclidean distance, which preserves nearest-neighbor ordering while avoiding the computational overhead of square root operations.

B. Cryptographic Commitment

```
class MerkleTree {
public:
    MerkleTree();

    int append_leaf(const std::vector<float>&
vec_data);

    std::string get_root();

    std::vector<std::string> get_proof(int
leaf_index);

private:
    std::vector<std::string> leaves;
    std::vector<std::vector<std::string>> tree;
    bool is_dirty;

    std::string hash_vector(const
std::vector<float>& vec_data) const;
    std::string hash_pair(const std::string& left,
const std::string& right) const;
    void rebuild_tree();
};
```

This module maintains the verifiable cryptographic accumulator. It implements a standard bottom-up Merkle tree utilizing OpenSSL's EVP interface for SHA-256 hashing.

- Leaf Generation: Leaf hashes are computed directly from the raw byte representation of the floating-point vector data.
- Tree Maintenance: To optimize performance during bulk insertions, the tree utilizes an is_dirty flag. Appending a leaf simply marks the tree as dirty; internal hashes are lazily recomputed only when a root or proof is explicitly requested.
- Proof Extraction: The get_proof function generates a cryptographic proof of inclusion for a given vector. It returns a flat list of sibling hashes in leaf-to-root order. The positional orientation of these hashes (left or right child) is left to be inferred by the client based on the parity of the leaf index.

C. Integration Layer

```
struct VerifiedResult {
    int id;
    std::vector<float> vector_data;
    std::vector<std::string> merkle_proof;
};

class VerifiableVectorDB {
public:
    VerifiableVectorDB(int M = 16, int
ef_construct = 50);

    int insert(const std::vector<float>& vec);

    std::vector<VerifiedResult> query(const
std::vector<float>& q, int K);

    std::string get_root();

private:
    SimpleHNSW index;
    MerkleTree merkle_tree;
};
```

This class acts as the system orchestrator, cleanly binding the SimpleHNSW index and the MerkleTree into a unified Verifiable Vector Database.

- Data Ingestion: When insert is called, the database routes the vector into both the HNSW graph (to maintain search navigability) and the Merkle tree (to maintain the cryptographic commitment), returning the assigned sequential identifier.
- Query Execution: The query method executes the search against the HNSW index to retrieve the nearest neighbor candidates. For each hit, it extracts the corresponding vector data and requests its specific inclusion proof from the Merkle tree.
- Unified Output: The final output is packaged into a VerifiedResult structure containing the vector ID, the raw vector coordinates, and the cryptographic proof, ready for client transmission.

D. Client Verification

This module is intended strictly for the relying party. It features a standalone implementation of the SHA-256 hashing routine and the verify_merkle_proof() function. It independently reconstructs the hash path from the provided leaf data up to the root, using index parity to determine concatenation order, and validates it against a trusted root.

E. System Dependencies

The codebase deliberately minimizes external dependencies to reduce the attack surface. It relies exclusively on the C++ Standard Library for core data structures and OpenSSL for hardware-accelerated cryptographic primitives.

V. RESULTS AND ANALYSIS

A. Functional Correctness (Small-Scale Test)

To verify the end-to-end integrity of the insertion, cryptographic commitment, and verification pipeline, we first executed the `test_small.cpp` harness. This test inserts a hand-authored dataset of five 3-dimensional vectors and issues a single query for the K=2 nearest neighbors.

```
#include "test_small.h"
#include "VerifiableVectorDB.h"
#include "ClientVerifier.h"
#include <iostream>
#include <vector>
#include <cassert>

static void print_vec(const std::vector<float>& v)
{
    std::cout << "[";
    for(size_t i = 0; i < v.size(); ++i) {
        std::cout << v[i];
        if(i + 1 < v.size()) {
            std::cout << ", ";
        }
    }
    std::cout << "]";
}

void run_small_test() {
    std::cout << "small test with verification\n";

    VerifiableVectorDB db(16, 50);

    std::vector<std::vector<float>> dataset = {
        {0.10, 0.20, 0.30},
        {0.90, 0.80, 0.70},
        {0.15, 0.25, 0.35},
        {0.50, 0.50, 0.50},
        {0.95, 0.85, 0.75}
    };

    for(size_t i = 0; i < dataset.size(); ++i) {
        int id = db.insert(dataset[i]);
        std::cout << "insert id=" << id << "
vec=";
        print_vec(dataset[i]);
        std::cout << "\n";
    }

    std::string root = db.get_root();
    std::cout << "root=" << root << "\n";

    std::vector<float> query = {0.12, 0.22, 0.32};
    int k = 2;
    std::cout << "query k=" << k << "\n";

    auto results = db.query(query, k);
    for(const auto& r : results) {
        std::cout << "hit id=" << r.id << " vec=";
        print_vec(r.vector_data);
        std::cout << "\n";
```

```
        bool ok =
verify_merkle_proof(r.vector_data, r.id,
r.merkle_proof, root);
        assert(ok);
        std::cout << "verified\n";
    }

    std::cout << "small test passed\n";
}
```

