

# Clustering

## 1. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors:

最早能处理十亿级数据的索引结构是基于倒排索引 (IVF)。它先用 K-Means 将特征空间划成 Voronoi 区域，每个区域对应一个聚类中心，检索时只需在少量相关区域中找候选，能在几十毫秒内获得不错的召回率。

后来提出了倒排多重索引 (IMI)，将向量空间切分成多个正交子空间，并分别对每个子空间做 K-Means 分区。最终的空间划分是各子空间区域的笛卡尔积，因此区域数量巨大，非常细粒度，每个区域里数据很少，使得候选集更精确，内存和速度都更高效。

但 IMI 的区域结构也带来问题：区域数量虽然理论上巨大，但大部分区域其实没有数据（空区域）。因为各子空间独立做 K-Means，但真实数据（尤其是 CNN 特征）的子向量之间并非独立，存在较强相关性。结果导致搜索时会浪费时间访问大量空区域，降低检索效率和最终性能。

### 1.1 相关技术：

#### PQ:

一种有损压缩技术，用于把高维向量压缩成很小的编码（通常是几十个字节），常用于大规模向量数据库放不进内存的情况。

做法概念很简单：

把一个 D 维向量切成 M 个子向量，每个子向量做独立的 K-Means 量化。

每个子空间有一个码本 (codebook)，码本通常有 256 个 codewords（这样一个 codeword 用 1 字节即可表示）。

**codebook (码本)** 就是一个“代表向量的集合”，用来近似替代原来的子向量。

通俗理解：

把一个子空间里的许多相似子向量，用少量“典型样本”来代表，这些典型样本就组成 codebook。更正式地说：

- PQ 会把 D 维向量切成 M 个子向量，每个子空间做一次 K-Means；
- K-Means 得到的聚类中心（例如 256 个）就是这个子空间的 **codebook**；
- 每个 codeword 是一个聚类中心，用来表示某个子向量的近似值。

因此，一个向量最终被编码成 M 个字节（每个字节是某个码本中 codeword 的索引），而不必存储原始向量，从而达到压缩效果。

压缩后向量不需要解压就能近似计算与查询向量的欧氏距离。计算时用预先计算好的查找表 (lookup table)，只需 M 次查表与相加即可完成距离估算，这就是所谓的 ADC (Asymmetric Distance Computation)。

#### IVFADC:

IVFADC 是一种大规模向量检索方法，它通过两点来提高速度和节省内存：

- ① 用倒排索引 (IVF) 减少候选数量；
- ② 用产品量化 (PQ) 压缩向量。

具体流程：

- 先用 K-Means 将向量空间划成 K 个区域，每个区域是一个 Voronoi Cell，对应一个中心 (codebook  $C = \{c_1 \dots c_K\}$ )。
- 每个数据库向量只会被分配到离它最近的中心所在的区域，因此搜索时只需要查少数几个区域，而不是全局搜索。
- IVFADC 不直接量化原始向量，而是量化“向量相对区域中心的偏移量” (residual / displacement)。
- 偏移量使用 PQ 编码，而且使用的是全局共享的 PQ codebooks (所有区域共用一套 PQ 码本)。

### IMI:

IMI 是对传统倒排索引 (IVF) 的升级。传统 IVF 用一个全维 K-Means 码本来划分空间，而 IMI 将向量空间切分成多个子空间 (一般是 2 个)，并分别为每个子空间训练一个独立的 K-Means 码本。

做法要点：

- 将 D 维向量分成两个  $D/2$  维子向量；
- 为每个子空间训练一个 K-Means 码本 (各有  $K$  个聚类中心)；
- 最终的空间划分不是单独使用某一个码本，而是这两个子码本区域的 笛卡尔积；

假设向量被分成两半：

- 前半部分用码本 A 聚成  $K=3$  类：A1、A2、A3
- 后半部分用码本 B 聚成  $K=3$  类：B1、B2、B3

如果像 IVF 一样只用一个码本，你只得到 3 个区域。

但 IMI 会组合生成：

(A1,B1)、(A1,B2)、(A1,B3)、(A2,B1)...一直到 (A3,B3)

