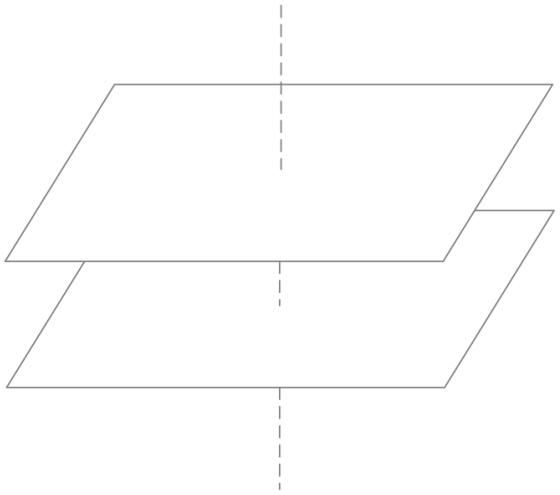
Introduction



Write You a Vector Database

In this tutorial, you will learn how to add vector extensions to an existing relational database system.

Vector Databases

Vector databases store vectors, and vectors are arrays of decimal values. Vector databases support efficient query and retrieval of vector data by storing vectors with compact format and creating vector indexes to accelerate similarity searches. Vector databases can be either a vector-oriented standalone database product that provides the above functionalities (i.e., Pinecone and Milvus), or a relational/NoSQL database system with vector extensions (i.e., PostgreSQL with pgvector, or Elasticsearch with vector searches).

Diving a little bit into PostgreSQL with pgvector, the extension adds the following vector capabilities on top of PostgreSQL.

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding vector(3)); -- vector
type
SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5; -- computing
nearest neighbors
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops); -- create vector
indexes
```

We will implement the same functionalities in this tutorial.

About This Tutorial

We have two variants of this tutorial: the C++ version is based on CMU-DB's BusTub system (used in CMU's Database Systems course), and the to-be-expected Rust version will be based on the RisingLight educational database system. Both of them are (1) relational DBMS that users can interact with using ANSI SQL interfaces (2) educational systems that are super easy to get started and do hands-on works. You will learn how to store the vectors, do similarity searches by computing nearest neighbors, and build vector indexes like IVFFlat and HNSW over the data throughout this tutorial.

We provide full solution for this tutorial on the `vectordb-solution` branch at https://github.com/skyzh/bustub-vectordb except the part of the tutorial that overlaps with CMU-DB's Database Systems course.

> Some part of this tutorial overlaps with Carnegie Mellon University's Database Systems course. Please follow the instructions in the tutorial on whether you can make some specific part of your implementation public on the Internet.

There are many other vector database tutorials on the Internet. The primary features of this tutorial: we focus on building a vector extension on a relational database system (instead of REST APIs or some other interfaces), and we focus on the implementation of the vector indexes (instead of directly using the Faiss library).

Prerequisites

You will need to know the basics of relational database systems and have some experience of system programming before starting this tutorial. You *do not* need to have a solid experience with database systems (i.e., complete the 15-445/645 Database Systems course). We assume you do not know much about the internals of the database systems and will have necessary contents to guide you through the system in the tutorial.

To complete the C++ version of this tutorial, you will need to know modern C++. The BusTub system is written in C++17 and you should feel comfortable with working on a C++17 codebase. You can quickly learn the necessary C++17 features that we will use in this tutorial by completing the C++ primer project of CMU-DB's Database Systems course.

To complete the Rust version of this tutorial, you will need to know Rust and feel comfortable with working on a Rust codebase.

Community

You may join skyzh's Discord server and study with the write-you-a-vector-db community.

About the Author

As of writing (at the beginning of 2024), Chi obtained his master's degree in Computer Science from Carnegie Mellon University and his bachelor's degree from Shanghai Jiao Tong University. He has been working on a variety of database systems including TiKV, AgateDB, TerarkDB, RisingWave, and Neon. Since 2022, he worked as a teaching assistant for CMU's Database Systems course for three semesters on the BusTub educational system, where he added a lot of new features and more challenges to the course (check out the re-designed query execution project and the super challenging multi-version concurrency control project). Besides working on the BusTub educational system, he is also a maintainer of the RisingLight educational database system. Chi is interested in exploring how the Rust programming language can fit in the database world. Check out his previous tutorial on building a vectorized expression framework type-exercise-in-rust and on building a LSM-based storage engine mini-lsm if you are also interested in that topic.

> Disclaimer: This tutorial is not affiliated with Carnegie Mellon University or CMU-DB Group. The C++ version of this tutorial is NOT a part/extension of CMU's 15-445/645 Database Systems course.

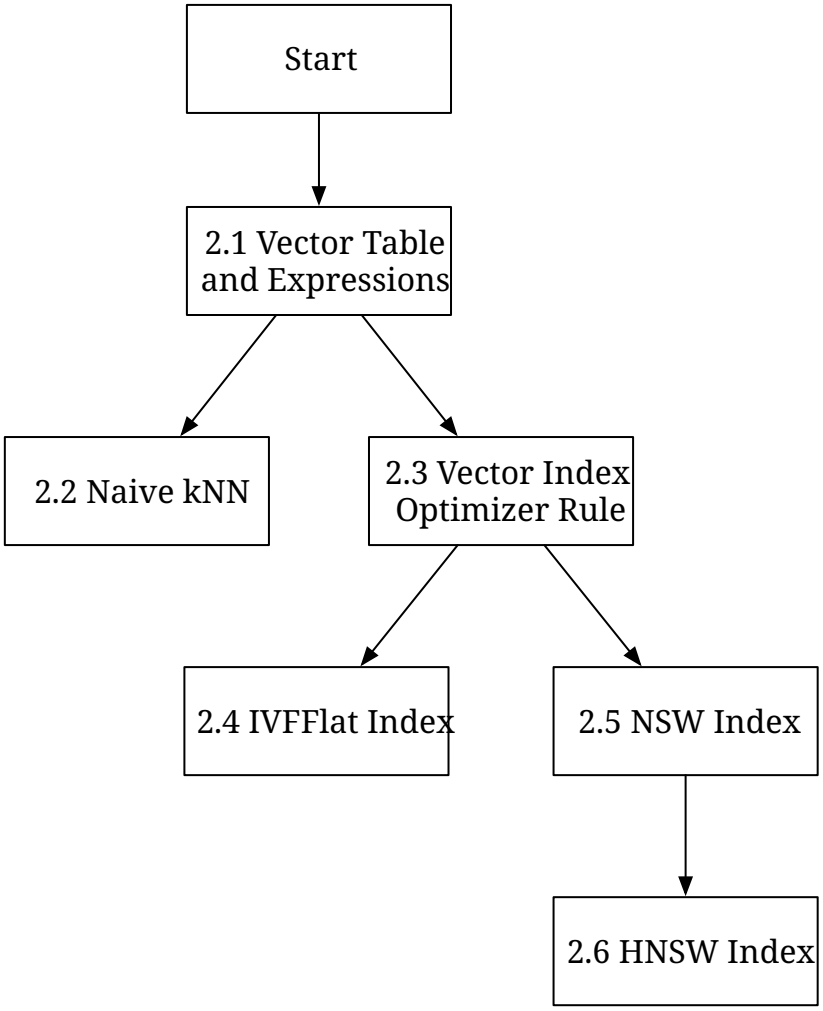
Your feedback is greatly appreciated. Welcome to join our Discord Community.

The C++ Way (over BusTub)

In this section, we will implement a vector database extension over the BusTub educational system.

Overview

You may approach this tutorial by implementing the things that interest you most first. The following figure is a learning path diagram that visualizes the dependencies between the chapters.



Environment Setup

You will be working on a modified codebase based on Fall 2023's version of BusTub.

```
git clone https://github.com/skyzh/bustub-vectordb
```

At minimum, you will need `cmake` to configure the build system, and `llvm@14` or Apple Developer Toolchain to compile the project. The codebase also uses clang-format and clang-tidy in `llvm@14` for style checks. To compile the system,

```
mkdir build && cd build
cmake ..
make -j8 shell sqllogictest
```

Then, you can run `./bin/bustub-shell` to start the BusTub SQL shell.

```
./bin/bustub-shell

bustub> select array [1.0, 2.0, 3.0];
+-------------+
| __unnamed#0 |
+-------------+
| [1,2,3]     |
+-------------+
```

In BusTub, you can use the `array` keyword to create a vector. The elements in a vector must be of decimal (double) type.

Extra Content

What did we change from the CMU-DB's BusTub codebase

The `bustub-vectordb` repository implements some stub code for you so that you can focus on the implementation of the vector things.

Buffer Pool Manager. We have a modified version of the table heap and a mock buffer pool manager. All the data stay in memory. If you are interested in persisting everything to disk, you may revert the buffer pool manager patch commit (remember to revert both the buffer pool manager and the table heap), and start from the 15-445/645 project 1 buffer pool manager.

Vector Expressions. The modified BusTub codebase has support for vector distance expressions.

Vector Indexes. The codebase adds support for vector indexes besides B+ tree and hash table indexes.

Vector Executors. With the vector index conversion optimizer rule and the vector index scan executor, users will be able to scan the vector index when running some specific k-nearest neighbor SQLs.

Your feedback is greatly appreciated. Welcome to join our Discord Community.

Vector Expressions and Storage

In this chapter, we will walk through some ramp-up tasks to get familiar with the BusTub system. You will be able to store vectors inside the system and compute vector distances after finishing all required tasks.

The list of files that you will likely need to modify:

```
src/execution/insert_executor.cpp                     (recommended to git-
ignore)
src/execution/seq_scan_executor.cpp                   (recommended to git-
ignore)
src/include/execution/executors/insert_executor.h     (recommended to git-
ignore)
src/include/execution/executors/seq_scan_executor.h   (recommended to git-
ignore)
src/include/execution/expressions/vector_expression.h
```

> **WARNING:** In this tutorial, you will implement a simplified version of the sequential scan and the insert executor. These implementations are different from the 15-445 course but we still recommend you not including these files in your git repository.

Computing Distances

In `vector_expressions.h`, you can implement some distance functions that we will use when building vector indexes and finding the k-nearest neighbors. You will need to implement 3 distance functions in `ComputeDistance`.

L2 distance (or Euclidean distance)

$$= \|\mathbf{a}-\mathbf{b}\| = \sqrt{(a_1-b_1)^2+(a_2-b_2)^2+\cdots+(a_n-b_n)^2}$$

Cosine Similarity Distance

$$= 1 - \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} = 1 - \frac{a_1b_1+a_2b_2+\cdots+a_nb_n}{\sqrt{a_1^2+\cdots+a_n^2}\sqrt{b_1^2+\cdots+b_n^2}}$$

Negative Inner Product Distance

$$= -\mathbf{a}\cdot\mathbf{b} = -(a_1b_1+a_2b_2+\cdots+a_nb_n)$$

Insertion and Sequential Scan

In this task, you will learn how BusTub represents data and how to interact with vector indexes.

Table Heap and Tuple Format

In BusTub, all table data are stored in a structure called table heap. The table heap is row-based storage that stores a collection of tuples on the disk. See the `TableHeap` class for more information.

A tuple is a serialized representation of a row in the database. For example, a tuple of integer `1, 2, 3` will be serialized into a tuple with hex representation of:

```
01 00 02 00 03 00
```

This serialized data will be stored on disk. To recover the values from the serialized value, you will need to provide a schema. For example, the schema for 3 integers is `[Int, Int, Int]`. With the schema, we can decode the tuple to three integer values.

In BusTub, there are 3 important data-representation structures.

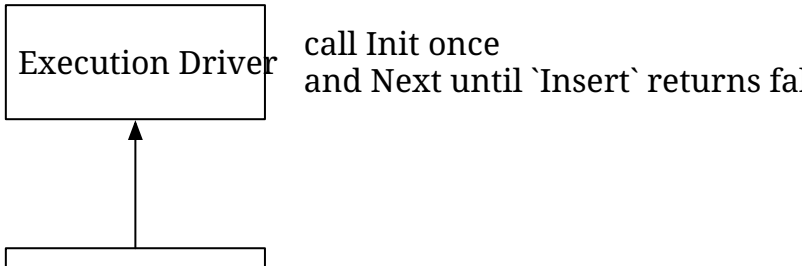
- `Tuple`: as above, serialized bytes that represent some values.
- `Schema`: as above, number of elements and the data type of each element, indicating the correct way to decode the tuple.
- `Value`: an in-memory representation of a value with type information, where users can convert it to a primitive type.

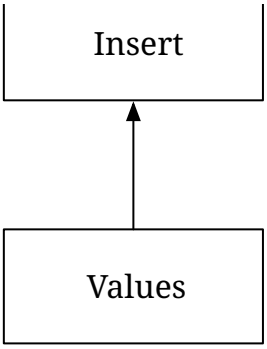
Related Lectures

- Database Storage Part 2 (CMU Intro to Database Systems)

Execution Model

BusTub uses the Volcano execution model. Each query executor defines an `Init` and a `Next` member functions. The query executors are organized in a tree. The top-most execution driver (see `execution_engine.h`) will call `Init` once and `Next` until the top-most executor returns false, which indicates there are no more tuples to be produced. Each executor will initialize and retrieve the next tuple from their child executors.





Related Lectures

- Query Execution Part 1 (CMU Intro to Database Systems)
- Query Execution Part 2 (CMU Intro to Database Systems

Insertion

SQL queries like `INSERT INTO` will be converted into an insert executor in the query processing layer.