The console output from this execution confirms that the HNSW index assigns sequential identifiers upon insertion, coupling properly with the Merkle tree's leaf indices.

```
small test with verification
insert id=0 vec=[0.1, 0.2, 0.3]
insert id=1 vec=[0.9, 0.8, 0.7]
insert id=2 vec=[0.15, 0.25, 0.35]
insert id=3 vec=[0.5, 0.5, 0.5]
insert id=4 vec=[0.95, 0.85, 0.75]
root=f0398e014286b6e25e7dc0bbff083306b89ce5e77d863b99e01566c243a6ee8c
query k=2
hit id=0 vec=[0.1, 0.2, 0.3]
verified
hit id=2 vec=[0.15, 0.25, 0.35]
verified
small test passed
```

Terminal output of test_small.cpp execution

B. Baseline Performance (Large-Scale Benchmark)

To establish baseline performance metrics, we executed the test_large.cpp benchmark. This configuration exercises the system with n = 1,000,000 uniformly distributed 16-dimensional vectors, configuring the index with M = 16 and ef_construct = 50, followed by 1,000 queries retrieving K = 10 nearest neighbors each.

```
#include "test_large.h"
#include "VerifiableVectorDB.h"
#include "ClientVerifier.h"
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
#include <cassert>

void run_large_test() {
    const int n = 1000000;
    const int dim = 16;
    const int m = 16;
    const int ef = 50;
    const int k = 10;
    const int num_queries = 1000;

    std::cout << "large benchmark with
verification n=" << n << " dim=" << dim << " M="
<< m << " ef=" << ef << "\n";

    VerifiableVectorDB db(m, ef);

    std::cout << "generating data\n";
    std::vector<std::vector<float>> dataset(n,
std::vector<float>(dim));
    std::mt19937 rng(1337);
    std::uniform_real_distribution<float>
dist(0.0f, 1.0f);
    for(int i = 0; i < n; ++i) {
        for(int d = 0; d < dim; ++d) {
            dataset[i][d] = dist(rng);
        }
    }

    std::cout << "inserting\n";
    auto t0 =
std::chrono::high_resolution_clock::now();
    for(int i = 0; i < n; ++i) {
        db.insert(dataset[i]);
        if((i + 1) % 200000 == 0) {
            std::cout << "insert progress " << (i
+ 1) << "/" << n << "\n";
        }
    }
    auto t1 =
std::chrono::high_resolution_clock::now();
    double insert_sec =
std::chrono::duration<double>(t1 - t0).count();
    std::cout << "insert done " << insert_sec <<
"s\n";

    std::string root = db.get_root();

    std::vector<float> query(dim);
    for(int d = 0; d < dim; ++d) {
        query[d] = dist(rng);
    }
    db.query(query, k);

    std::cout << "query+verify k=" << k << " x" <<
num_queries << " (" << (num_queries * k) << "
proofs)\n";

    double total_query_ms = 0.0;
    double total_verify_ms = 0.0;

    for(int i = 0; i < num_queries; ++i) {
        query[0] += 0.001f;

        auto q0 =
std::chrono::high_resolution_clock::now();
        auto results = db.query(query, k);
        auto q1 =
std::chrono::high_resolution_clock::now();

        for(const auto& r : results) {
            bool ok =
verify_merkle_proof(r.vector_data, r.id,
r.merkle_proof, root);
            assert(ok);
        }
        auto q2 =
std::chrono::high_resolution_clock::now();
```

```
        total_query_ms +=
std::chrono::duration<double, std::milli>(q1 -
q0).count();
        total_verify_ms +=
std::chrono::duration<double, std::milli>(q2 -
q1).count();
    }

    double avg_query_ms = total_query_ms /
num_queries;
    double avg_verify_ms = total_verify_ms /
num_queries;
    double avg_total_ms = avg_query_ms +
avg_verify_ms;

    std::cout << "query avg " << avg_query_ms <<
"ms verify avg " << avg_verify_ms << "ms total "
<< avg_total_ms << "ms " << (1000.0 /
avg_total_ms) << " qps\n";
    std::cout << "large test passed\n";
}
```

The terminal output provides our initial performance data:

```
large benchmark with verification n=1000000 dim=16 M=16 ef=50
generating data
inserting
insert progress 200000/1000000
insert progress 400000/1000000
insert progress 600000/1000000
insert progress 800000/1000000
insert progress 1000000/1000000
insert done 417.701s
query+verify k=10 x1000 (10000 proofs)
query avg 0.290234ms verify avg 0.55335ms total 0.843585ms 1185.42 qps
large test passed
```

