Streaming ANNS with guarantees

Suhas Jayaram Subramanya
(suhasj@cs.cmu.edu)

November 2022

1 Basics

There exists an in-memory graph with each adjacency list represented by an 8-byte pointer. Unless stated otherwise, pointers (in this document) always point to a valid memory address with capacity to store a maximum of R edges for any vertex. In DRAM, our graph can be represented as an array of pointers (e.g. `G = new uint64_t[N_MAX];` for N_{max} points in the index).

Claim: When multiple operations (insert/delete) are modifying the graph G in parallel, we don't need to ensure that each operation gets access to an immutable *globally* consistent adjacency list for every vertex. Let G_t be the state of the graph at time t. If an insert is triggered at time $t_i = t$, Aspen ensures that this insert will see G_t until it finishes execution at time t_f when it commits updates to G_t. If another thread finishes modifying the graph at time t_2 (and $t_i < t_2 < t_f$), then under Aspen, the insert that began at $t_i < t_2$ must not see changes committed to G_t at t_2. For our purpose, I argue that this property is not necessary.

Vamana graphs are approximations to an ideal Monotonic Relative Neighborhood Graph. Since the underlying graph is an *approximate* graph anyway, letting the insert thread see the updated edges at t_2 is mostly beneficial and rarely harmful. For example – say, the insert thread is about to read $G(v)$ for some vertex v and $G(v)$ is updated to $G(v)^+$. If $G(v)^+$ has better edges than $G(v)$, it will improve the quality of visited list used for insertion. For the other case, if $G(v)^+$ is worse than $G(v)$, there exists sufficient redundant paths in the graph for the insert algorithm to get to the right neighborhood (`GreedySearch` might converge a bit slower). Basically, the argument here is that reading $G(v)^+$ vs $G(v)$ does not drastically change the visited set that is used for `RobustPrune`; and even if it did, both visited sets are approximate anyway and the resulting vertex set would have been approximate too.

Proposal: Make operations *locally* consistent – graph update operations get atomic access to adjacency lists. This is a finer level of update and gives us higher concurrency due to the size of the graph. Update operations on adjacency lists are atomic with all reads being atomic, but only successful writes are guaranteed to be atomic. Since $G(v)$ is an 8-byte pointer and x86 guarantees 64-bit reads are atomic, reading $G(v)$ is always atomic. Failed writes don't modify state of the graph and the insertion algorithm will have to handle write failures. We'll touch on why some writes can fail and how to handle the failed writes later.

2 Notation

- G - Graph with N vertices already.
- v - `vec(v)` gives us the d dimensional vector represented by vertex v. $G(v)$ is the adjacency list for vertex v.

3 Log basics

We will assume a WAL-style logging for this system with each log entry described by a 6 tuple: `<TIMESTAMP, TXN-ID, OLD-PTR, NEW-PTR, DEL-SET, ADD-SET>` where:

- `TIMESTAMP` – Wall-clock timestamp generated by the thread logging the entry
- `TXN-ID` – A GUID for each sequence of operations on the graph
- `OLD-PTR` – Raw 8-byte pointer value; exact PTR $\rightarrow$ VERTEX-ID map can also be constructed using the log (like register-renaming in computer architecture). You can also think of raw pointer values as representing *unique* versions of the adjacency lists.
- `NEW-PTR` – New 8-byte pointer value containing changes described in `DEL-SET` and `ADD-SET`
- `DEL-SET` – Exact set of edges to delete for the adjacency list stored in `OLD-PTR`
- `ADD-SET` – Exact set of edges to add to the adjacency list stored in `OLD-PTR`

4 Insert operations

Let us try to insert a new vertex r into G. We'll first run `GreedySearch(G, r)` to get a visited set, then run `RobustPrune` on this to get $G(r)$. This bit does not require any concurrency since we assume atomic reads for adjacency lists in `GreedySearch(r)`. We can also write $G(r)$ to DRAM since we haven't yet added any in-edges to r in G, so `GreedySearch` can access r in G yet. `InterInsert` then triggers the following loop for each out-edge (r, v) – (a) read $G(v)$ from DRAM, (b) compute updates $G(v)' \leftarrow G(v) \setminus D_{rv} \cup A_{rv}$ for some removed edges D_{rv} and added edges A_{rv}, and (c) write $G(v)'$ to DRAM. Concurrency affects steps (a) and (c), and not (b). We will use Compare and Swap operations (CAS) to write back $G(v)'$ to DRAM to ensure writes are atomic if they succeed. So when can writes fail?

Consider the following sequence of events for some vertex v being modified by 2 threads (thread-0 and thread-1): thread-0 reads $G(v)$ for its step-(a) $\rightarrow$ thread-1 succeeds in CAS$(G(v), G(v)^1)$ for its step-(c) $\rightarrow$ thread-0 attempts to CAS$(G(v), G(v)^0)$ in its step-(c). Thread-0's CAS will fail since thread-1's CAS succeeded and $G(v)^1 \neq G(v)$. So, the insert logic has two options –

- Option-1: Force over-write thread-1's results using CAS$(G(v)^1, G(v)^0)$. This is not a good idea.
- Option-2: Re-run steps (a) and (b) again and retry CAS with updated $G(v)^0$. This is the right way to ensure quality of graph does not degrade over time.