```
bustub> explain (o) insert into t1 values (array [1.0, 2.0, 3.0]);
=== OPTIMIZER ===
Insert { table_oid=24 }
  Values { rows=1 }
```

We have already provided the implementation of values executor that produces the user-provided values in the insert statement. You will need to implement the insert executor. The insert executor should do all the insertion works in the `Init` function and return a single number indicating rows processed in the `Next` function.

You will also need to get all vector indexes from the catalog (using the executor context), and insert the corresponding data into the vector index. All vector indexes can be dynamically casted to `VectorIndex*`. You can use `index->GetKeyAttrs()` to retrieve the vector column to build index on, and it should always be one column of vector type. After you know which column to build the index, you can extract the `Value` from the child executor (values) output tuple, and then use `Value::GetVector` to convert it to a `std::vector<double>` vector. With that, you may call `index->InsertVectorEntry` to insert the data into vector indexes.

You can get necessary information (i.e., table oid) from the query plan.

Sequential Scan

In sequential scan, you may create a table iterator and store it in the executor class in `Init`, and emit tuple one by one in `Next`. You do not need to consider the case that a

tuple might have been deleted. In this tutorial, all tables are append-only.

You may get the table heap structure by accessing the catalog using executor context. After getting the table heap, you may create a table iterator by using `MakeIterator`. You may retrieve the current tuple pointed by the iterator by calling `TableIterator::GetTuple`, and move to the next tuple by using prefix `++` operator. `TableIterator::IsEnd` indicates whether there are more tuples in the table heap.

You can get necessary information (i.e., table oid) from the query plan.

Testing

You can run the test cases using SQLLogicTest.

```
make -j8 sqllogictest
./bin/bustub-sqllogictest ../test/sql/vector.01-insert-scan.slt --verbose
```

The test cases do not do any correctness checks and you will need to compare with the below output by yourself. Note that we do not test index insertions for now, and you can validate if your implementation of index insertion is correct in later tasks.

▸ Reference Test Result

Bonus Tasks

At this point, you should have implemented the basic read and write flows when a user requests to store some data in the system. You may choose to complete the below bonus tasks to challenge yourself.

Implement the Buffer Pool Manager

We already provide you a mock buffer pool manager and a table heap so that you do not need to interact with the disk and persist data to the disk. You can implement a real buffer pool manager based on project 1 of 15-445/645. Remember to revert both the buffer pool manager change and the table heap change before starting implementing the project, otherwise there will be memory leaks and deadlock issues.

Implement Delete and Update

You may implement the delete and update executor to update the data in the table heap and the vector indexes. When you delete or update an entry, BusTub does not actually remove the data from the table heap. Instead, it sets the deletion marker. Therefore, you

can use the `UpdateTupleMeta` function when deleting a record, and convert update to a deletion followed by an insertion. Also remember to update the vector indexes if you implement these two executors. You might need to add new member functions to `VectorIndex` class to remove data from vector indexes.

Insert Validation

It is possible that a user might insert a vector of dimension 3 or 5 to a `VECTOR(4)` column. In insertion executor, you may do some validations to ensure the received tuples are of the correct schema.

Again, please keep your implementation in this section private and do not put them in a public repo, especially if you want to approach the bonus tasks, because they overlap with the CMU-DB's Database Systems course projects.

Naive K-Nearest Neighbors

In this task, we will implement a naive k-nearest neighbor search by simply scanning the table, computing the distance, and retrieving the k-nearest elements.

The list of files that you will likely need to modify:

```
src/execution/sort_executor.cpp                          (KEEP PRIVATE)
src/execution/topn_executor.cpp                          (KEEP PRIVATE)
src/execution/limit_executor.cpp                         (KEEP PRIVATE)
src/include/execution/executors/sort_executor.h          (KEEP PRIVATE)
src/include/execution/executors/topn_executor.h          (KEEP PRIVATE)
src/include/execution/executors/limit_executor.h         (KEEP PRIVATE)
src/optimizer/sort_limit_as_topn.cpp                     (KEEP PRIVATE)
```

> **WARNING:** Some part of this chapter overlaps with the CMU-DB's Database System course and we ask you NOT to put the above files in a public repo.

Naive K-Nearest Neighbors

In vector databases, one of the most important operations is to find nearest neighbors to a user-provided vector in the vector table using a specified vector distance function. In this task, you will need to implement some query executors in order to support nearest neighbor SQL queries as below:

```
CREATE TABLE t1(v1 VECTOR(3), v2 integer);
SELECT v1 FROM t1 ORDER BY ARRAY [1.0, 1.0, 1.0] <-> v1 LIMIT 3;
```

The query scans the table, computes the distances between vectors in the table and `<1.0, 1.0, 1.0>`, and returns 3 nearest neighbors to the query vector. Explain the query, and you will see the query plan as below.

```
bustub> explain (o) SELECT v1 FROM t1 ORDER BY ARRAY [1.0, 1.0, 1.0] <-> v1
LIMIT 3;
=== OPTIMIZER ===
Limit { limit=3 }
  Sort { order_bys=[("Default", "l2_dist([1.000000,1.000000,1.000000],
#0.0)")] }
    Projection { exprs=["#0.0"] }
      SeqScan { table=t1 }
```

BusTub uses limit and sort executor to process this SQL query. In this part, you will need to implement these two executors, and optimize them into a top-k executor which computes nearest-k neighbors more efficiently.

Sort and Limit Executor

The sort executor pulls all the data from the child executor, sort them in memory, and emit the data the the parent executor. You should order the data as indicated in the query plan. In the above example, the query plan indicates the data to be ordered by `l2_dist([1.000000,1.000000,1.000000], #0.0)` in the default (ascending) order. `#0.0` is a column value expression which returns the *first* column in the *first* child executor. You may use `Evaluate` on an expression to retrieve the distance to be ordered.

After getting all the data and RIDs from the child executor in sort executor's `Init` function, you can use `std::sort` to sort the tuples. The comparison function should be implemented as a for loop over the query plan's order-by requirement. You can then implement sort executor's `Next` function as emitting sorted tuples one by one.

Limit executor returns the first `limit` number of elements from the child executor. You can get all necessary information in the query plan, and stop getting data from the child executor and emitting to the parent executor when the limit is reached.

After implementing these two executors, you should be able to get k-nearest neighbors of the base vector in BusTub.

Testing Sort + Limit

At this point, you can run the test cases using SQLLogicTest.

```
make -j8 sqllogictest
./bin/bustub-sqllogictest ../test/sql/vector.02-naive-knn.slt --verbose
```

The test cases do not do any correctness checks and you will need to compare with the below output by yourself. Your result could be different from the reference solution because your way of breaking the tie (i.e., when two distances are the same) might be different.

▸ Reference Test Result

Top-N Optimization

To retrieve the k-nearest neighbor, you do not need to sort the entire dataset. You may use a binary heap (`priority_queue` in C++ STL) to compute the same result set with more efficiency. This requires you to combine sort and limit executor into a single top-n executor.

The first step is to write an optimizer rule to convert sort and limit into a top-n executor. You will need to match a limit plan node with a sort child plan node, get necessary information (order-bys and limit), and then create a top-n plan node. There are already some example optimizer rule implementaions and you may refer to them.

Then, you may implement the top-n executor. The logic is similar to the sort executor that you do all the computation work in the `Init` function and then emit the top-k tuples in the `Next` function one by one. You will need to maintain a max-heap that contains the minimum k elements when scanning from the child executor.

Related Lectures

- Query Planning & Optimization (CMU Intro to Database Systems)

Testing TopN

At this point, you can run the test cases using SQLLogicTest.

```
make -j8 sqllogictest
./bin/bustub-sqllogictest ../test/sql/vector.02-naive-knn.slt --verbose
```

The test cases do not do any correctness checks and you will need to compare with the below output by yourself. Your result could be different from the reference solution because your way of breaking the tie (i.e., when two distances are the same) might be different. Note that you should see `TopN` instead of sort and limit plan nodes in your explain result.

▸ Reference Test Result

Bonus Tasks

Now that you have a better view of how BusTub works, you may choose to complete the below bonus tasks to enhance your understanding and challenge yourself.

Construct vectors from string

Currently, the query processing layer only supports creating a vector from array keyword and a list of decimal values like `SELECT ARRAY [1.0, 1.0, 1.0]`. You may extend the syntax to support (1) create a vector from integers `SELECT ARRAY [1, 1.0, 1]` and (2) create a vector from string `SELECT '[1.0, 1.0, 1.0]'::VECTOR(3)`.

Again, please keep your implementation in this section private and do not put them in a public

repo because they overlap with the CMU-DB's Database Systems course projects.

Your feedback is greatly appreciated. Welcome to join our Discord Community.

Matching a Vector Index

Before building and using vector indexes, we will need to identify SQL queries that can be converted to an index scan, and use the index when it is possible to do so. In this task, you will implement the code to identify such queries and convert them into a vector index search plan node.

The list of files that you will likely need to modify:

```
src/optimizer/vector_index_scan.cpp
src/optimizer/optimizer_custom_rules.cpp
```

Related Lectures

- Query Planning & Optimization (CMU Intro to Database Systems)

Matching the Plan Structure

```
CREATE TABLE t1(v1 VECTOR(3), v2 integer);
CREATE INDEX t1v1hnsw ON t1 USING hnsw (v1 vector_l2_ops) WITH (m = 5,
ef_construction = 64, ef_search = 10);
```

You will need to convert queries like below to a vector index scan.

```
EXPLAIN (o) SELECT v1 FROM t1 ORDER BY ARRAY [1.0, 1.0, 1.0] <-> v1 LIMIT 2;
EXPLAIN (o) SELECT * FROM t1 ORDER BY ARRAY [1.0, 1.0, 1.0] <-> v1 LIMIT 2;
EXPLAIN (o) SELECT v1, ARRAY [1.0, 1.0, 1.0] <-> v1 FROM t1 ORDER BY ARRAY
[1.0, 1.0, 1.0] <-> v1 LIMIT 2;
EXPLAIN (o) SELECT v2, v1 FROM t1 ORDER BY ARRAY [1.0, 1.0, 1.0] <-> v1 LIMIT
2;
```

...where the distance expressions appear in the order-by expressions.

You may choose to run the conversion rule before or after the top-n optimization rule. If you choose to invoke this rule before converting to top-n, you will need to match sort and limit plan nodes. Otherwise, you may want to match the top-n plan node. The rule order can be found in `optimizer_custom_rules.cpp`.

There are three cases you should consider:

Case 1: TopN directly followed by SeqScan

```
TopN { n=2, order_bys=[("Default", "l2_dist([1.000000,1.000000,1.000000],
#0.0)")]}
  SeqScan { table=t1 }
```

Case 2: TopN followed by Projection

```
TopN { n=2, order_bys=[("Default", "l2_dist([1.000000,1.000000,1.000000],
#0.0)")]}
  Projection { exprs=["#0.0", "l2_dist([1.000000,1.000000,1.000000], #0.0)"]
}
    SeqScan { table=t1 }
```

Case 3: TopN followed by Projection with column reference shuffled

```
TopN { n=2, order_bys=[("Default", "l2_dist([1.000000,1.000000,1.000000],
#0.1)")]}
  Projection { exprs=["#0.1", "#0.0"] }
    SeqScan { table=t1 }
```

As a rule of thumb, you should always figure out which vector column does the user wants to compare with. In all cases, it will be the first column of the sequential scan, that is to say, the first column in the vector table.

In case 1, as the top-n node directly references the first column `#0.0`, you can easily retrieve the column.

In case 2, the top-n node references `#0.0`, which is the first column of the projection node, and the first column of the projection column is `#0.0`.

In case 3, the top-n node references `#0.1`, which is the second column of the projection node, which eventually references `#0.0` of the sequential scan executor.

After identifying the index column, you may iterate the catalog and find the first available index to use. Then, you may replace the plan with a vector index scan plan node.

The vector index scan plan node will emit the table tuple in its original schema order. Therefore, in case 2 and 3, you will also need to add a projection after the vector index scan plan node in order to keep the semantics of the query.

Index Selection Strategy

In the `Optimizer` class, you may make use of the `vector_index_match_method_` member variable to decide the index selection strategy.

- `unset` or empty: match the first vector index available.

- `hnsw`: only match the HNSW index.
- `ivfflat`: only match the IVFFlat index.
- `none`: do not match any index and do exact nearest-neighbor search.

Testing

You can run the test cases using SQLLogicTest.