总区域数 =  $3 \times 3 = 9$

这就是笛卡尔积：把两个子空间的聚类结果“两两组合”。

→ 因此区域数量从  $K$  增长为  $K^2$ ，非常巨大。

好处：

区域变得极度细粒度，每个区域包含的数据更少，搜索时只需访问极小部分区域就能找到最近邻，提高速度。

IMI 还提出了“multi-sequence” 算法，用于高效找出与查询向量最接近的区域顺序。

在压缩方式上，IMI 仍使用 PQ 对向量的残差 (相对区域中心的偏移量) 进行编码，并共享 PQ 码本。

## 1.2 Inverted Index Revisited

### 与IMI的对比：

Structure	Inverted Index	Inverted Multi-Index
Candidate lists quality	Medium	<b>High</b>
Query assignment & indexing cost	Medium	<b>Low</b>
Number of random memory accesses during search	<b>Small</b>	Large
Performance increase from large $K$	<b>High</b>	Small
Memory consumption scalability	<b><math>O(K)</math></b>	<b><math>O(K^2)</math></b>

K 代表 codebook sizes (centroids 的数量)

IMI 的划分非常细，但这种“极细粒度”带来了问题：

### 1) IMI 搜索时需要访问更多区域

因为区域太多、每个区域包含的数据太少，为凑出足够候选 (candidates)，IMI 必须跳来跳去访问很多区域。每跳一次就是一次随机内存访问，而随机访问比 PQ 查表慢，尤其当 PQ 码长很短时，这会拖慢搜索速度。

### 2) IMI 的性能随着 K 增大更快饱和，不如 IVF 受益大

扩大码本大小 K 能让区域划分更精细。

但实验发现：

- 对 IMI，K 增大后划分质量提升有限，很快“饱和”；
- 对 IVF，提高 K 效果更明显，可以持续提升聚类精度。

Inverted Index			Inverted Multi-Index		
K	Average distance	Memory	K	Average distance	Memory
$2^{18}$	0.315	97Mb	$2^{13}$	0.345	256Mb
$2^{20}$	0.282	388Mb	$2^{14}$	0.321	1024Mb
$2^{22}$	0.259	1552Mb	$2^{15}$	0.305	4096Mb

### 3) IMI 的内存开销随 $K^2$ 增长，非常快变大

IMI 要维护  $K^2$  个倒排链表（因为两个子空间码本笛卡尔积）。

如  $K = 2^{15}$ ，则区域数 =  $2^{30}$ ，对十亿数据需要多额外 ~4 字节/向量，这很可观。

因此 IMI 对大码本 不节省内存，甚至变得昂贵。

得出结论：

- 用更大码本的倒排索引 (IVF) 比 IMI 更好：

虽然 IVF 的区域数量比 IMI 少，但增大 K 后，IVF 的分区质量提升更明显。

- 但 阻碍 IVF 使用超大码本的唯一问题是：

查询时需要把 query 分配到 K 个中心中最近的几个，而这个过程是  $O(K)$  的，K 大时会变得很慢。

- 作者指出，现在这个问题已经可以解决：

由于近几年 ANN (近似最近邻搜索) 技术进步，可以用“高精度的近似搜索”来完成 query 的聚类中心分配，不再需要精确比对所有 K 个中心。

实现：

分组 (grouping) 和子区域剪枝 (pruning)，用来提高压缩精度和检索效率。核心解释如下：

#### 1. 目的

在 IVFADC 中，每个倒排索引区域 (region) 包含许多向量，直接用 PQ 压缩整个区域的残差会有一定误差。

为提升压缩精度和召回率，作者提出把每个区域再划分成更小的 子区域 (subregion)，类似 Voronoi 区域。

#### 2. 分组 (grouping) 方法

- 每个区域都有一个中心  $c$ ，找到它的 L 个邻居中心  $s_1, \dots, s_L$ 。
- 每个区域的子中心单独训练，以其中一个区域的中心  $c$  为例，有  $x_1 \dots x_n$  的点属于这个区域，现在要把这些点重新分配给各个子中心
- 子中心 (subcentroid) 不是用 K-Means 训练，而是用 **凸组合**：

$$\text{subcentroid} = c + \alpha(s_l - c)$$

其中  $\alpha$  是可学习的缩放因子，保证内存消耗很小，不需要存储完整的子码本。每个邻居中心都会产生一个新的子中心。

- 原本以  $c$  为重心的点被分配到最近的子中心，同一个子中心的向量连续存储，形成组。（比如  $l = 4$ ，这样原本都属于中心为  $c$  区域的点就又被分成了4个子区域）

## 压缩优化

- 向量相对于子中心的残差比相对于  $c$  的残差更小，因此 PQ 可以更精确压缩（同样码长下误差更小）。
- 提升了压缩精度和召回率。

## 距离计算 (query 时)

$$\begin{aligned} \|q - c - \alpha(s - c) - [r_1, \dots, r_M]\|^2 &= (1 - \alpha)\|q - c\|^2 + \\ &+ \alpha\|q - s\|^2 - 2 \sum_{m=1}^M \langle q_m, r_m \rangle + const(q) \end{aligned}$$

