Vector DB @ Billions Scale

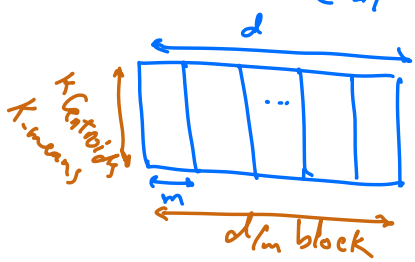
Product Quantization

- Store: Cache, RAM, Disk(SSD)
- Index: Inverted Index, Disk-Resident Graphs
- Search: Table lookup, Modified-Greedy Search

(A) Inverted Index using Product Quantization (IVF...)

↳ Inverted File System

Overview: Given n vectors in $\mathbb{R}^d$, a PQ-Code has a size of $\frac{d}{m}\log k$



e.g., $m = 4$, $k = 16$

$\Rightarrow |PQ\text{-Code}| = \frac{d}{4}\log_2 16 = d$

$\Rightarrow$ Original Size $= 32 \times d$ (Real numbers)

query time?

Brute-Force: Compare $PQ(q)$ to every other Point, return the NN.

$\Theta(n\frac{d}{m}) \ll \Theta(nd)$

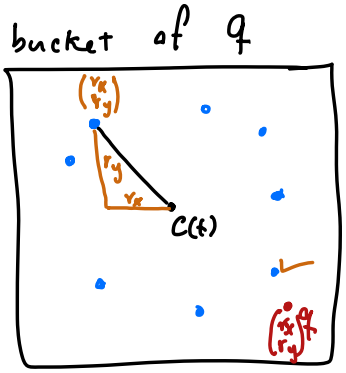
Idea 1: Use $PQ(t)$ as the hash bucket, lookup $PQ(q)$ in the hash Table.

↳ Issue: what if the bucket at $PQ(q)$ is empty?

IVF Idea:

- Coarse Quantization: Pick $m' \gg m$ e.g., $m' = 32$
  + Bucketize Using Large block width (m')
  + Assign each vector to the corresponding bucket
  + let $C^i(t)$ be the Centroid of bucket at t on block i
    ↳ maintain $R^i(t)$ as $PQ^i(t) - C^i(t)$

- Query answering:
  - Find the NN bucket to q
  - For all vectors in bucket at q, Check $Res(t)$ against $Res(q)$ and return the k-most-Sim.



DiskANN: Builds a disk-res. graph

1. Use Product Quantization to Compress vectors.
2. Only keep the PQ-Codes on RAM. ⇒ The graph & full-prec. vectors are stored on Disk.
3. Always start the Search from the Same Node (S)
   S ≃ the medioid of the graph.
   → Cache the vicinity of S.
4. Use a Monotonic Graph (Vamana graph)
5. Modify the Greedy Algorithm to follow best-first-Search instead

---

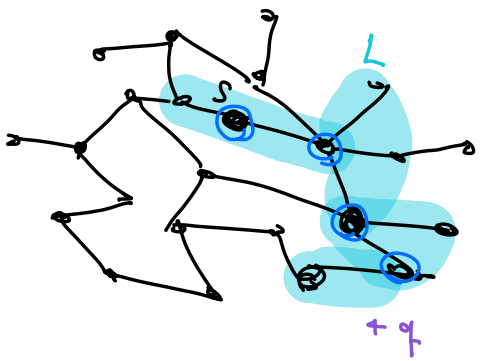
Further optimizations

- Graph Construction: Sharding
- Search: Beam Search (Piggybacking)

Algorithm 1: GreedySearch(s, x_q, k, L)

Data: Graph G with start node s, query x_q, result size k, search list size $L \geq k$
Result: Result set $\mathcal{L}$ containing k-approx NNs, and a set $\mathcal{V}$ containing all the visited nodes

```
begin
    initialize sets L ← {s} and V ← ∅        (→ Promising nodes discovered)
    while L \ V ≠ ∅ do                       (↳ Nodes visited)
        let p* ← arg min_{p∈L\V} ||x_p − x_q||
        update L ← L ∪ N_out(p*) and
          V ← V ∪ {p*}
        if |L| > L then
            update L to retain closest L
              points to x_q
    return [closest k points from L; V]
```



Algorithm 3: Vamana Indexing algorithm

Data: Database P with `n` points where `i`-th point has coords $\mathtt{x}_i$, parameters α, `L`, `R`
Result: Directed graph `G` over P with out-degree `<=R`
begin
  `initialize` G `to a random R-regular directed graph`
  `let s denote the medoid of dataset` P
  `let` σ `denote a random permutation of` $1..n$
  for $1 \leq i \leq n$ **do**
    `let` $[\mathcal{L}; \mathcal{V}] \leftarrow$ GreedySearch$(s, \mathtt{x}_{\sigma(i)}, 1, L)$
    `run` RobustPrune$(\sigma(i), \mathcal{V}, \alpha, \mathtt{R})$ `to update out-neighbors of` $\sigma(i)$
    for *all points* `j` *in* $N_{\text{out}}(\sigma(i))$ **do**
      if $|N_{\text{out}}(j) \cup \{\sigma(i)\}| >$ `R` **then**
        `run` RobustPrune$(j, N_{\text{out}}(j) \cup \{\sigma(i)\}, \alpha, \mathtt{R})$ `to update out-neighbors of` j
      else
        `update` $N_{\text{out}}(j) \leftarrow N_{\text{out}}(j) \cup \sigma(i)$

Algorithm 2: RobustPrune$(p, \mathcal{V}, \alpha, R)$

Data: Graph G, point $p \in P$, candidate set $\mathcal{V}$, distance threshold $\alpha \geq 1$, degree bound R
Result: G is modified by setting at most R new out-neighbors for p
begin
  $\mathcal{V} \leftarrow (\mathcal{V} \cup N_{\text{out}}(p)) \setminus \{p\}$
  $N_{\text{out}}(p) \leftarrow \emptyset$
  while $\mathcal{V} \neq \emptyset$ **do**
    $p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$
    $N_{\text{out}}(p) \leftarrow N_{\text{out}}(p) \cup \{p^*\}$
    if $|N_{\text{out}}(p)| = R$ **then**
      `break`
    for $p' \in \mathcal{V}$ **do**
      if $\alpha \cdot d(p^*, p') \leq d(p, p')$ **then**
        `remove` p' `from` $\mathcal{V}$

