Vector Indexing For Databases

Github Link: https://github.com/IshanDesai1/buzzDB-index

The main goals for this project were to learn more about different types of vector indexes, and how to implement some of them. Concrete progress: I was able to implement two LSH indexes and a flat index.

Motivation:

I chose vector indexing as the topic since we spent a good amount of time in class talking about indexes for different types of data (strings, ints, custom classes, etc.) but weren't able to cover vectors. Vector indexing is a newer and important field since lots of datasets (for machine learning specifically) can be represented as a bunch of feature vectors. Accessing lots of data quickly and efficiently helps power machine learning models.

I started off by researching different types of vector indexes. From Kukreja, et. al. I found several different types of indexes such as flat index, inverted indexes, r-trees, locality sensitive hashing (LSH), hierarchical navigable small worlds index (HSNW), etc. For this project I chose to implement flat index, and two types of locality sensitive hash indexes.

A flat index stores the vectors exactly as is and then computes some distance metric (euclidean) over all the entries in the dataset in order to find the nearest neighbors. Locality sensitive hashes use some hash function that clusters similar items together (opposite of what hashes usually try to accomplish) and then the index searches just that small subset of the data which collided in order to return the nearest neighbors. I also used the ND-RTree from buzzDB as another benchmark.

Implementation:

I started with the ND RTree skeleton and then extracted common functionality which would be used by all the indexes. Some common functions were computeEuclideanDistance(), getNearestNeighbors, computeDotProduct, etc.

Implementation for the flat index was fairly simple since it just appends items to a vector.

flatIndex.h > ...

```
#ifndef FLATINDEX_H
#define FLATINDEX_H
#include "utils.h"
#include "point.h"
class FlatIndex {
    private:
    std::vector<Point> index;

    public:
    void insert(const Point& queryPoint) {
        index.emplace_back(queryPoint);
    }
    std::vector<Point> nearestNeighbor(const Point& queryPoint, unsigned long int k) {
        return getNearestNeighbors(queryPoint, index, k);
    }
    std::string get_name() {
        return "FlatIndex";
    }
};

#endif
```

The important part is that it can act as a source of truth since it will always find the closest neighbors. Implementation for the single hash LSH:

```cpp
#ifndef SINGLEHASHLSHINDEX_H
#define SINGLEHASHLSHINDEX_H
#include "utils.h"
#include <unordered_map>
#include <queue>
class SingleHashLSHIndex {
    private:
    std::unordered_map<int, std::vector<Point>> index;
    std::vector<float> unit_vector = getRandomUnitVector();
    int gamma;

    int hash(const Point& queryPoint) {
        return computeDotProduct(queryPoint.coordinates, unit_vector) / gamma;
    }

    public:
    SingleHashLSHIndex(int gamma = static_cast<int>(std::sqrt(NUM_POINTS))) : gamma(gamma) {}

    void insert(const Point& queryPoint) {
        int bucket = hash(queryPoint);
        index[bucket].emplace_back(queryPoint);
    }

    std::vector<Point> nearestNeighbor(const Point& queryPoint, unsigned long int k) {
        int bucket = hash(queryPoint);
        return getNearestNeighbors(queryPoint, index[bucket], k);
    }
    std::string get_name() {
        return "SingleHashLSHIndex";
    }
};

#endif
```

I followed the paper by Andoni and Indyk in order to come up with this hash function. The basic idea is we generate a random unit vector for the LSH. When an entry is inserted into the database, we compute the dot product with the unit vector (projecting it onto the plane), and then divide by some predetermined gamma (in order to place data into buckets for grouping). I defaulted to the square root of the number of points for gamma to cut down the search space.