- 计算查询向量到压缩向量的距离，可以通过拆分公式，把距离分成几部分：
  - 1) query 到区域中心的距离
  - 2) query 到子中心的距离（可重用，避免重复计算）
  - 3) PQ 残差部分
- 还有一个与 query 无关的常数项，量化后存储在每个点的编码中。

### 3. 子区域剪枝 (pruning)

- 在搜索时，不必访问每个子区域，只访问最有可能包含最近邻的子区域。
- 实践中可以过滤掉一半子区域而不降低精度，从而加速搜索。

### 4. 缩放因子学习

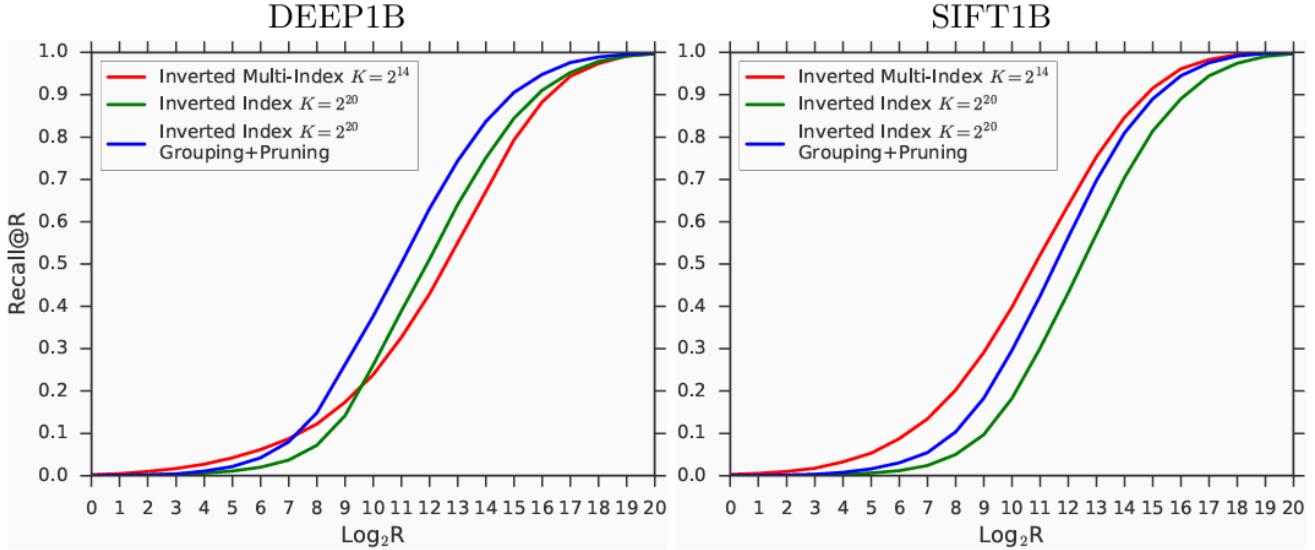
- $\alpha$  通过最小化向量到子中心距离学习得到（约束在  $[0,1]$  内，使子中心在区域中心和邻居中心之间）。
- 先固定每个向量的子中心索引，求最佳  $\alpha$ ；再更新  $\alpha$  得到闭式解。

### 5. 效果

- 分组+剪枝提高了候选列表质量和压缩精度，从而提升整体检索性能。
- 这套方法在倒排索引 (IVF) 上非常有效，但在 IMI 上难以发挥，因为 IMI 的区域数量太多，分组和剪枝的额外开销过大。

## 实验:

*Recall@R* measure which is calculated as a rate of queries for which the true nearest neighbor is presented in the short-list of length R



**Fig. 2.** Recall as a function of the candidate list length for inverted multi-indices with  $K=2^{14}$ , inverted index with  $K=2^{20}$  with and without pruning. On DEEP1B the inverted indices outperform the IMI for all reasonable values of  $R$  by a large margin. For SIFT1B the candidate lists quality of the inverted index with pruning is comparable to the quality of the IMI for  $R$  larger than  $2^{13}$ .