```
make -j8 sqllogictest
./bin/bustub-sqllogictest ../test/sql/vector.03-index-selection.slt --verbose
```

The test cases do not do any correctness checks and you will need to compare with the below output by yourself.

▸ Reference Test Result

Bonus Tasks

Optimize more queries

There are many other queries that can be converted to a vector index scan. You extend the range of queries that can be converted to a vector index scan in your optimizer rule. For example,

```
EXPLAIN (o) SELECT * FROM (SELECT v1, ARRAY [1.0, 1.0, 1.0] <-> v1 as
distance FROM t1) ORDER BY distance LIMIT 2;
```

The query plan for this query is:

```
TopN { n=2, order_bys=[("Default", "#0.1")]}
  Projection { exprs=["#0.0", "l2_dist([1.000000,1.000000,1.000000], #0.0)"]
}
    SeqScan { table=t1 }
```

This query saves one extra computation of the distance compared with the one in the sqllogictest. You may modify your optimizer rule to convert this query into a vector index scan.

IVFFlat (Inverted File Flat) Index

IVFFlat (InVerted File Flat) Index is a simple vector index that splits data into buckets (aka. quantization-based index) so as to accelerate vector similarities search.

The list of files that you will likely need to modify:

```
src/include/storage/index/ivfflat_index.h
src/storage/index/ivfflat_index.cpp
```

Related Readings

- Product Quantization from Pinecone's Faiss Manual. You may skip the product quantization part and take a look at the IVF visualization.

Overview

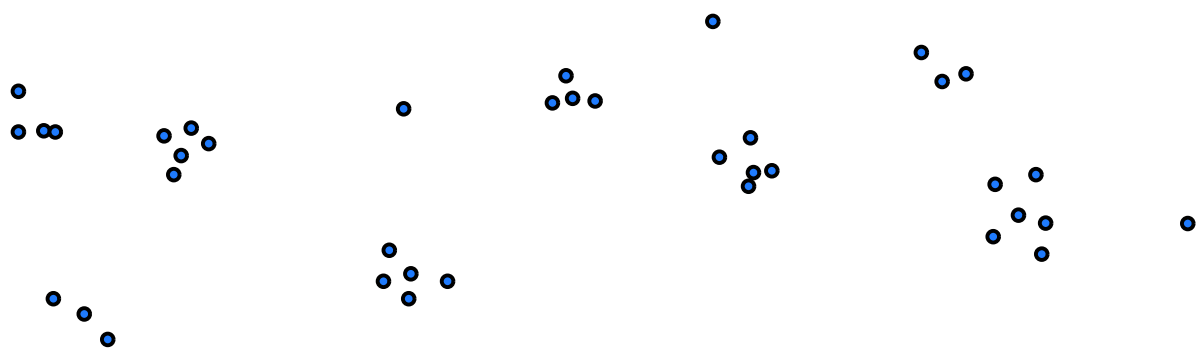
The core of IVFFlat index is to build some buckets for vector data based on distances. By splitting buckets, the index can narrow down the range of data to be searched for each query, so as to accelerate vector similarity searches.

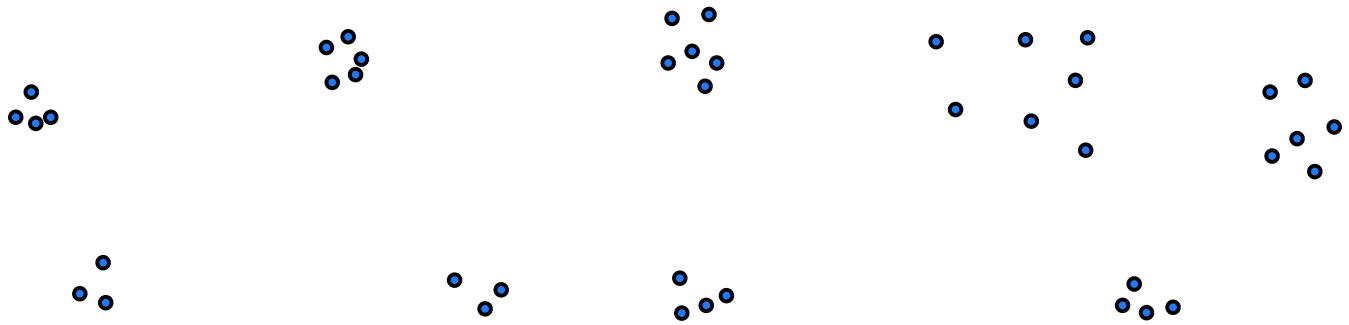
IVFFlat index search yields approximate nearest neighbors, which means that the result of the index lookup might not be 100% the same as the naive kNN implementation.

Index Building

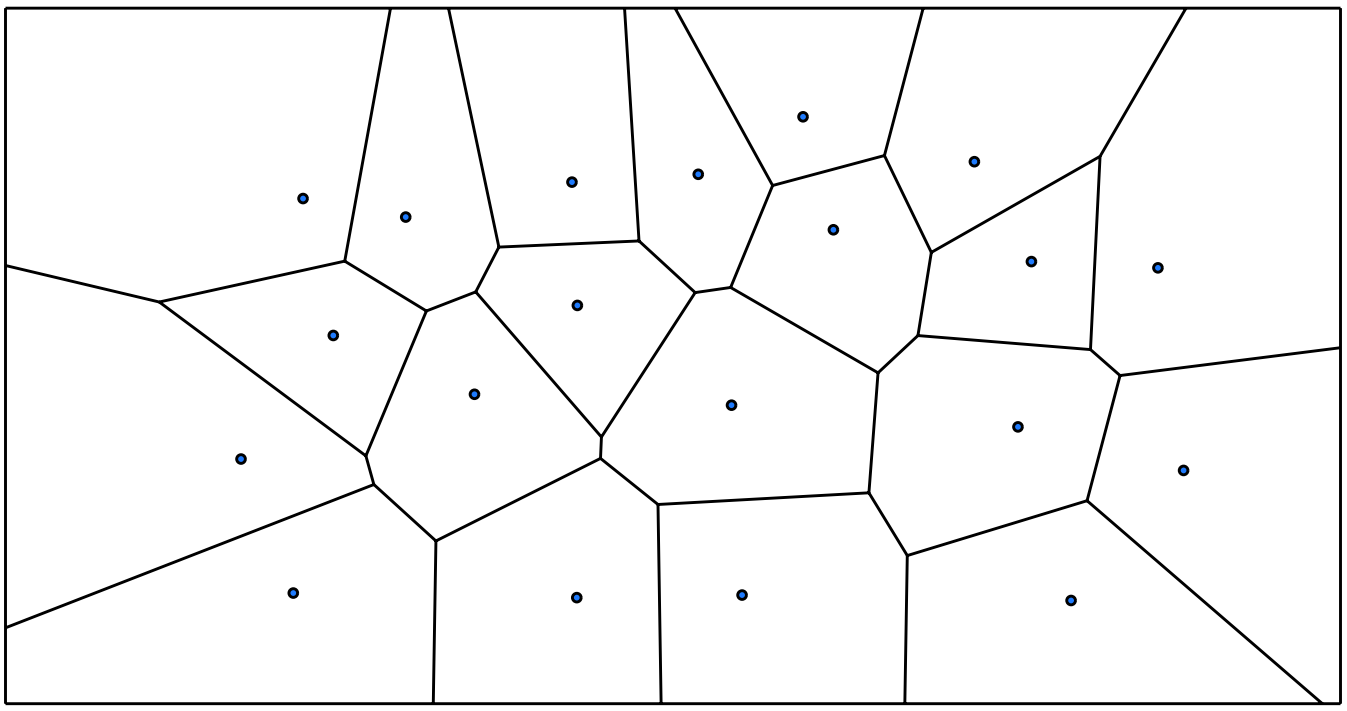
A naive implementation of IVFFlat index should be built upon a large amount of existing data. The user should populate the vector table and then request building an IVFFlat index. During the process, the algorithm finds `lists` number of centroids using the K-means algorithm. Then, vectors are stored within the bucket corresponding to the nearest centroid.

Assume that we have vectors in a 2-dimension space as below:

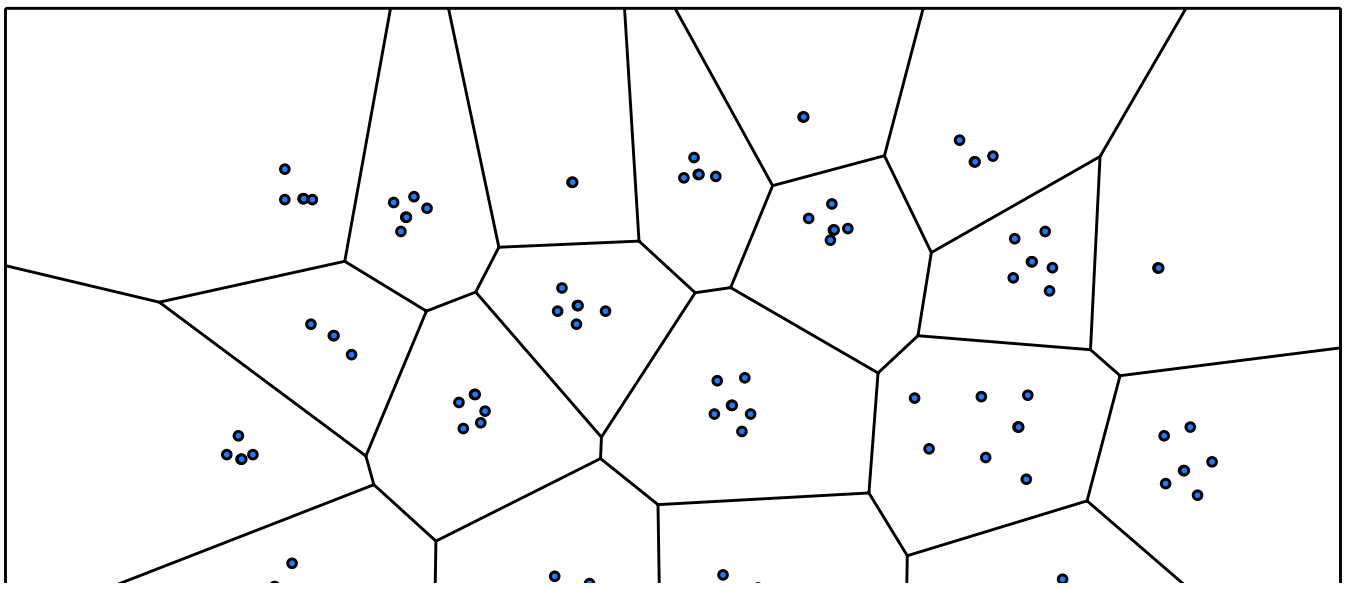


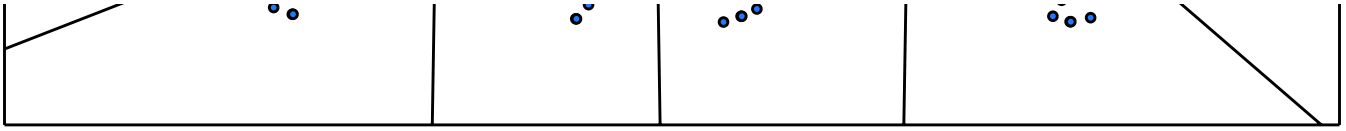


When the user requests to build an index on existing data, the algorithm first finds the centroids using the K-means algorithm. Each centroid corresponds to a bucket, which will store a cluster of vectors. Each vertex on edge in the below Voronoi diagram has the same euclidean distance to the two nearest centroids, which implies the edge of the split buckets.



Now that we have the centroids, the algorithm can go through all vectors and put them into the corresponding bucket based on the nearest centroid to the vector.