4.1 Logging

CAS operations must be logged before executing. There is a simple mapping between CAS operations and the log entries –

- `TIMESTAMP` – Generated timestamp at logging time
- `TXN-ID` – GUID for insert r
- `OLD-PTR` – Raw 8-byte pointer value of $G(v)$
- `NEW-PTR` – New 8-byte pointer value for $G(v)'$
- `DEL-SET` – D_{rv}
- `ADD-SET` – A_{rv}

If the logged CAS fails, then the insert thread will retry CAS with an updated $G(v)$. If step-(c) executes successfully, the WAL will contain a log entry with a later `TIMESTAMP` with an updated `OLD-PTR`. The previous log entry for the failed CAS can stay in the WAL (i.e. no-op on the failed log entry that might already be persisted to disk) as it is possible to ignore failed CAS log entries while replaying the log. One way – if currently parsing entry with timestamp t_f and old-ptr p_f, this is a failed CAS entry if there exists a successful CAS entry with timestamp $t_s > t_f$ and old-ptr $p_s = p_f$ (and their `TXN-ID` fields will be different).

5 Log replay

Using our persistent WAL, we can recover the state of the graph until the last logged entry in the WAL. If there were inserts or deletes **in flight**, we have 2 options –

- **Undo** in-flight operations using WAL as if they never executed – This is easy to do and requires no additional metadata to be logged to WAL.
- **Redo** in-flight operations using WAL as if they did finish execution – This is also easy to do, but requires us to log the vector `vec(r)` being insert to the WAL as well.

We'll first cover log replay when no operations were in flight at the time the system crashed. We'll then look at the other, more interesting case where there were some operations in flight when the system crashed.

5.1 Redo log operations

Let there be a checkpoint-ed state of the graph G_t at some time t and our WAL has logs for all operations done on G_t from t to some time $t_{crash} > t$ where the system crashed without having any operations in flight. This is easily handled by bringing up G_t into DRAM, replaying each operation in the log as described without running any additional compute (since all operations are consistent and completed in WAL). The final state of the graph G after re-doing the operations in WAL on G_t WAS the state of the graph before the system crashed (strong guarantee as WAL imposes an ordering on the operations as well).

5.2 Redo in-flight operations

Let there be a checkpoint-ed state of the graph G_t at some time t and our WAL has logs for all operations done on G_t from t to some time $t_{crash} > t$ where the system crashed having some set of operations in flight – $O = [O_1, O_2, \ldots, O_k]$ where $\{O_1 \ldots O_k\}$ are sorted in ascending order of the start of their execution in the WAL. In this subsection, we will develop a procedure to replay in-flight operations using the WAL as if they did finish execution and to do so, we will assume that we have `vec(r)` available for all in-flight operations.

Let t_1 be the earliest in-flight operation that was not completed before $t_{crash} > t_1$. Then, for all operations that finished before t_1, we can replay the log using the process discussed above since they are not impacted by any in-flight operations. There may still be many operations that overlap with in-flight operations, and were completed sometime between t_1 and t_{crash}. Since we can recover the state of the graph till t_1, we will assume that we have already done that and that the WAL only contains log entries after t_1. We are now interested in replaying operations from t_1 to t_{crash} and *beyond* – what could have been if all in-flight ops completed.

For ops completed in $[t_1, t_{crash}]$, we can continue replaying as we did before. For in-flight operations, we use graph state at t_1 (i.e. G_{t_1}) to compute the visited sets for insertions (and process any deletions as appropriate) and given some strict ordering of these in-flight operations (any order, shouldn't matter), we will arrive at a graph $G_{t_{complete}}$ when all in-flight operations are completely re-done. $G_{t_{complete}}$ **could have been** the state of the in-memory graph at some point had all the in-flight operations completed, so it is valid and consistent.

5.3 Undo in-flight operations

This is a more interesting case where you would like to retain all inserts that fully completed before t_{crash} in the WAL and we would like to discard all the modifications to the index after the last fully completed insert.

Let t_f be the timestamp for the last completed insert in the WAL and let t_i be the start of the first in-flight operation in the WAL. If $t_i \geq t_f$, we are already done and we can re-do operations till t_f to get the state of the graph before the earliest in-flight operation began in the WAL. What happens when $t_i < t_f$?