对于 **SIFT1B 数据集** (SIFT 特征是直方图形式) , IMI 在小候选数 ( $R$  较小) 时略优, 因为:

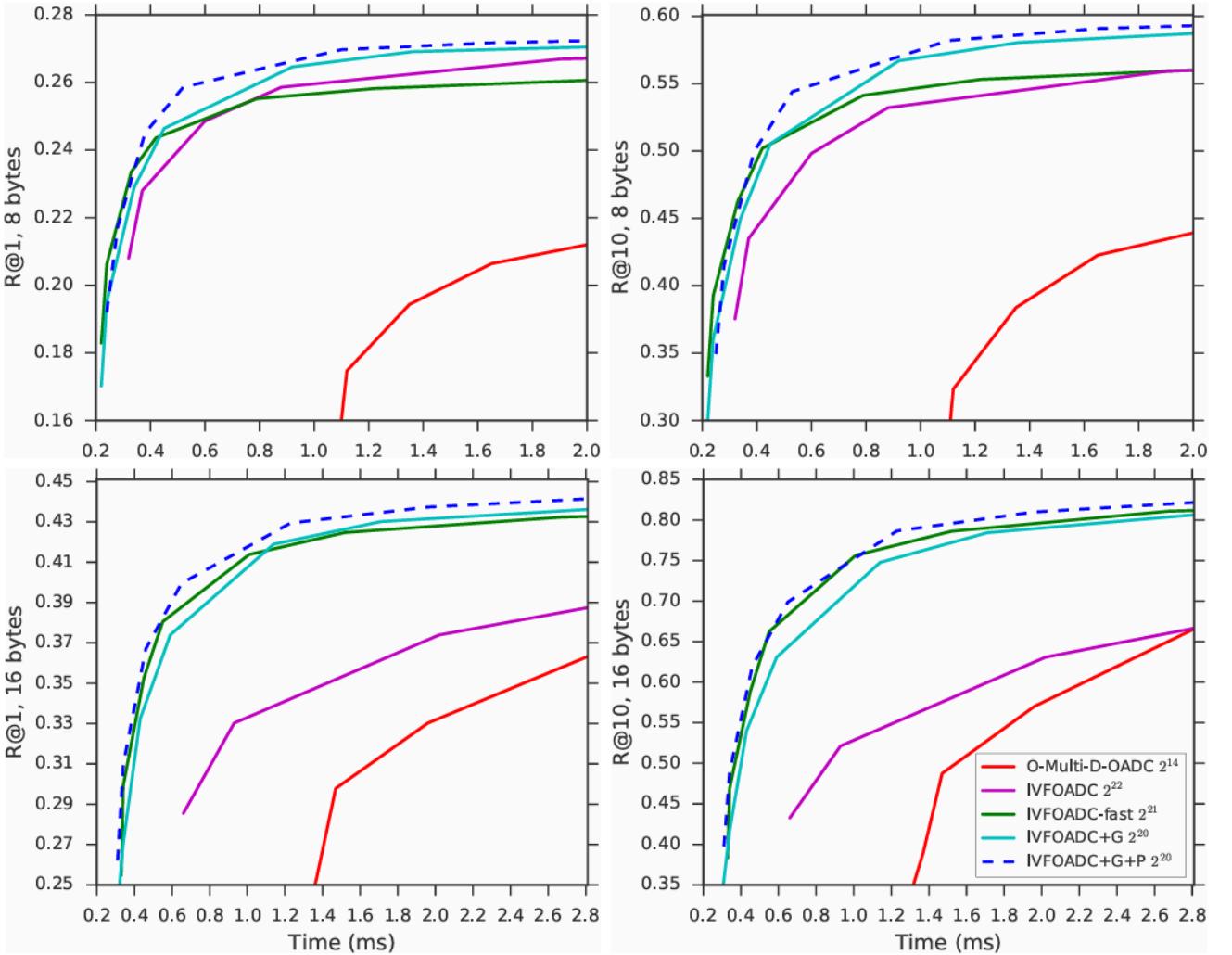
→ SIFT 向量可自然分成几乎独立的两部分 (图像不同区域的特征) , 非常适合 IMI 这种“把高维空间拆成两个子空间分别聚类”的做法。

但当  $R > 2^{13}$  时, 两者性能相当。

不过, IMI 的 **运行开销更大**:

- 它需要多次随机内存访问 (访问多个小区域) ,
- 并使用效率较低的 **multi-sequence algorithm** 来遍历候选区域,
- 导致搜索时间明显更慢。

**搜索时间:**



**Fig. 3.** The  $R@1$  and  $R@10$  values after reranking as functions of runtime on the DEEP1B. The systems based on the inverted index substantially outperform the IMI-based system. The IVFOADC system with grouping outperforms the IVFOADC systems with larger codebooks for the same memory consumption.