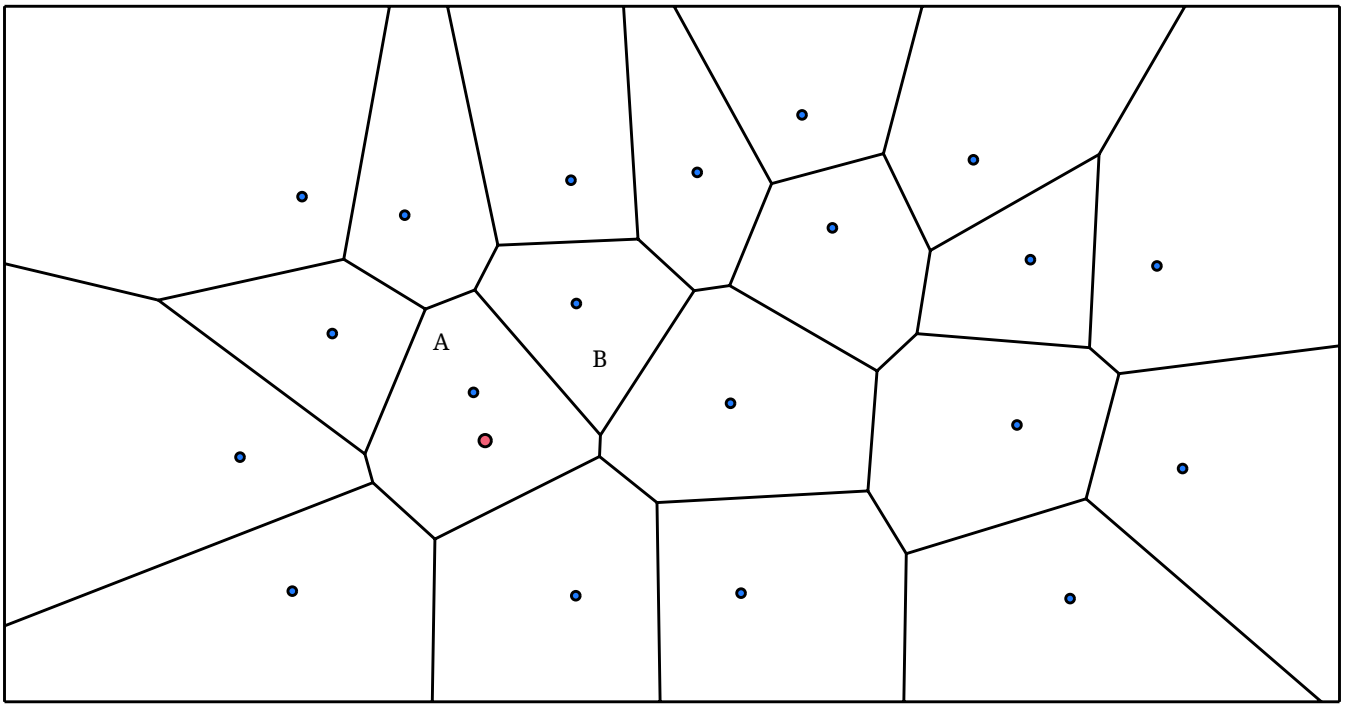
Credit: the graph is generated with https://websvg.github.io/voronoi/ and edited with OmniGraffle.

Going through the process, you will find that, in a naive implementation of IVFFlat index,

- It can only be built upon existing data. You cannot start an IVFFlat index from empty set.
- The centroids are decided at the index build time, and if the data inserted later have different distribution from the initial data, the index will become inefficient.

Insertion

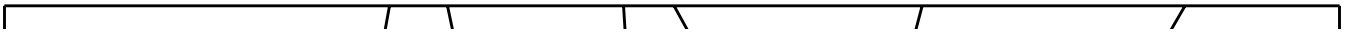
The insertion process simply finds the nearest centroid to the vector and puts the vector into that bucket.

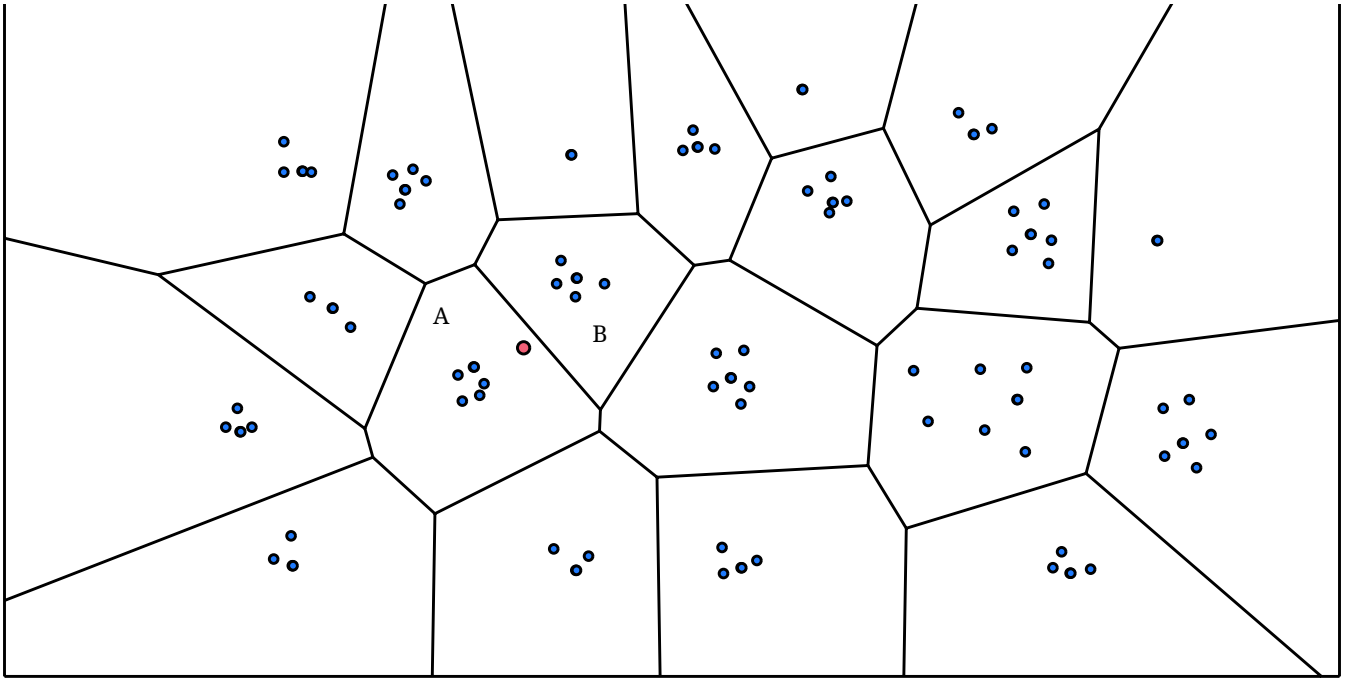


In the above example, the red vertex will be added to the A bucket.

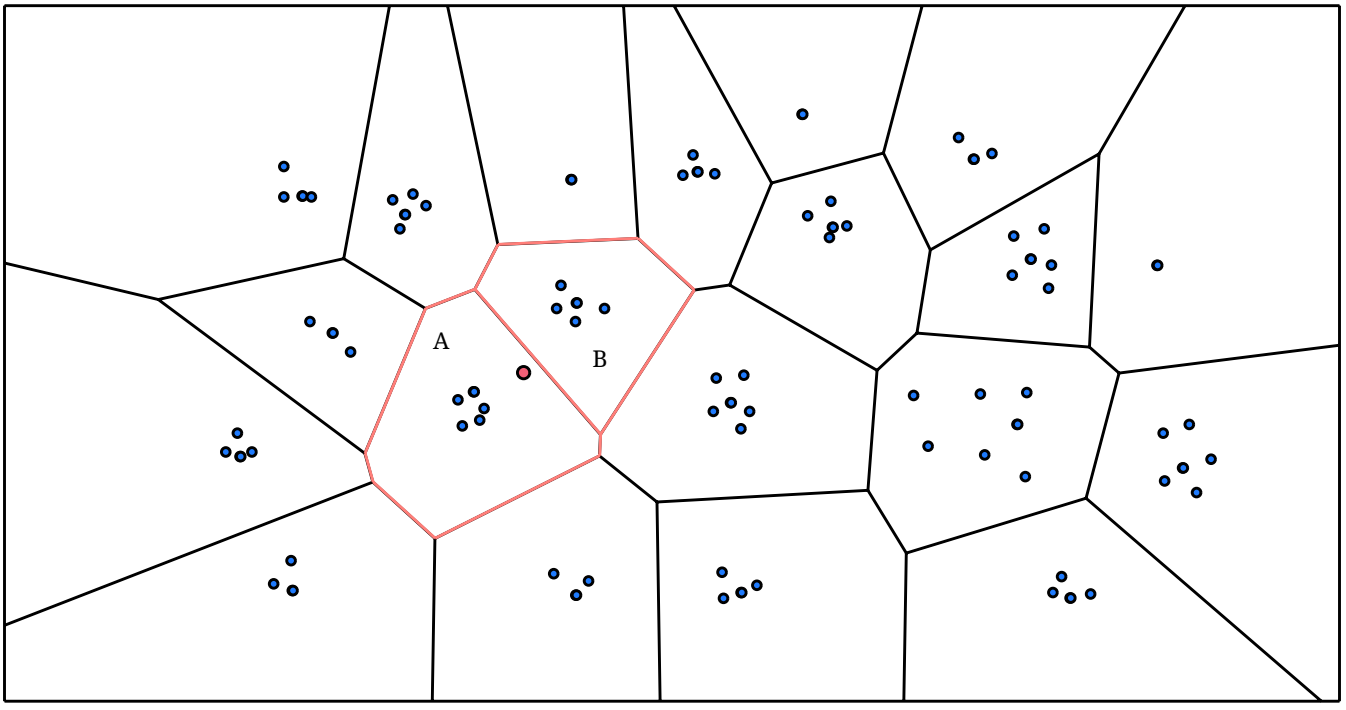
Lookup

The lookup process will find the nearest `probe_lists` centroids and iterate all vectors in these buckets to find the approximate nearest k neighbors. To explain why this is necessary, let us take a look at the below example.





The red vertex is the base vector that the user wants to find, for example, its 5 nearest neighbors. If we only search the A bucket (which the red vertex should be in), we will find all 5 vertices in the A bucket. However, there might be some vertices in a nearby B bucket that has shorter distance to the base probe vector. Therefore, to increase the accuracy of the search, it would be better to probe more than one bucket for a base vector, in order to get more accurate nearest neighbors.



In the index lookup implementation, you will need to probe `probe_lists` number of centroid buckets, retrieving k nearest neighbors from each of them (we call them local result), and do a top-n sort to get the k nearest neighbors (global result) from the local result.

Implementation

You may implement the IVFFlat index in `ivfflat_index.cpp`. Some notes:

- You may implement the K-means algorithm to run for a fixed amount of iterators (i.e., 500) instead of waiting for converging.
- You may need to store the indexed vectors along with RIDs of the data because the database system will need the RIDs to retrieve the corresponding row.
- Remember that you can use `ComputeDistance` from `vector_expressions.h` to compute the distance between two points.

And the pseudo code for a naive K-means implementation:

```
def FindCentroid(vert, centroids):
    return centroids.min(|centroid| distance(centroid, vert))

def FindCentroids(data, current_centroids):
    buckets = {}
    for vert in data:
        centroid = FindCentroid(vert, current_centroids)
        buckets[centroid].push(vert)
    new_centroids = []
    for bucket in buckets:
        new_centroids = avg(buckets[centroid])
    return new_centroids

centroids = random_sample(initial_data, num_lists)
for iter in 0..500:
    centroids = FindCentroids(initial_data, centroids)
return centroids
```

Testing

At this point, you can run the test cases using SQLLogicTest.