Let's pick the span in the WAL corresponding to $[t_i, t_f]$. We need to design a mechanism to selectively keep the updates that belong to completed operations in $[t_i, t_f]$, but discard updates committed by in-flight ops. Let us pick one vertex $v \in G$ and look at updates committed by both in-flight and committed ops. Let the sequence of updates committed to $G(v)$ be described in the WAL as $O = [O_1, O_2, \dots O_k]$. Let $O = O_g \cup O_b,\ O_g \cap O_b = \phi$ be a partitioning of O such that O_g contains all updates committed by the completed operations and O_b are updates committed by the in-flight operations (partially inserted, but committed adjacency list update to $G(v)$ in memory before crash). Let the state of graph at t_i be G_i and $G_i(v)$ be the state of adjacency list for vertex v at time t_i. We can then use the exact set of updates described in each of O_b and O_g to *undo* the ops in O_b using the following steps –

1. Compute set of edges deleted by in-flight ops in O_b: $D_b \leftarrow \left(\bigcup_{O_b} \texttt{DEL-SET}\right)$; filter D_b to not contain any edges to points corresponding to in-flight inserts
2. Similarly, compute set of edges deleted and added by completed ops in O_g: D_g, A_g; filter out any edges to in-flight inserts
3. Compute new candidate edges C as $C \leftarrow G_i(v) \cup D_b \cup A_g \cup D_g$
4. New adjacency list $G(v)$ is then computed using $\texttt{RobustPrune}(v, C)$

This algorithm attempts to restore the edges that were deleted by in-flight inserts and re-computes the best edge set if you executed all updates in O_g at once without committing any updates from O_b. At the end of this algorithm, the adjacency list for the vertex v, $G(v)$ contains no edges to in-flight inserts and its quality is as good as it would have been if O_b did not execute and O_g committed in one operation. So, we've successfully developed an *Undo* operation for the WAL, at least for in-flight inserts.

5.4 Undo completed inserts

Another interesting case where you would like to start at a graph state and roll-back some updates to the graph using the WAL. Here, we will assume that we start with graph state G_t at time t and we would like to roll-back an insertion that started execution in the WAL at time $t_i < t$, so we will assume that the WAL has log entries going back all the way to at least some time $t_0 < t_i < t_f < t$

where t_f is the timestamp when the insert O_r completed. Same as before, we will pick a particular vertex, say v, and show how to roll-back updates from O_r to v.

1. Compute set of all edges added to $G(v)$ after time t_i: $A \leftarrow \left(\bigcup_{t_i}^{t} \texttt{ADD-SET}\right)$; similarly, compute $D \leftarrow \left(\bigcup_{t_i}^{t} \texttt{DEL-SET}\right)$.
2. Filter out any edges in A and D to the operation corresponding to O_r.
3. Compute new candidate edges C as $C \leftarrow G_t(v) \cup D \cup A$
4. New adjacency list $G(v)$ is then computed using $\texttt{RobustPrune}(v, C)$

When this algorithm finishes, the graph does not contain any information about the operation O_r since step-2 prunes out the vertex whose insertion is being rolled-back. The state of graph after the roll-back is also valid ANN graph state since step-4 restores the navigability properties of the graph after rolling back the insertion.

6 ACID Properties

Our transactions have Atomicity, Consistency, and Durability, but not Isolation. We'll discuss each property and how our system behaves wrt the property below –

- **Atomicity** – "*All changes to data are performed as if they are a single operation.*". Each update to an adjacency list (a statement in the transaction) either executes (CAS succeeds) or it does not. If our CAS succeeds, the update was written to our WAL and successfully applied. If our CAS fails, the update is written to WAL, but not applied. If this CAS is replayed, it will be ignored in favour of a later CAS to the same vertex, so we have atomicity in our updates. Further, since x86 guarantees atomic reads at 64 bit granularity, we are guaranteed to see atomic reads in our system. If an insert is in-flight, it will eventually commit its last update and its updates would be written to the graph.
- **Consistency** – "*Data is in a consistent state when a transaction starts and when it ends.*". Our update rules make changes to the graph in a predictable and consistent way. Atomic updates on adjacency lists ensures that we have a valid and consistent ANN graph at every point in time; so our system has consistency.
- **Isolation** – "*The intermediate state of a transaction is invisible to other transactions.*" Our update rules do not provide isolation by design. Isolation would slow down update propagation in the graph for no apparent benefit w.r.t. ANN search. Since the graph itself is an approximation to the true MRNG for the data points, isolation is not necessary as each

statement in the transaction (i.e. each adjacency list update for a given insert) sees a consistent and valid ANN graph. So, Isolation is not necessary and is missing in our system by design.

- **Durability** – "*After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.*". Our WAL allows us to persist the updates from inserts to disk. Since we assume a persistent WAL, when the last update in an insert is committed, it gets persisted to disk. If a crash occurs after the last commit, the WAL contains a record of completion. If a crash occurs before the last commit, the WAL contains a start of transaction, but not the completion; so we can either choose to re-do this in-flight transaction or un-do it using the rules described previously. In both cases, if an inserts is tagged completed, its updates are persisted to disk using the WAL, giving us durability.