Implementation for Multi Hash LSH:

```cpp
class MultiHashLSHIndex {
    private:
    std::unordered_map<std::string, std::vector<Point>> index;
    std::vector<std::vector<float>> unit_vectors;
    int gamma;
    int num_hashes;
    std::string get_bucket(const Point& queryPoint) {
        auto hashes = hash(queryPoint);
        auto bucket = concatenate(hashes);
        return bucket;
    }
    std::vector<int> hash(const Point& queryPoint) {
        std::vector<int> hashes;
        for (int i = 0; i < num_hashes; i++) {
            hashes.emplace_back(static_cast<int>(computeDotProduct(queryPoint.coordinates, unit_vectors[i]) / gamma));
        }
        return hashes;
    }
    std::string concatenate(std::vector<int> hashes) {
        std::string bucket = "";
        for (int i = 0; i < num_hashes; i++) {
            bucket += std::to_string(hashes[i]) + "-";
        }
        return bucket;
    }
    public:
    MultiHashLSHIndex(int gamma = static_cast<int>(std::sqrt(NUM_POINTS)), int num_hashes = 5) : gamma(gamma), num_hashes(num_hashes) {
        for (int i = 0; i < num_hashes; i++) {
            unit_vectors.emplace_back(getRandomUnitVector());
        }
    }
    void insert(const Point& queryPoint) {
        index[get_bucket(queryPoint)].emplace_back(queryPoint);
    }
    std::vector<Point> nearestNeighbor(const Point& queryPoint, unsigned long int k) {
        std::string bucket = get_bucket(queryPoint);
        return getNearestNeighbors(queryPoint, index[bucket], k);
    }
    std::string get_name() {
        return "MultiHashLSHIndex";
    }
};
```

Here, I followed the paper by Phillips which largely follows the same idea as single LSH hashing. The key difference is we generate many unit vectors and then concatenate the hashes in order to create our buckets. I chose to string concatenate but bitwise is also a valid approach.

The driver class has the code for timing. We measure the time taken to build the index as well as query for data.

driver.cpp > main()

```cpp
#include "common.h"
int main() {
    MultiHashLSHIndex index;
    auto points = getClusteredData();
    auto start_time = std::chrono::system_clock::now();

    for (const Point& point : points) {
        index.insert(point);
    }
    auto index_build_time = std::chrono::system_clock::now();
    std::mt19937 gen(42);
    std::uniform_int_distribution<> dis(1,100000);
    const int pointID = dis(gen);
    Point queryPoint = points[pointID];

    int k = 5;
    std::vector<Point> nearestNeighbors = index.nearestNeighbor(queryPoint, k);
    auto query_completion_time = std::chrono::system_clock::now();

    std::cout << "The " << k << " nearest neighbors to (" << queryPoint.label << "):" << std::endl;
    printPoint(queryPoint);
    for (const Point& p : nearestNeighbors) {
        printPoint(p);
    }
    auto total_time = std::chrono::duration_cast<std::chrono::milliseconds>(query_completion_time - start_time);
    auto index_time = std::chrono::duration_cast<std::chrono::milliseconds>(index_build_time - start_time);
    auto query_time = std::chrono::duration_cast<std::chrono::milliseconds>(query_completion_time - index_build_time);
    std::cout << "index: " << index.get_name()
        << " data dimensionality: " << POINT_COORDINATES
        << " num points: " << NUM_POINTS
        << " total time: " << total_time.count() << " milliseconds"
        << " index time: " << index_time.count() << " milliseconds"
        << " search time: " << query_time.count() << " milliseconds"
        << std::endl;
    return 0;
}
```

