Code Reorganization

Magdalen Dobson

November 2022

1 Introduction

Some applications of ANNS require stronger guarantees than DiskANN currently provided. Chiefly among these is linearization of insertions, deletions, and queries, as well as efficient snapshotting. In order to support running the DiskANN algorithm using data structures that provide these guarantees, we need to abstract the core functioning of the algorithm—that is, the building, searching, and update logic—from the core functions of the object that stores the graph—primarily reading and writing to the graph object, but also functions such as saving and loading.

In this document I propose a framework for abstracting the DiskANN code to use multiple graph types. DiskANN will now be organized using a `Graph` object, where data will be stored and manipulated, and an `Index` object, where building and updating takes place.

One example of an alternative framework is the Aspen graph data structure. An initial implementation of diskann using aspen can be found at `https://github.com/ParAlg/CPAM/tree/versionedDiskANN/examples/diskann`. At a high level, Aspen will provide linearizability by directing nearest neighbor queries to a past *version* of the graph, while a batch of insertions and deletions is implemented without locks on a new version. When the batch operation ends, a new version is released, and future queries are answered using this version. The Aspen graph is also a purely functional data structure, so it is fully persistent and can be easily snapshotted.

Our general approach is that the `Graph` will store data, adjacency information, and auxiliary structures needed to support reading and writing. It may also need to store information about the state of the index to ensure that operations are performed legally.

There are some outstanding questions about how such a framework should look to a developer and to a user. For example, when using Aspen, should the user have to manually call a batch of insertions or deletions to trigger, or should this be handled entirely by the backend?

In the attached pseudocode, the object `Graph` contains the following methods; all are virtual and will be overloaded by classes inheriting `Graph`.

1. Saving and loading: methods `save_graph()` and `load_graph()`.

2. Adding new points to the index. Methods `insert()` and `delete()` take in a vector of new tags as well as their corresponding data points and make them live in `Graph`.

3. Reading:

   (a) `read()` return an adjacency list for some vertex ID.

   (b) `read_static()` when the graph is in a state where adjacency lists are not being modified, there is no need to take locks while reading. In some `Graph`s, this function will redirect to `read()`.

   (c) `read_coords()` reads data associated with a tag.

   (d) `size()` returns the number of out-neighbors associated with a tag.

4. Writing:

   (a) `write()` takes in a vector of tags and a new adjacency list for each tag and updates the adjacency list information accordingly.

   (b) `write_inplace()` is used for building a static graph; it is mostly for use with Aspen, where it makes in-place updates instead of functional ones. Intended for settings where thread safety is not an issue.

5. Safety and other management:

   (a) Safety calling static functions

      i. `Graph` object will contain a boolean flag for whether the index is dynamic or not; changed using `declare_index_dynamic()`.

   (b) Safety during insertions

      i. For Aspen, the user will supply insertions in batches which are processed in one call. The `insert_tick()` function ensures that no new writes/versions are released during this time.

   (c) Safety during deletions

      i. The `delete_tick()` function works the same as the insert version.

      ii. Since Aspen processes a delete set using multiple calls to `consolidate_deletes()`, we provide a boolean flag which indicates whether a delete epoch is currently running, and which stops another one from starting or tags being deleted from the graph. This is shown in `initiate_delete_epoch()`.