横着画线，对应相同的R@1/10，然后比较x轴的t大小

		DEEP1B					SIFT1B				
Method	$K$	R@1	R@10	R@100	t	Mem	R@1	R@10	R@100	t	Mem
O-Multi-D-OADC [24]	$2^{14}$	0.397	0.766	0.909	8.5	17.34	0.360	0.792	0.901	5	17.34
Multi-LOPQ [4]	$2^{14}$	0.41	0.79	-	20	18.68	<b>0.454</b>	<b>0.862</b>	0.908	19	19.22
GNOIMI [5]	$2^{14}$	0.45	0.81	-	20	19.75	-	-	-	-	-
IVFOADC+G+P	$2^{20}$	<b>0.452</b>	<b>0.832</b>	<b>0.947</b>	<b>3.3</b>	17.87	0.405	0.851	<b>0.957</b>	<b>3.5</b>	18

**Table 4.** Comparison to the previous works for 16-byte codes. The search runtimes are reported in milliseconds. We also provide the memory per point required by the retrieval systems (the numbers are in bytes and do not include 4 bytes for point ids).

## 2. Learning to Route in Similarity Graphs:

### 问题：局部最优困境 (local minima)

- 在实际应用中，贪心路由可能会**陷入局部最优**：查询可能停留在某个子最优节点，而无法到达真正的最近邻。
  - 这种情况会导致搜索结果不准确。



## 论文的提出方法：学习路由函数

- 作者提出通过**学习 (learn) 路由函数**来解决局部最优问题。
  - 核心思路是**引入全局结构信息**，不仅仅依赖局部最近邻信息。
  - 具体做法是：对图的每个节点增加一个可学习的表示 (representation)，使得从起始节点到查询最近邻的路由能更加“智能”，避免卡在局部最优节点。

**Learning to Search (L2S)** 方法及其特点，可以分解如下理解：

1. **方法本质**: L2S 是一类用于 **结构化预测 (structured prediction)** 的方法, 它通过学习如何在可能解的空间中“搜索”来找到最优解。换句话说, 它不直接预测结果, 而是学习 **搜索策略**。

## 2. 核心机制：

- 引入一个 **参数化模型 (parametric model)** 来描述搜索过程。
  - 通过训练调整这些参数，使模型尽量模仿或逼近 **最优搜索策略 (optimal search strategy)** 。
  - 在训练中，模型像一个“**搜索智能体 (search agent)**”，学习跟随 **专家策略 (expert search procedure)** 。
  - 专家策略通常由能产生最优结果的算法提供，例如在机器翻译任务中生成最优翻译。

2.1 可学习的随机搜索

在相似性图中进行最近邻搜索的经典方法（beam search）及其改进为可学习的随机搜索

## 1. 传统 Beam Search (算法 1) :

- 从一个初始顶点  $v_0$  开始，维护两组数据：已访问顶点集  $V$  和候选顶点堆  $H$ ，堆中存储每个顶点到查询  $q$  的距离。
  - 每次从堆中选择距离查询最近的顶点  $v_i$  进行扩展，把其邻居加入堆（若未访问过）。
  - 当堆为空或达到运行预算（如距离计算次数上限）时停止，最后返回距离查询最近的  $k$  个顶点。

## 2. 随机化 (Stochastic Search) :

- 不再总是选择最小距离的顶点，而是根据 softmax 概率(所有点相加概率为1) 采样下一个顶点（随机选择）：

$$P(v_i|q, H) = \frac{e^{-d(v_i, q)/\tau}}{\sum_{v_j \in H} e^{-d(v_j, q)/\tau}}$$

- 采样结束后，从已访问顶点集  $V$  中再次根据 softmax 采样  $k$  个顶点作为搜索结果。

对每个访问过的顶点  $v_i \in V$ ，计算它与查询  $q$  的相似度分数（如  $f(v_i) \cdot g(q)$ ）。

用 softmax 把这些分数转成概率：越相似的点，概率越大。

根据这个概率分布，从  $V$  里采样  $k$  个点作为输出结果。

- 当  $\tau \rightarrow 0^+$  时，softmax 的分布会变得非常“尖锐”——概率几乎全集中在分数最高（距离最近）的顶点上。此时算法就退化为普通的贪心 beam search，每次都选最优顶点。