Correctness was measured against the flat index. The flat index will always return the closest neighbors because it iterates over all the data points naively and computes the distances. We only keep the k closest neighbors. In the following screenshot we see the flat index and the single hash LSH index were able to return the same top 5 closest vectors in a set of 100,000 entries, each with 4096 features:

```
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$ ./random.out
The 5 nearest neighbors to (Point_1347):
Point_1347 (91.0301, 38.9219, 68.8458, 25.159, 34.5192, 66.9846, 20.3364, 42.0971, 11.8605, 30.4489, ...)
Point_1376 (89.4611, 38.7409, 70.4815, 28.0313, 30.2413, 68.8415, 24.6217, 34.449, 12.7432, 30.445, ...)
Point_1360 (90.1657, 42.4087, 69.8517, 27.8649, 31.6785, 75.1584, 20.2927, 36.1673, 9.95881, 29.2649, ...)
Point_1408 (94.0638, 38.3622, 76.7805, 21.4572, 26.4338, 67.7994, 17.3692, 42.9556, 6.23001, 28.4387, ...)
Point_1191 (95.755, 36.6369, 77.3804, 20.8164, 35.2803, 71.2825, 23.6525, 44.1883, 13.5453, 34.3503, ...)
Point_1347 (91.0301, 38.9219, 68.8458, 25.159, 34.5192, 66.9846, 20.3364, 42.0971, 11.8605, 30.4489, ...)
index: MultiHashLSHIndex data dimensionality: 4096 num points: 100000 total time: 7163 milliseconds index time: 3747 milliseconds search time: 3415 milliseconds
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$ g++ -fdiagnostics-color -std=c++17 -O3 -Wall -Werror -Wextra driver.cpp  -o random.out
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$ ./random.out
The 5 nearest neighbors to (Point_1347):
Point_1347 (91.0301, 38.9219, 68.8458, 25.159, 34.5192, 66.9846, 20.3364, 42.0971, 11.8605, 30.4489, ...)
Point_1376 (89.4611, 38.7409, 70.4815, 28.0313, 30.2413, 68.8415, 24.6217, 34.449, 12.7432, 30.445, ...)
Point_1360 (90.1657, 42.4087, 69.8517, 27.8649, 31.6785, 75.1584, 20.2927, 36.1673, 9.95881, 29.2649, ...)
Point_1408 (94.0638, 38.3622, 76.7805, 21.4572, 26.4338, 67.7994, 17.3692, 42.9556, 6.23001, 28.4387, ...)
Point_1191 (95.755, 36.6369, 77.3804, 20.8164, 35.2803, 71.2825, 23.6525, 44.1883, 13.5453, 34.3503, ...)
Point_1347 (91.0301, 38.9219, 68.8458, 25.159, 34.5192, 66.9846, 20.3364, 42.0971, 11.8605, 30.4489, ...)
index: FlatIndex data dimensionality: 4096 num points: 100000 total time: 4817 milliseconds index time: 1454 milliseconds search time: 3362 milliseconds
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$
```

Same outcome for Multi hash LSH:

```
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$ ./random.out
The 5 nearest neighbors to (Point_66):
Point_66 (54.7463, 85.0016, 14.4168, 86.6534, 83.4736, 45.3881, 58.2108, 11.6765, 93.2189, 5.73319, ...)
Point_26472 (43.6872, 27.9819, 71.0005, 27.9044, 83.2128, 6.82923, 72.6411, 30.2656, 91.359, 28.5726, ...)
Point_28163 (72.6569, 85.8412, 80.0031, 17.6755, 19.8616, 11.0879, 12.7358, 34.3465, 94.9424, 38.3525, ...)
Point_72688 (92.2367, 66.1008, 34.0786, 56.0508, 54.6057, 55.2123, 15.6665, 46.263, 59.6098, 92.3568, ...)
Point_21872 (91.2368, 0.122039, 73.6684, 46.1162, 93.1915, 52.1882, 3.17248, 18.2941, 74.3796, 31.4315, ...)
Point_66 (54.7463, 85.0016, 14.4168, 86.6534, 83.4736, 45.3881, 58.2108, 11.6765, 93.2189, 5.73319, ...)
index: FlatIndex data dimensionality: 4096 num points: 100000 total time: 4168 milliseconds index time: 814 milliseconds search time: 3354 milliseconds
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$ g++ -fdiagnostics-color -std=c++17 -O3 -Wall -Werror -Wextra driver.cpp  -o
 random.out
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$ ./random.out
The 5 nearest neighbors to (Point_66):
Point_66 (54.7463, 85.0016, 14.4168, 86.6534, 83.4736, 45.3881, 58.2108, 11.6765, 93.2189, 5.73319, ...)
Point_26472 (43.6872, 27.9819, 71.0005, 27.9044, 83.2128, 6.82923, 72.6411, 30.2656, 91.359, 28.5726, ...)
Point_28163 (72.6569, 85.8412, 80.0031, 17.6755, 19.8616, 11.0879, 12.7358, 34.3465, 94.9424, 38.3525, ...)
Point_72688 (92.2367, 66.1008, 34.0786, 56.0508, 54.6057, 55.2123, 15.6665, 46.263, 59.6098, 92.3568, ...)
Point_21872 (91.2368, 0.122039, 73.6684, 46.1162, 93.1915, 52.1882, 3.17248, 18.2941, 74.3796, 31.4315, ...)
Point_66 (54.7463, 85.0016, 14.4168, 86.6534, 83.4736, 45.3881, 58.2108, 11.6765, 93.2189, 5.73319, ...)
index: SingleHashLSHIndex data dimensionality: 4096 num points: 100000 total time: 4837 milliseconds index time: 1370 milliseconds search time: 3466 mil
liseconds
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$
```