Terminal output of test_large.cpp execution

Using this data as a foundation, a fully realized evaluation methodology would expand upon these results along the following operational axes:

- Insertion Throughput: Insertion cost in an HNSW-style index is dominated by the beam search performed at each layer a new node participates in. As observed in our baseline test, inserting one million vectors took **417.701 seconds** (roughly **2,394 insertions per second**). A fuller methodology would sweep dataset size, M, and ef_construct, both for the ANN index alone and for the combined system, in order to isolate the marginal cost contributed by appending Merkle leaves. Given the lazy-rebuild design described above, this cryptographic overhead should be negligible per insertion but will appear as periodic latency spikes whenever a root or proof is requested after a batch of insertions.
- Query and Verification Latency: Query latency in HNSW is governed primarily by the search-time ef parameter and the dimensionality and size of the dataset. Our baseline benchmark separates per-query search time from per-query verification time. The results show an average query search time of **0.290234 ms**, while the verification of the 10 resulting proofs took an **average of 0.55335 ms**, yielding a combined total latency of **0.843585 ms** (a throughput of **1185.42 QPS**). Notably, at this specific scale and configuration, the client-side cryptographic verification dominates the time spent on the actual vector search. A fuller methodology would additionally sweep ef and K across representative dataset sizes, to quantify how the overhead the verification layer adds on top of unmodified ANN search scales as these parameters change.
- Proof Generation and Verification Time and Size Scaling: Because Merkle proof length and verification cost are theoretically $O(log\, n)$ in the number of committed leaves, a complete evaluation must benchmark both proof size (number of hash entries) and the time to generate and to independently verify a proof, across several orders of magnitude of dataset size. This will confirm the logarithmic scaling and quantify the constant-factor computational cost of SHA-256 hashing at extreme scales.
- Recall and Accuracy at Varying Search Parameters: Since the underlying index is approximate, recall relative to true (exact) nearest neighbors must be measured as a function of ef_construct, ef_search, and M, using a held-out or synthetic ground-truth set. This characterizes the recall–latency tradeoff inherent in the chosen ANN configuration.
- Rebuild Amortization: Given the lazy Merkle tree rebuild strategy, an extended evaluation would benchmark the cost of a rebuild as a function of the number of pending insertions since the last rebuild. This would characterize the resulting latency distribution for queries that immediately follow large insertion batches versus queries issued after the tree has already been fully updated.

VI. CONCLUSION

In this paper, we presented a verifiable vector database that integrates a Hierarchical Navigable Small World (HNSW) index with a cryptographic Merkle tree. This architecture successfully couples approximate nearest neighbor retrieval with cryptographic commitments, enabling clients to independently verify the authenticity and membership of returned vectors via inclusion proofs. Our baseline evaluation confirmed the system's functional correctness; while the verification layer introduces client-side processing and lazy-rebuild overhead, it preserves the sublinear query advantages of the underlying HNSW graph.

Despite these capabilities, current limitations present clear opportunities for future work. First, while Merkle proofs guarantee data membership, they cannot cryptographically prove search exhaustiveness, leaving the system vulnerable to a compromised server selectively omitting true nearest neighbors. Second, the system's reliance on sequential identifier coupling currently precludes out-of-order insertions or authenticated deletions. Future research must explore dynamic accumulator structures for fully mutable indexes and graph traversal verification to guarantee search completeness,

driving this verifiable architecture toward production readiness.

SOURCE CODE REPOSITORY AT GITHUB

https://github.com/BP04/integrity-preserving-vector-search

VIDEO LINK AT YOUTUBE

https://youtu.be/TvHVMISz4jY

ACKNOWLEDGEMENT

The author sincerely thanks God Almighty for providing the strength and opportunity to complete this paper. The author also extends gratitude to Dr. Ir. Rinaldi, M.T., for his teachings and guidance throughout the II4021 Cryptography course. Finally, the author would like to thank VecML Inc. for inspiring the topic explored in this paper and for fostering an environment that encourages curiosity, innovation, and continuous learning.

REFERENCES

[1] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 4, pp. 824–836, 2020.

[2] R. C. Merkle, "Protocols for public key cryptosystems," in Proceedings of the 1980 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 1980, pp. 122–134.

[3] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," Federal Information Processing Standards Publication FIPS 180-4, U.S. Department of Commerce, Gaithersburg, MD, USA, 2015.

[4] [5] Y. A. Malkov, D. A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," Information Systems, vol. 45, pp. 61–68, 2014.

[5] J. Katz and Y. Lindell, Introduction to Modern Cryptography, 3rd ed. Boca Raton, FL, USA: CRC Press, 2020.

[6] [7] OpenSSL Software Foundation, "EVP Message Digest Routines," OpenSSL Documentation. [Online]. Available: https://www.openssl.org/docs/man3.0/man3/EVP_DigestInit.html

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2026

Benedict Presley
13523067