当  $\tau > 0$  时，softmax 分布更平滑，算法在选择下一个顶点时会有一定概率选到次优点，引入随机性，从而能探索更多路径、避免陷入局部最优。

## 3. 引入可学习映射：

- 将距离  $d(v_i, q)$  替换为顶点和查询的可学习向量内积：

$$P(v_i|q, H) = \frac{e^{<f(v_i), g(q)>}}{\sum_{v_j \in H} e^{<f(v_j), g(q)>}}$$

- $f(v_i)$  是数据库点的神经网络映射， $g(q)$  是查询 query 的神经网络映射。

原本 beam search 里使用欧氏距离  $d(v_i, q)$  来判断哪个顶点更接近查询；

现在改为用两个神经网络  $f(v_i)$  和  $g(q)$  的内积 (dot product) 作为相似度：

内积  $f(v_i) \cdot g(q)$  越大，表示两个向量越相似；内积大  $\Rightarrow$  向量方向接近  $\Rightarrow$  特征相似  $\Rightarrow$  数据点更相似  
而搜索算法通常需要“距离越小越相似”的度量；

所以用负号把内积变成“伪距离”：

$$d(v_i, q) = -f(v_i) \cdot g(q)$$

这样越相似（内积大）的点，负内积就越小，被认为“距离更近”。

这让模型能通过学习映射函数  $f$  和  $g$  来优化搜索路径。

## 4. 最终输出步骤：

- 由于训练中使用的是内积而非原始欧氏距离，需要对 top-k 检索结果根据原始欧氏距离重新排序，保证返回真正的最近邻。

## 2.2 最佳路由：

定义一个理想的“参考函数”  $\text{Ref}(H)$ ：（这个函数每次都可以通过图上跳数找到离真实最近邻最近的顶点）

- 这个函数能从当前候选集合  $H$  中选出离真实最近邻  $v^*$  最近的顶点，
- “近”的衡量不是欧氏距离，而是图上跳数 (hops) ——即通过图边需要几步能到达  $v^*$ 。

## 最优路由序列 (optimal routing sequence) :

- 如果一个搜索算法在每次迭代时，都选择  $v_i = \text{Ref}(H)$ ，  
也就是总是扩展那个离真实最近邻最近的节点；

- 并且最终找到了真实最近邻  $v^*$ ,
- 那么这条顶点访问序列就称为“最优路由”。

最后的要求：

- 当搜索结束后，算法输出的 top-k 结果中应包含  $\text{Ref}(V)$ ，即那个真正的最近邻节点。

### 如何在训练阶段实际构造 $\text{Ref}(H)$ 的值：

#### 1. 计算 $\text{Ref}(H)$

- 对每个训练查询  $q$  和候选顶点集合  $H$ ，用 **广度优先搜索 (BFS)** 计算每个顶点到真实最近邻  $v^*$  的图跳数 (hop count)。

#### 2. 加速训练的做法

- 在训练开始前 **预算所有训练查询的跳数**，避免每次迭代重复 BFS。(训练特指神经网络训练)

模型是神经网络，需要优化

- $f(v)$  和  $g(q)$  是可学习的神经网络映射，它们一开始的参数是随机初始化的，
- 一次计算或一次前向传播无法让模型学会正确的“选择最优顶点”的策略。
- 这些预算结果存储在 **持久化缓存** 中，训练时按需读取。

#### 3. 节省内存的策略

- 不存储所有顶点到  $v^*$  的跳数，只存储“可能会被搜索访问的顶点”。
- 用启发式方法筛选这些顶点：
  - 如果存在一条从起点到  $v^*$  的近似最优路径经过它，那么该顶点被选中，
  - 具体来说，考虑所有路径长度最多比最优路径多 5 跳的顶点。

### 训练可学习搜索模型以执行最优路由的目标函数 (loss function) :

#### 1. 训练目标

- 模型的目标是**尽量让随机搜索模仿最优路由 (Opt(q))**，也就是让每一步选择的顶点概率最大化。
- 最直观的方法 (naive approach) 是**最大化最优路径的对数似然**：

$$J_{\text{naive}} = \mathbb{E}_{q,v} \left[ \log P(\text{Opt}(q)|q) \right]$$

- 其中  $\text{Opt}(q)$  表示查询  $q$  的最优路由序列 ( $\text{Ref}(H)$  指定的顶点序列)，
- $P(v_i | H_i)$  是在每一步随机搜索中模型选择该顶点的概率，
- $P(v | \text{TopK}, q, V)$  是最终 top-k 输出中包含真实最近邻  $v$  的概率。

$E_{q,v}[\cdot]$  表示“对所有训练查询  $q$  和对应的真实最近邻  $v$  的平均值”。

- $q$  是训练查询
- $v$  是查询  $q$  的真实最近邻点

#### 作用

- 训练目标  $J_{\text{naive}}$  需要对所有训练样本进行平均，使模型学习到一般性的搜索策略，而不是只针对某一个查询。
- 在实现中，这通常就是**对训练数据 batch 求平均**，类似标准神经网络的 loss 计算方式。