Accuracy test: randomly select 20 points and compare the 10 nearest neighbors to the flat index. Compute a percent and return it:

```
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$ ./random.out
accuracy test for index: MultiHashLSHIndex data dimensionality: 4096 num points: 100000 accuracy: 1.00
```

```
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$ ./random.out
accuracy test for index: SingleHashLSHIndex data dimensionality: 512 num points: 100000 accuracy: 1
```

Runtime:

In order to measure performance we insert 100,000 points for each model, with data dimensionality varying from 128 to 4096 in powers of 2. We also create random data and clustered data. The raw results can be found here. The graphs are included on the next page of the report.

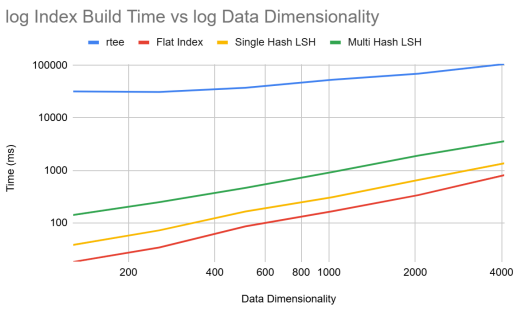
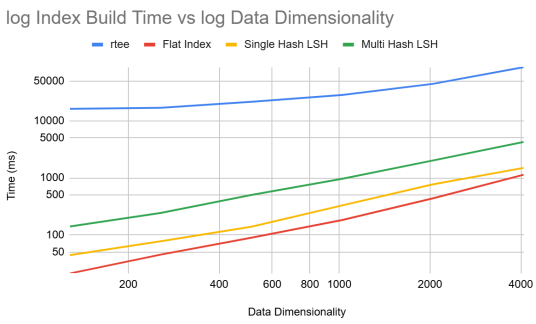
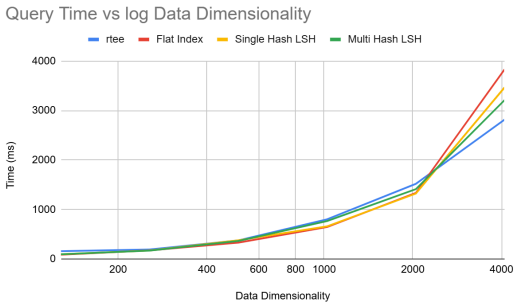
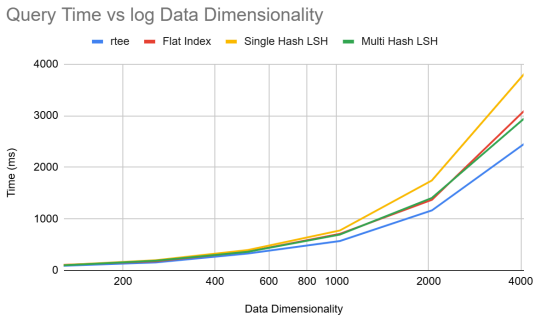
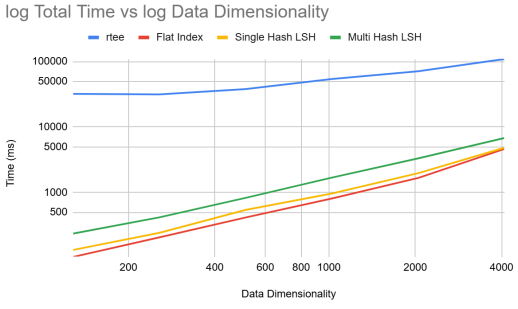
Key findings:

The fastest overall time came down to the flat index. It far outperformed the Multi-Hash LSH and the R-Tree but was similar in performance to the Single-Hash LSH. However on closer inspection we see query time for the flat index increased the fastest out of all the indices. It performed the fastest because the index build time was as simple as possible. This is not ideal for the database since the index is going to be created infrequently but queried often.

We see the opposite in the r-tree. The r-tree has the fastest query time but the slowest time to build the index. This is a better model for most database systems due to frequency of querying. The r-tree also performed significantly better when the data was clustered instead of truly random. This is due to how it creates its bounding boxes. The other indexing methods did not see a significant difference from clustered vs random data. The r-tree also took significantly more time to index data as the dimensionality increased whereas the other indexes increased mostly linearly.

Single and Multi Hash LSH performed about similarly to the flat index in both building the index and querying. Despite placing the data into separate buckets based off the indexing strategy, it seems the bulk of the time came from other unoptimized parts of the computation.

Note: I wanted to test dimensionality greater than 4096 but was running into memory issues. Adding a buffer manager would add more variance to the timing so I decided against that.

	Random Data	Clustered Data
Index Time		
Query Time		
Total Time		

Next steps:

I plan on writing a more robust accuracy metric by 11/22/24. Comparison against flat indexing is the correct methodology for measuring accuracy but I would like to randomly sample 10-20 points, find their nearest 10 or so neighbors and then compute how many of them are the same between the two algorithms. This would give a concrete number to see the correctness of each algorithm. **Update**: Finished 11/18/2024

```
ishan@DESKTOP-5IDISGQ:/mnt/c/Users/ishan/gatech/classes/CS6422/buzzDB-index$ ./random.out
accuracy test for index: SingleHashLSHIndex data dimensionality: 512 num points: 100000 accuracy: 1
```

As a stretch goal I could implement HNSW. HNSW is a graph-based index with accurate and fast query times. where vectors are represented as nodes and edges represent the distance between them. The graph has m layers where nodes are randomly sampled to be on a given layer. The topmost layer is sparse and gradually more elements are present as you go down the layers of the graph until the bottom layer has all of the nodes. The basic idea is since the top layer is sparse you can quickly get to the “closest” vector within that layer and then traverse downwards and repeat the process until you find the actual closest vector. This index is used in several large companies/technologies such as Apache, Facebook AI, MongoDB, etc.

Work Cited

Andoni, Alexandr, and Piotr Indyk. “Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions.” *People.Csail.Mit.Edu*, 2008, people.csail.mit.edu/indyk/p117-andoni.pdf.

Kukreja, Sanjau, et al. “Vector Databases and Vector Embeddings-Review.” *Ieeexplore.Ieee.Org*, 2023, www.ieee.org/publications/subscriptions/products/mdl/ieeexplore-access.html.

Malkov, Yu. A., and D. A. Yashunin. “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs.” *arXiv.Org*, 14 Aug. 2018, arxiv.org/abs/1603.09320.

Phillips, Jeff. *Locality Sensitive Hashing*, University of Utah, 2013, users.cs.utah.edu/~jeffp/teaching/cs5955/L6-LSH.pdf.

Appendix:

Raw results dataset: BuzzDB Extra Credit Project Data (https://docs.google.com/spreadsheets/d/1GFwfK8x3xyLt80VwRpFFZhFAv78Wgg0X6DC3jJ5vT88/edit?gid=188045022#gid=188045022)

Github Link: https://github.com/IshanDesai1/buzzDB-index