```
make -j8 sqllogictest
./bin/bustub-sqllogictest ../test/sql/vector.04-ivfflat.slt --verbose
```

The test cases do not do any correctness checks and you will need to compare with the below output by yourself. Your result could be different from the reference solution because of random stuff (i.e., random seed is different). You will need to ensure all nearest neighbor queries have been converted to a vector index scan.

▸ Reference Test Result

Bonus Tasks

Implement the Elkan's Accelerated K-means algorithm

pgvector uses Elkan's Accelerated K-means algorithm to make the index building process faster.

Persist Data to Disk

You may implement the buffer pool manager and think of ways to persist the index data to the disk.

Rebuilding the Index

If the later inserted data have a different distribution compared with the data during index building, you might need to rebuild the index in order to make the approximate nearest neighbor result accurate.

Deletion and Updates

The current implementation (and vector index interfaces) only supports insertions. You may add new interfaces to the vector index and implement updates and deletions.

Your feedback is greatly appreciated. Welcome to join our Discord Community.

NSW (Navigable Small Worlds) Index

Before building the so-called HNSW (Hierarchical Navigable Small Worlds) index, we will start with the basic component of the HNSW index -- NSW (Navigable Small Worlds). In this chapter, we will build a graph-based index structure for vectors.

The list of files that you will likely need to modify:

```
src/include/storage/index/hnsw_index.h
src/storage/index/hnsw_index.cpp
```

Related Readings

- Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs
- Hierarchical Navigable Small Worlds (HNSW) from Pinecone's Faiss Manual

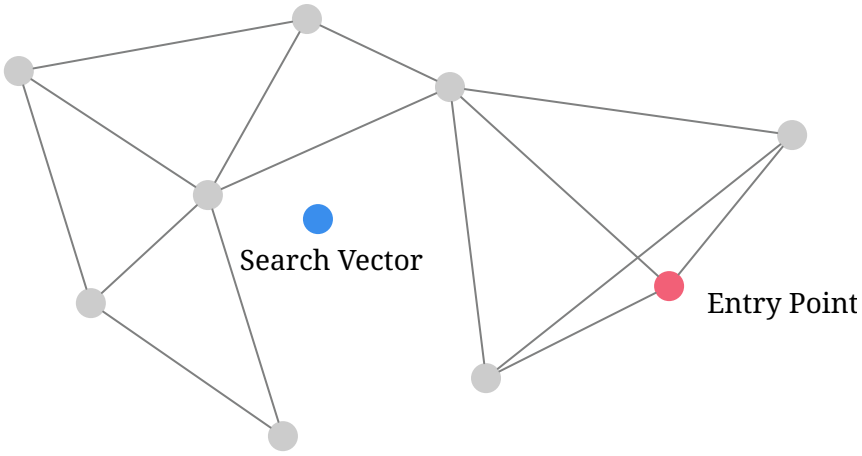
Overview

In NSW, you start from one or more entry points, and greedily visit the neighbors to find a point closer to the search vector. At some point, when there are no nearer point to explore, the search can be stopped and the approximate nearest neighbors are returned.

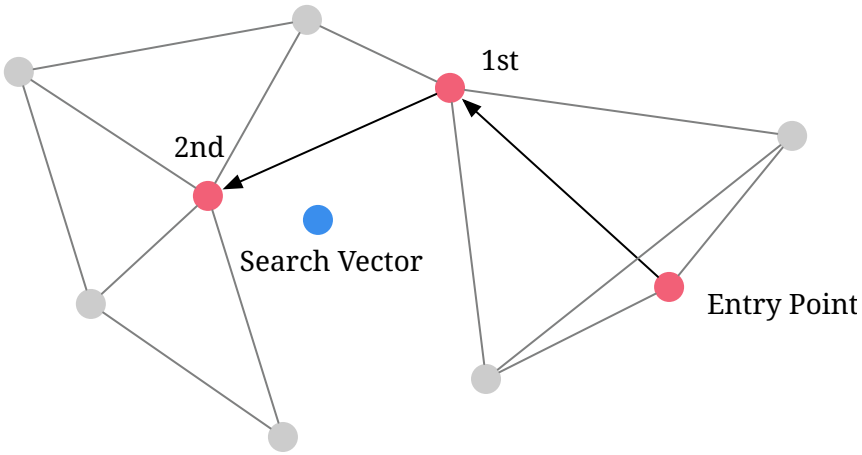
Layer Search

One Entry Point, 1-Nearest Neighbor

Let us start with searching exactly one nearest neighbor within an NSW graph of one entry point. Assume we have already built an NSW index over the below vectors in a 2-dimension space. The layer has one entry point as indicated in the figure in red color.



An intuitive way to find the nearest neighbor is to find a closer point to the search vector in all the neighbors as below.

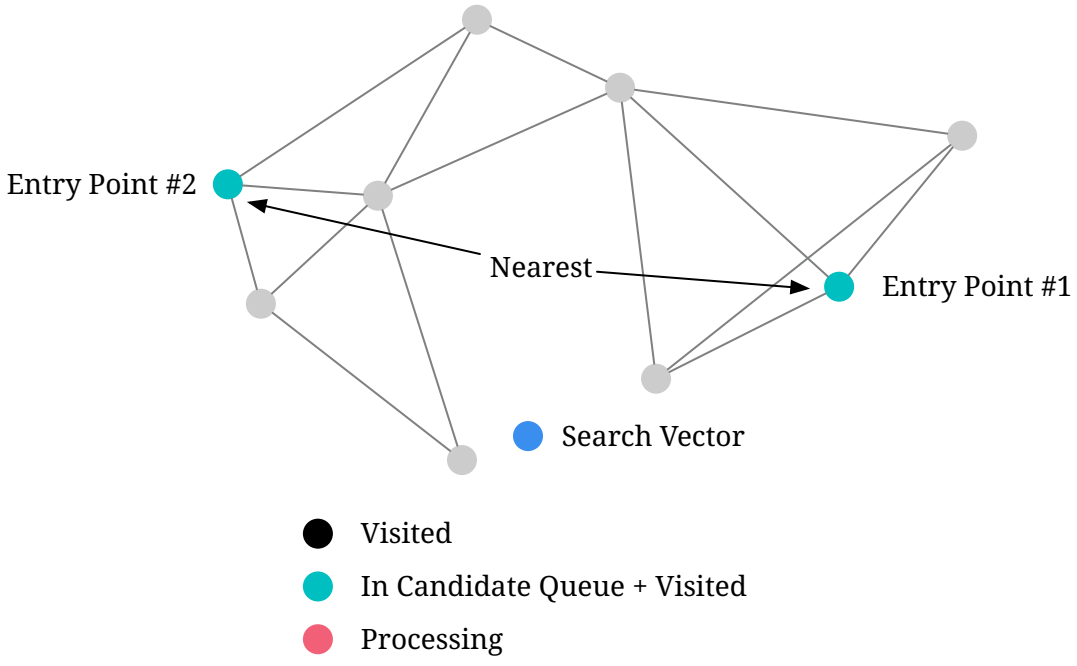


By greedily move to a vector closer to the search target, we will likely find the nearest neighbor.

Multiple Entry Points, k-Nearest Neighbors

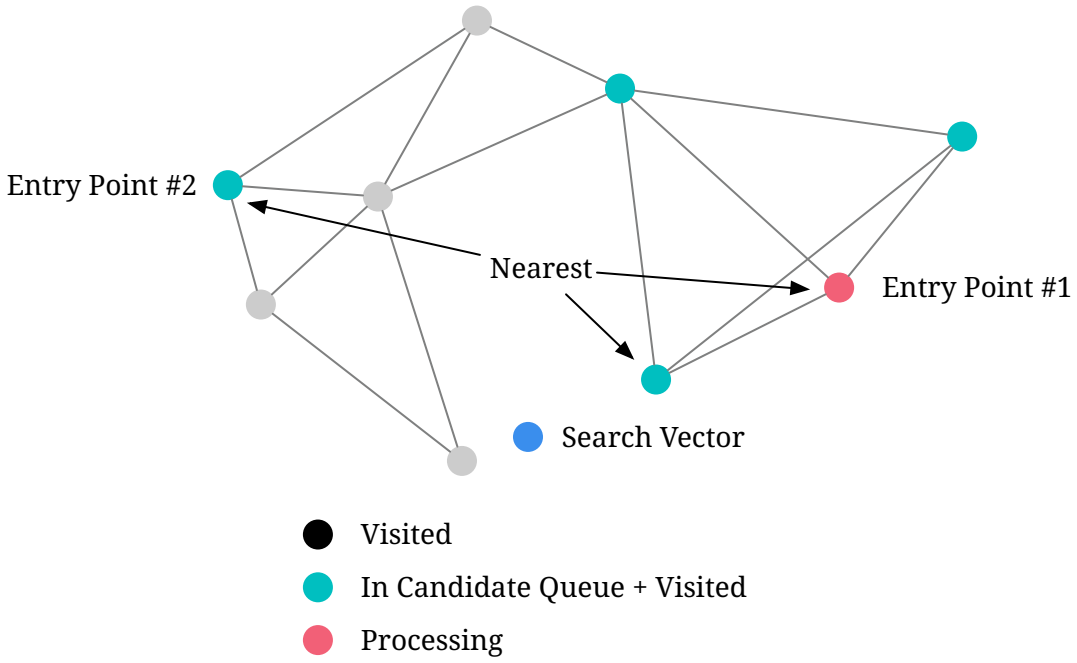
Now let us move forward and find 3 nearest neighbors with 2 entry points in the following NSW index. The full NSW algorithm maintains 2 data structures: an explore candidate priority queue and a result binary heap. The explore candidate priority queue maintains the candidate vectors to visit. The top element in the queue is the current to-be-explored nearest node. The result set is a binary heap as in the top-k executor. It is a max-heap and maintains the k-nearest neighbors across all vectors that we have visited.

Before exploring, we add the entry points to both the explore candidate priority queue `C` and the result set `W`.

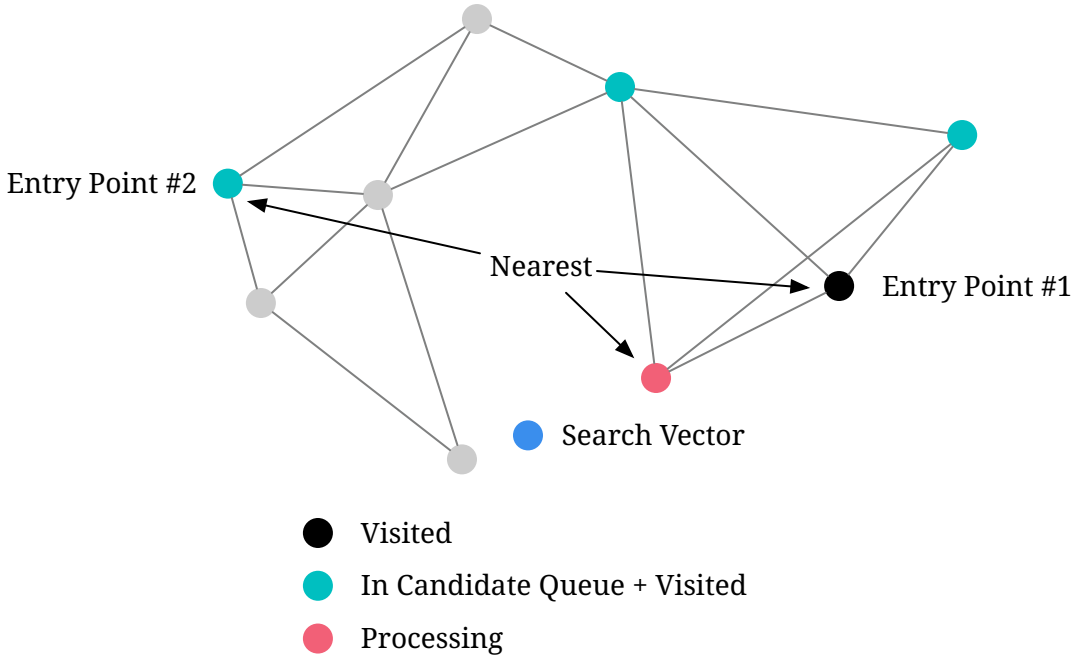


Then, we pop the nearest vector (which is entry point #1) to explore from `C` queue, and add all its neighbors to the `C` queue and the `W` set. The `W` set only maintains the k-

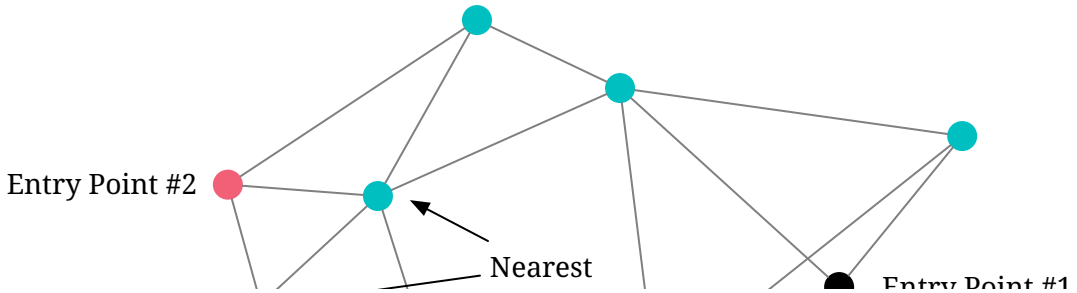
nearest neighbors and therefore it will not have more then 3 elements.

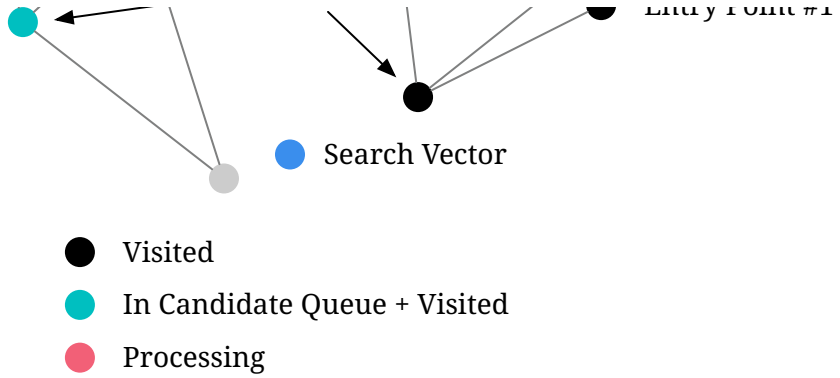


In the next iteration, we pop the nearest vector (which is the red point as below) from the `C` queue. All its neighbors have been visited, so we do not add more elements to the candidate explore queue.

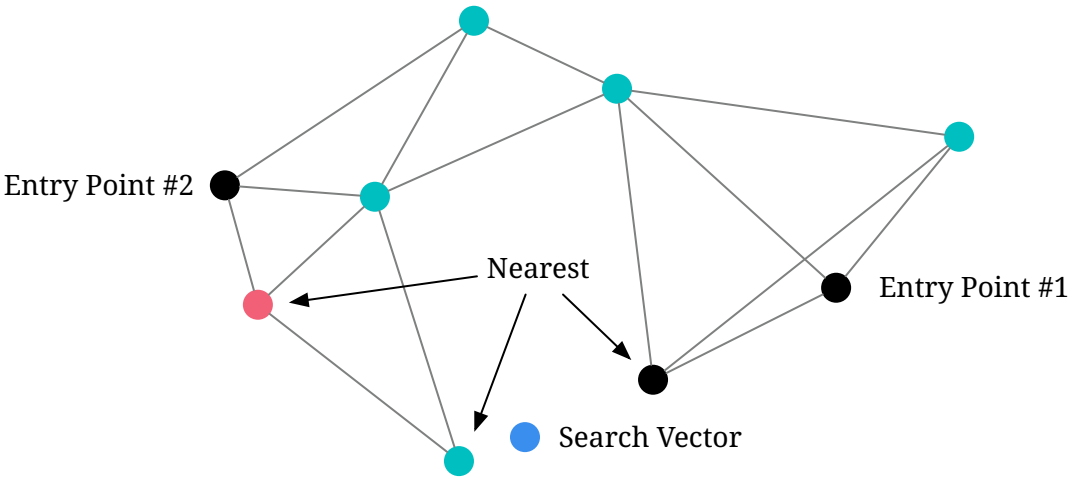


Then, we pop the next vector from the `C` queue, which is entry point #2. We add all its neighbors to `C` and update `W`.

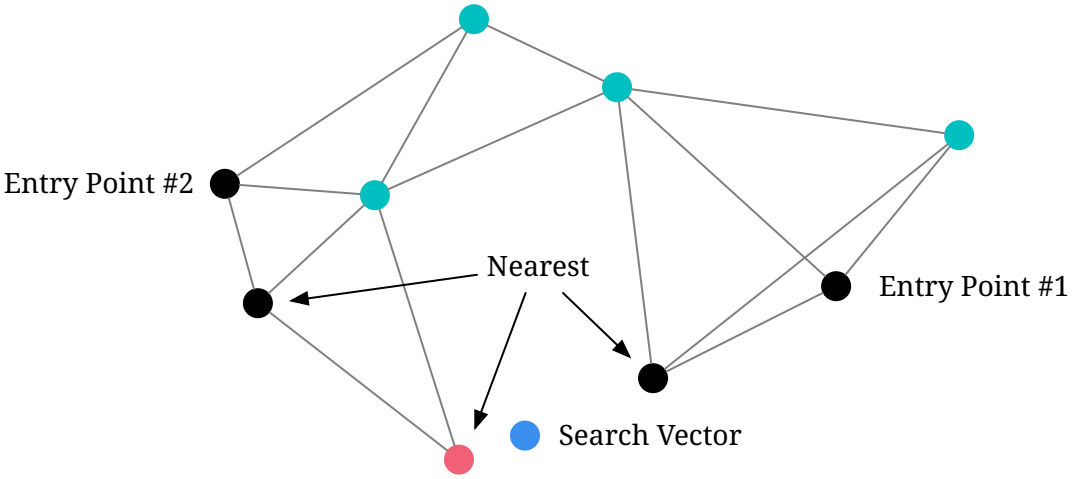




After that, we continue to explore the vector as indicated below.

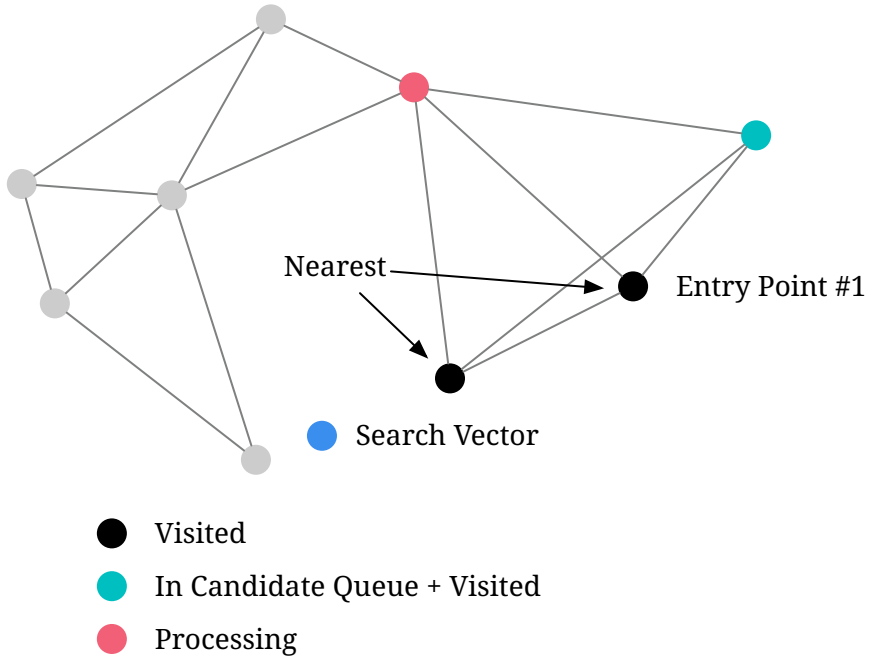


When we reach the vector as below, all vectors in the candidate explore queue have larger distance to the search vector than the vectors in the result set. Therefore, we can stop searching at this point.



Why we need multiple entry points

Still consider the above example, if we want to find the 2-nearest neighbors and only have 1 entry point, the search algorithm will stop at the below status.



At this point, all candidates in the `C` queue have larger distance than all vectors in the result set. It reaches the search end condition and will not be able to explore the actual nearest neighbors.

Pseudo Code