#### 2. 展开形式

$$J_{\text{naive}} = \mathbb{E}_{q,v} \left[ \sum_{v_i \in H_i, H_i \in \text{Opt}(q)} \log P(v_i|q, H_i) + \log P(v \in \text{TopK}|q, V) \right]$$

- 第一项：每一步沿最优路径选择的顶点概率最大化
- 第二项：确保最终 top-k 输出中包含真实最近邻 v

### 3. 意义

- 训练时，通过最大化这个目标函数，模型学会在图中选择“最优顶点”，并最终把最近邻包含在 top-k 结果里。
- 当 top-k 中包含真实最近邻时，即使搜索是随机的，通过 rerank 也能保证成功找到最近邻。

#### 问题所在：

- 如果训练只让模型模仿最优路径 (optimal routing trajectory)，模型只见过“理想状态”的候选集合  $H$ ；
- 没有学习如何处理偏离最优路径的情况。

#### 应用到新查询时的风险

- 一旦模型在某一步选择了非最优顶点（犯错），这个错误顶点会加入堆  $H$ ；
- 下一步的候选集合就和训练时完全不同，模型没有见过这种状态。
- 结果是：模型可能连续犯错，搜索性能大幅下降。

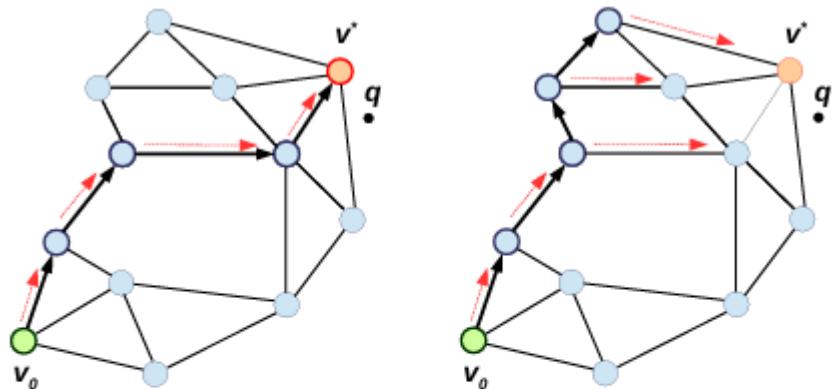
#### 核心问题

- 模型缺乏错误补偿能力 (error recovery)，只模仿理想专家，不能应对实际运行中可能出现的偏离。

#### 改进训练策略：

##### 1. 核心思想

- 允许模型按照当前参数  $f()$ ,  $g()$  在图中搜索，即便它可能选到次优顶点；
- 搜索结束后，再用最优路由的监督信号去更新参数，让模型在已经访问的顶点上尽量跟随最优路径。
- 这样模型学到的不只是理想状态，而是“如何从可能出现的错误状态中纠正回最优路径”。



*Figure 1. Left:* teacher forcing, objective is added up over optimal routing. The sequence of visited vertices is drawn in bold. Training supervision is shown with red arrows. **Right:** imitation learning, model made an error at step 3 and diverged from optimal routing. Objective is computed from the vertices visited by the model.

##### 2. 新的训练目标 ( $J_{\text{imit}}$ )

$$J_{\text{imit}} = \mathbb{E}_{q,v} \left[ \sum_{v_i \in H_i^{\text{Search}(q)}} \log P(v_i = \text{Ref}(H_i) | q, H_i) + \log P(v \in \text{TopK} | q, V) \right]$$

- 这里  $H_i^{\text{Search}(q)}$  是模型当前参数生成的搜索轨迹，而不是仅仅理想轨迹；
- 每个访问过的顶点都用 Ref(H) 作为目标进行监督，确保模型学习在错误状态下也能纠正。

### 3. 区别于原始目标 ( $J_{\text{naive}}$ )

- $J_{\text{naive}}$  只考虑最优轨迹，模型未见过错误状态；
- $J_{\text{imit}}$  考虑模型实际可能走过的轨迹，训练更稳健，对新查询性能更好。

**计算训练目标  $J_{\text{imit}}$  的梯度和 top-k 选择概率：**

#### 1. 梯度计算 (要使得这个概率最大，要求导)

- 第一项  $\log P(v_i = \text{Ref}(H_i) | q, H_i)$  可以直接对模型参数  $\theta$  求导，因为  $P(v_i | q, H_i)$  是由公式(2)的 softmax 定义的，可微分。
- 这一部分就是标准的监督信号，让模型学会在每一步访问的顶点上尽量跟随最优路由。