```
C <- entry_points as min heap on distance
W <- entry_points as max heap on distance
visited <- entry_points as unordered set
while not C.empty():
    node <- pop C (nearest element in C)
    if dist(node) > dist(top W): # top W is the furthest element in W
        break
    for neighbor in neighbors of node:
        if not visited neighbor:
            visited += neighbor
            C += neighbor
            W += neighbor
            retain k-nearest elements in W
return W
```

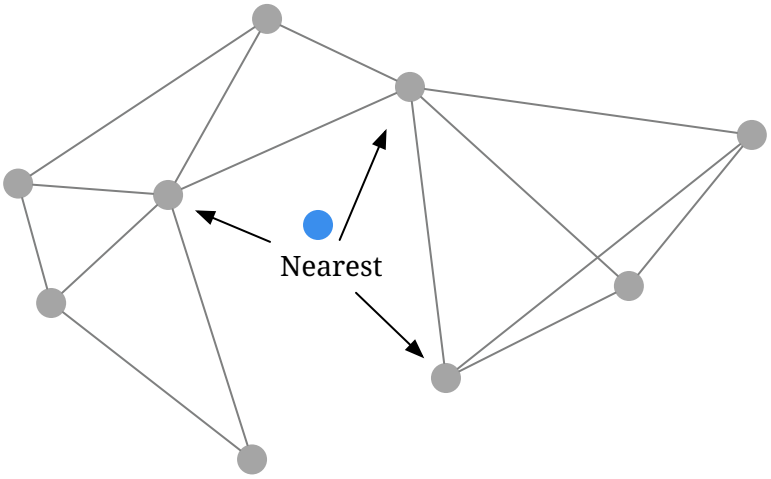
The pseudo code is the same as in the HNSW paper. The input parameters for this algorithm are:

- limit: number of neighbors to search. Keep at most this number of elements in the result set.
- base_vector: the search target.
- entry_points: the entry points.

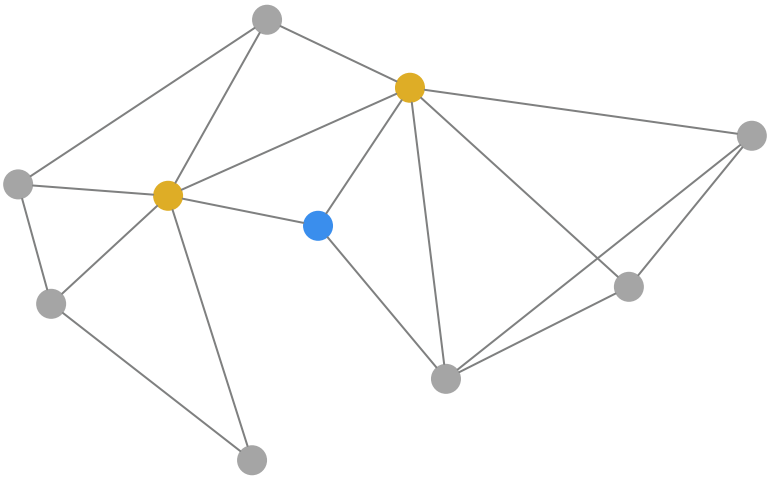
You may implement the search layer functionality in `NSW::SearchLayer`. By default, the starter code uses the first vertex inserted into a layer as the entry point. You may also implement the algorithm to randomly sample multiple entry points.

Insertion

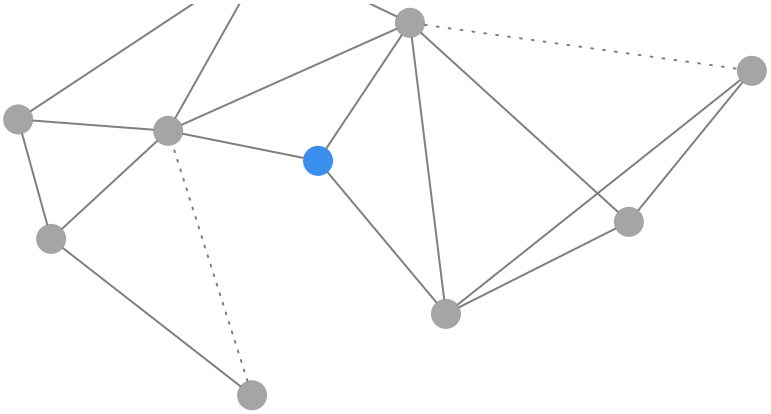
Insertion follows the same process as layer search. We simply find the k-nearest neighbors of the to-be-inserted vector in a layer, and add edges from the inserting vector to the neighbors. For example, we want to establish 3 connections, where the number 3 is the `m` parameter of the index, and therefore we find 3 nearest neighbors to the blue vector as below.



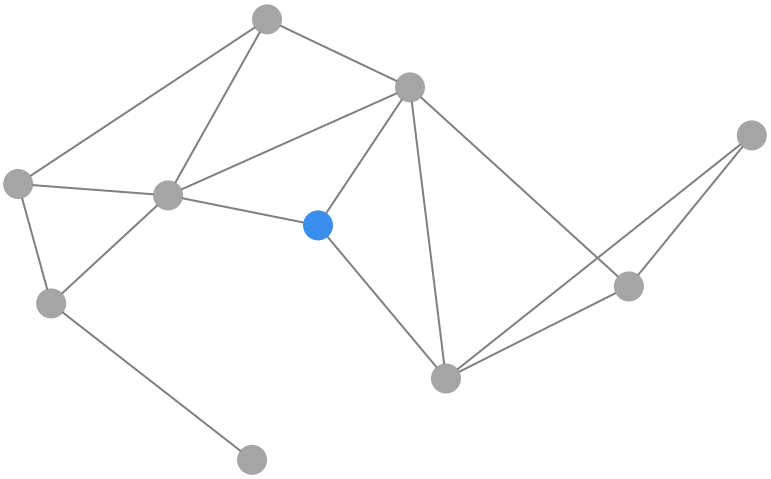
Now we add edges between the vector and the neighbors. However, the two yellow points now have too many connections, controlled by the `max_m` parameter. This may make the search layer process slower. Therefore, we need to cut down number of connections after adding more edges.



To cut down the conections, we simply re-compute the k-nearest neighbors of the two yellow nodes, where `k = max_m`.



And we only keep the k-nearest neighbor connections after inserting the new vector.



Implementation

You may implement the NSW index in `hnsw_index.cpp` at this point. The starter code implements the HNSW index class as a one-layer index, which is equivalent to the NSW index. You will need to use the following parameters in your implementation:

Testing

At this point, you can run the test cases using SQLLogicTest.

```
make -j8 sqllogictest
./bin/bustub-sqllogictest ../test/sql/vector.05-hnsw.slt --verbose
```

The test cases do not do any correctness checks and you will need to compare with the below output by yourself. Your result could be different from the reference solution because of random stuff (i.e., random seed is different). You will need to ensure all nearest neighbor queries have been converted to a vector index scan.

▸ Reference Test Result

Bonus Tasks

Implement Better Neighbor-Selection Algorithm

The paper Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs describes two ways of selection a neighbor: simply based on distance (as in this chapter), or a better way with heuristics.

Persist Data to Disk

You may implement the buffer pool manager and think of ways to persist the index data to the disk.

Deletion and Updates

The current implementation (and vector index interfaces) only supports insertions. You may add new interfaces to the vector index and implement updates and deletions.

Your feedback is greatly appreciated. Welcome to join our Discord Community.

HNSW (Hierarchical Navigable Small Worlds) Index

Now that we built NSW indexes in the previous chapter, we can now have multiple layers of NSW indexes and add hierarchy to the index structure to make it more efficient.

The starter code for this part is ready but the write-up is still working-in-progress.

The list of files that you will likely need to modify:

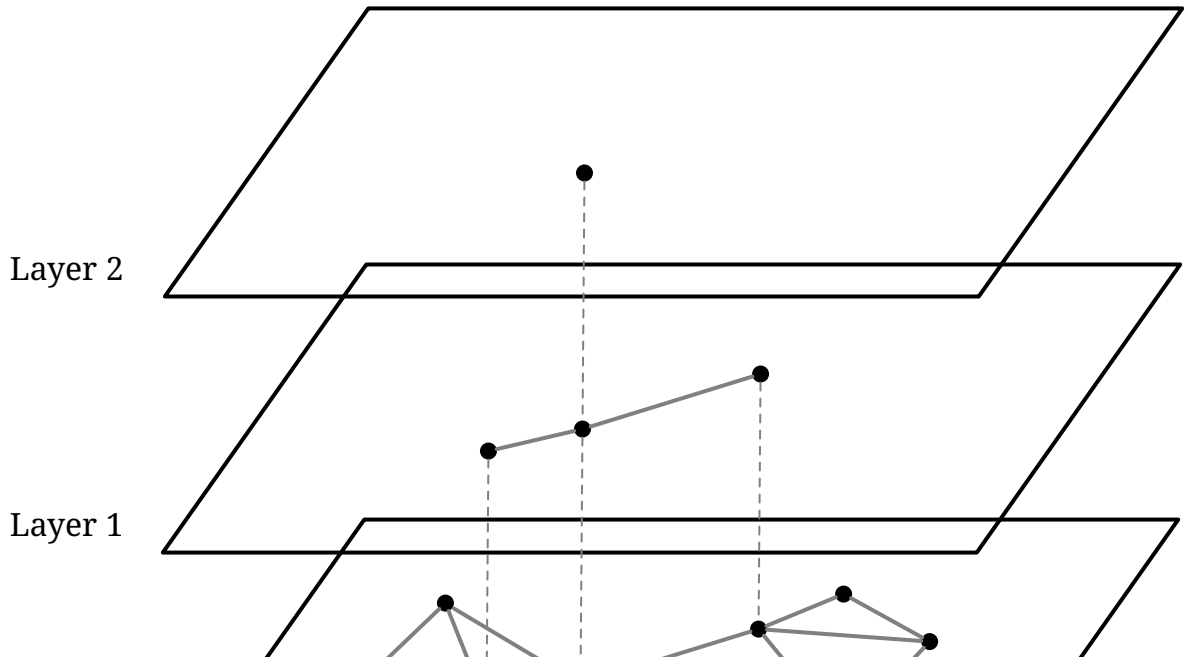
```
src/include/storage/index/hnsw_index.h
src/storage/index/hnsw_index.cpp
```

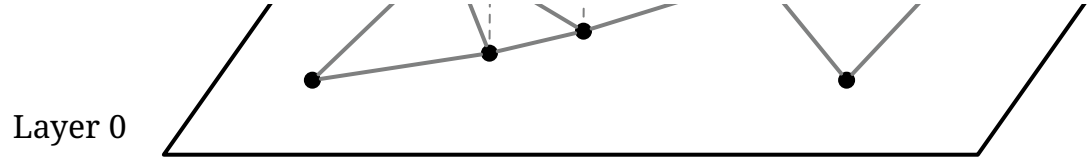
Related Readings

- Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs
- Hierarchical Navigable Small Worlds (HNSW) from Pinecone's Faiss Manual

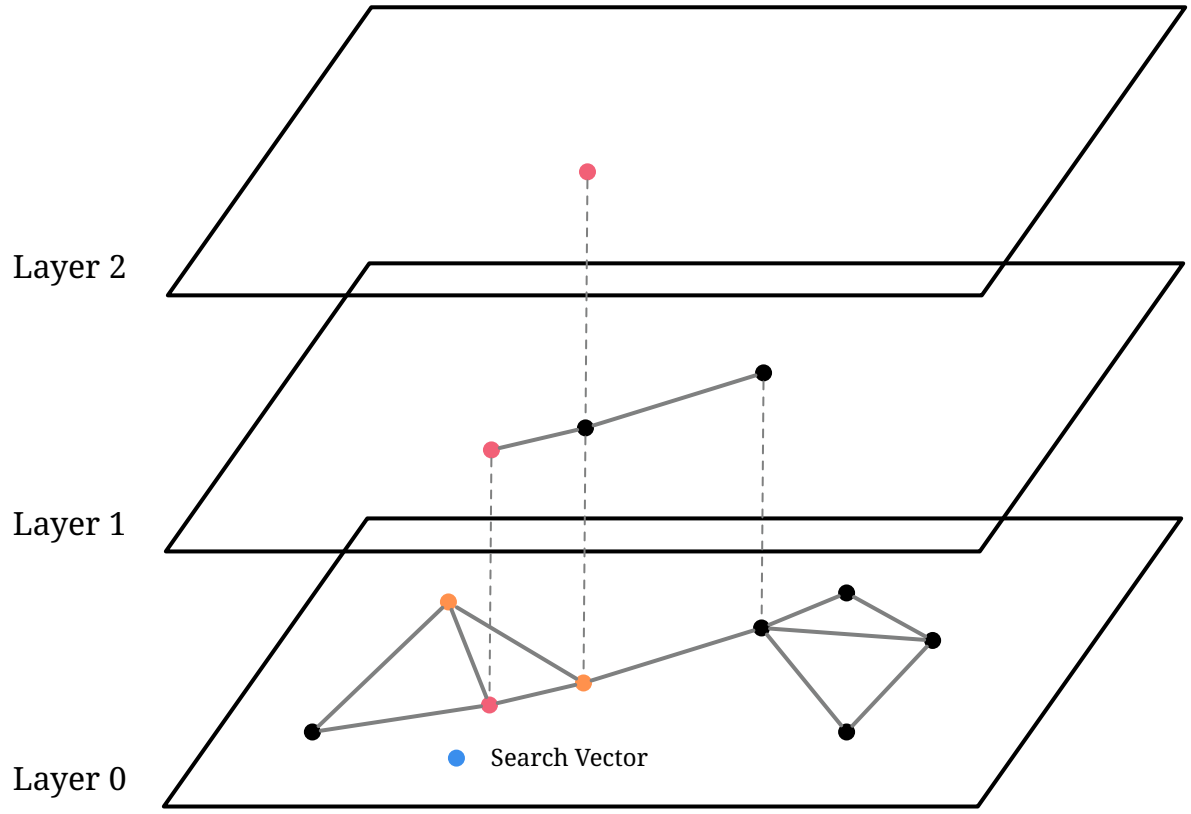
Overview

HNSW adds layered structure to the NSW index, therefore making the search process faster. The idea is similar to the skiplist data structure or mipmaps in computer graphics. The bottom-most level 0 NSW layer contains all information, and we randomly put some vectors to the upper layer (more upper layer has fewer elements), which are also NSW indexes. The search process starts from the upper-most layer, and uses neighbors in that layer as the entry points of the lower layer.





Index Lookup



The algorithm starts from the upper-most layer. As above, it first searches the nearest neighbor to the search target in layer 2, then use that neighbor as the entry point to search the nearest neighbor in layer 1, and lastly, use the entry point to find the k-nearest (3-nearest) neighbors in layer 0.

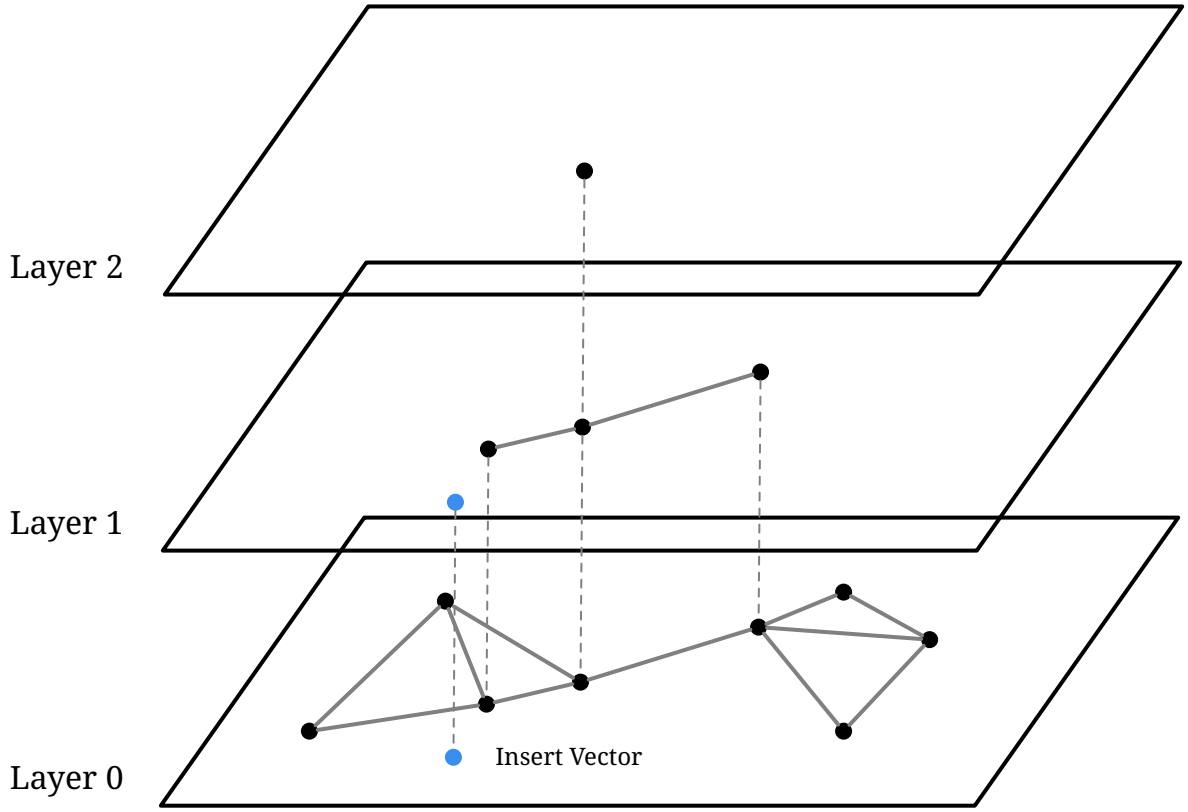
Pseudo Code

```
ep = upper-most level entry point
for go down one level until last level:
    ep <- layers[level].search(ep=ep, limit=1, search_target)
return layers[0].search(ep=ep, limit=limit, search_target)
```

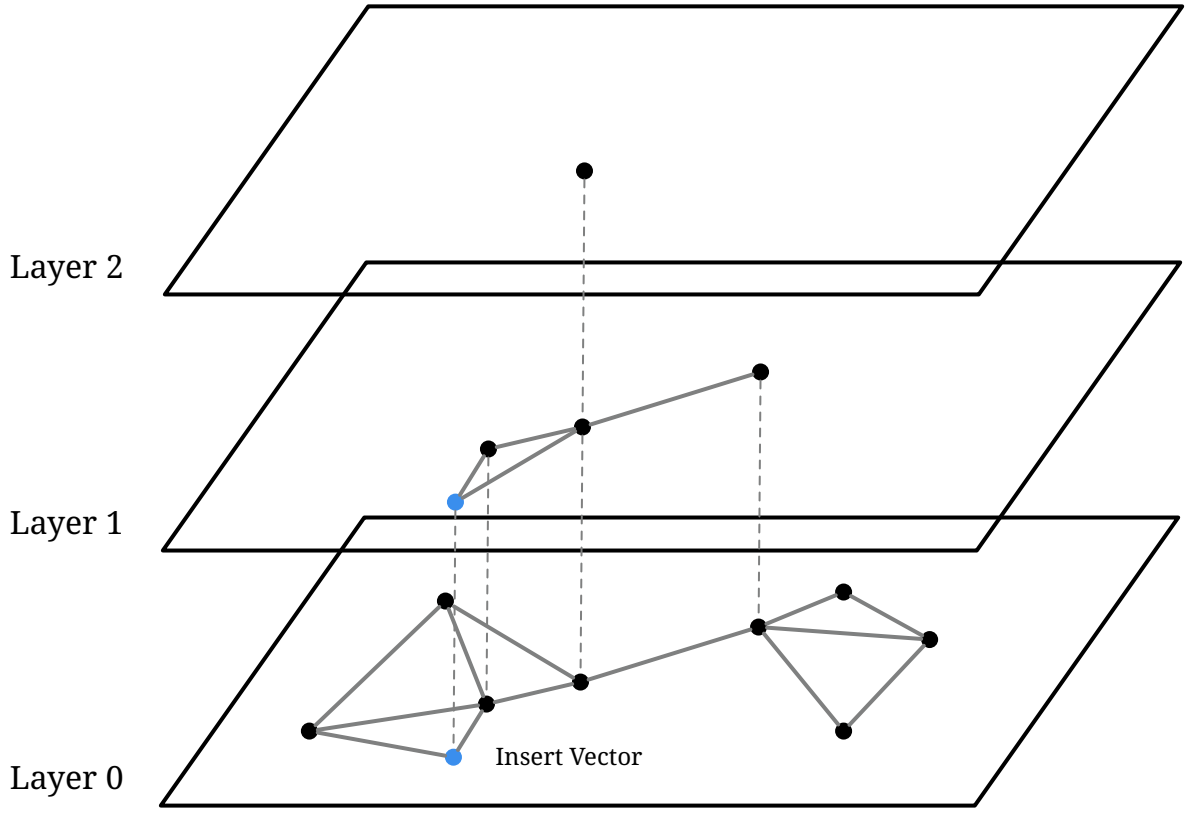
Insertion

Before inserting, we will need to decide which layer and below levels to insert the vector. From the HNSW paper, this is computed by $\text{level} = \lfloor -\ln(\text{unif}(0 \ldots 1)) \times m_L \rfloor$.

For example, we decide to add the vector to level 1 and below.



The algorihm searches the entry point to level 1 by searching the nearest neighbor starting from the upper-most layer. When it reaches level 1 and below, it searches `ef_construction` neighbors as the entry point, and selects the m-nearest neighbors from the `ef_construction` nodes to establish edges.



Pseudo Code

```
ep = upper-most level entry point
target_level = generate random level based on m_L
for go down one level until target_level + 1
    ep <- layers[level].search(ep=ep, limit=1, search_target)
for go down one level until level 0
    ep <- layers[level].search(ep=ep, limit=ef_construction, search_target)
    neighbors <- m-nearest neighbor in ep
    connect neighbors with search_target
    purge edges of neighbors if larger than m_max of that layer
```

You may also refer to the HNSW paper for a more detailed pseudo code.

Implementation

You have already implemented an NSW index in the previous chapter. Therefore, you may change the HNSW index implementation to have multiple layers of NSW. In your implementation, you do not need to reuse your NSW's insertion implementation as we will need to reuse the entry points from the previous layer in the HNSW insertion process.

Testing

At this point, you can run the test cases using SQLLogicTest.

```
make -j8 sqllogictest
./bin/bustub-sqllogictest ../test/sql/vector.05-hnsw.slt --verbose
```

The test cases do not do any correctness checks and you will need to compare with the below output by yourself. Your result could be different from the reference solution because of random stuff (i.e., random seed is different). You will need to ensure all nearest neighbor queries have been converted to a vector index scan.

▸ Reference Test Result

Epilogue

At this point, you should have learned how to add vector capabilities to an existing relational database system and should have a solid understanding of how vector indexes work. However, there are still some tasks that need to be completed before your vector extension is ready for production. These include persisting the vector indexes to disk, supporting deletion and updates, preventing out-of-memory issues, ensuring efficient index lookups with the disk format, and making the index compatible with the underlying database system, including transactions and multi-version indexes.

We Need Your Feedback

To help us continously improve the course and enhance your (or other students') learning experience, we sincerely hope that you can tell us something about your learning path. Your feedback is important for us!

Vector Databases or Databases with Vector Extension?

Adding vector capabilities to a database system is relatively easy. In Professor Andy Pavlo's Databases in 2023: A Year in Review in Ottertune blog, he mentioned that the engineering effort required to introduce a new access method and index data structure for vector search is *not that much*. This tutorial should have demonstrated that extending a database system with vector capabilities can be accomplished without fundamentally altering the system, and by adding additional components, it can easily support vector similarity searches.

> There are two likely explanations for the quick proliferation of vector search indexes. The first is that similarity search via embeddings is such a compelling use case that every DBMS vendor rushed out their version and announced it immediately. The second is that the engineering effort to introduce what amounts to just a new access method and index data structure is small enough that it did not take that much work for the DBMS vendors to add vector search. Most vendors did not write their vector index from scratch and instead just integrated one of the several high-quality open-source libraries available (e.g., Microsoft DiskANN, Meta Faiss).
> *-- Andy Pavlo, Databases in 2023: A Year in Review on Ottertune Blog*

Acknowledgments

Personally speaking, vector database is still a new concept to me. I still remember my first involvement with vector databases while interning at Neon in the summer of 2023. One day, Nikita added me to a Slack channel called `#vector`, where I discovered that Konstantin was working on a new vector extension for PostgreSQL called pgembedding incorporating HNSW support. Eventually, this extension got discontinued after pgvector added HNSW support later in 2023. Nevertheless, it was my initial exposure to vector searches and vector databases, and the challenges that developers face when dealing with vector indexes in relational databases are exciting areas for long-term exploration.

At the end of the tutorial, I would like to thank the 15-445 course staff for landing my giant pull request https://github.com/cmu-db/bustub/pull/682 that adds vector type to the upstream BusTub repo. Thank you Yuchen, Avery, and Ruijie for reviewing my stuff and supporting my work!

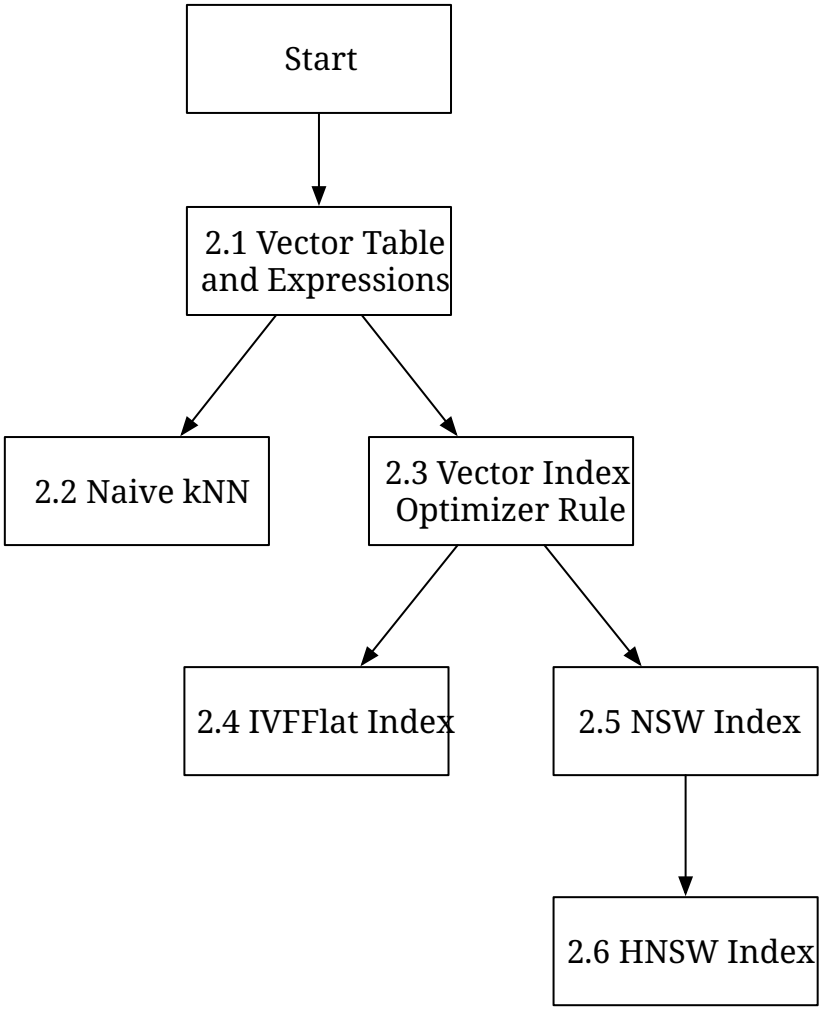
Your feedback is greatly appreciated. Welcome to join our Discord Community.

The C++ Way (over BusTub)

In this section, we will implement a vector database extension over the BusTub educational system.

Overview

You may approach this tutorial by implementing the things that interest you most first. The following figure is a learning path diagram that visualizes the dependencies between the chapters.



Environment Setup

You will be working on a modified codebase based on Fall 2023's version of BusTub.

```
git clone https://github.com/skyzh/bustub-vectordb
```

At minimum, you will need `cmake` to configure the build system, and `llvm@14` or Apple Developer Toolchain to compile the project. The codebase also uses clang-format and clang-tidy in `llvm@14` for style checks. To compile the system,

```
mkdir build && cd build
cmake ..
make -j8 shell sqllogictest
```

Then, you can run `./bin/bustub-shell` to start the BusTub SQL shell.

```
./bin/bustub-shell

bustub> select array [1.0, 2.0, 3.0];
+-------------+
| __unnamed#0 |
+-------------+
| [1,2,3]     |
+-------------+
```

In BusTub, you can use the `array` keyword to create a vector. The elements in a vector must be of decimal (double) type.

Extra Content

What did we change from the CMU-DB's BusTub codebase

The `bustub-vectordb` repository implements some stub code for you so that you can focus on the implementation of the vector things.

Buffer Pool Manager. We have a modified version of the table heap and a mock buffer pool manager. All the data stay in memory. If you are interested in persisting everything to disk, you may revert the buffer pool manager patch commit (remember to revert both the buffer pool manager and the table heap), and start from the 15-445/645 project 1 buffer pool manager.

Vector Expressions. The modified BusTub codebase has support for vector distance expressions.

Vector Indexes. The codebase adds support for vector indexes besides B+ tree and hash table indexes.

Vector Executors. With the vector index conversion optimizer rule and the vector index scan executor, users will be able to scan the vector index when running some specific k-nearest neighbor SQLs.

Your feedback is greatly appreciated. Welcome to join our Discord Community.

Working on Large Datasets

The Rust Way (over RisingLight)

We will likely release this chapter by the end of 2024.