#### 2. 第二项：top-k 概率

- 这项是真实最近邻 v 被选入最终 top-k 输出的概率。
- 理论上，要计算 exact probability，需要考虑从已访问顶点 V 中选 k 个顶点的所有组合，这对大 k 是不可行的。

#### 3. 近似方法

- 采用类似 Wiseman & Rush 的采样方法：
  - 先从候选集合 V (不含 v) 中依概率采样 k-1 个顶点，不放回；
  - 再计算 v 在剩余集合中被选中的概率。
- 这样就得到一个可微的近似 top-k 概率，因为选顶点时用的是 softmax 概率，梯度可以通过概率传播回神经网络参数 f 和 g；

**实验设置和评价指标：**

#### 1. 计算预算限制 (DCS)

- 为每次查询设置最大距离计算次数 (Distance Computations, DCS)，例如 128、256、512。
- 不同 DCS 对应低、中、高 Recall@1 的实验情境。
- R@x 的计算是在搜索返回的 Top-k 候选结果中进行的，如果  $x \leq k$ ，就统计真实最近邻是否出现在前 x 个中

#### 2. 评价指标：Recall@R

- 计算在 top-R 候选中包含真实最近邻的查询比例。
- 是主要性能指标，用于衡量路由算法在有限计算预算下的准确性。

#### 3. 训练注意事项

- 每个 DCS 值需要单独训练，使得顶点表示  $f(v)$  能适应不同的计算预算限制。

#### 4. 实验图结构

- 使用 Hierarchical Navigable Small World 图 的底层作为搜索图，模型在此基础上学习可训练路由表示。

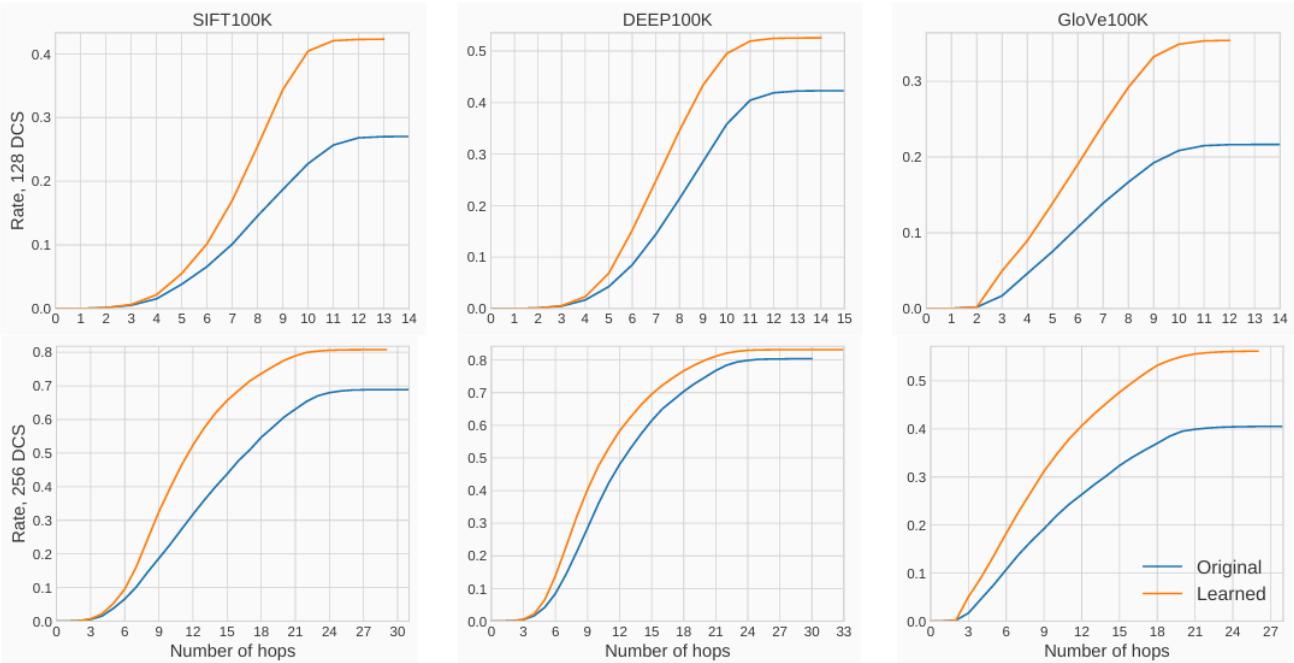


Figure 4. The rates of queries, for which the actual nearest neighbor was successfully found, as a function of hops made by the search algorithm for  $DCS=128$  and  $DCS=256$ . On all the datasets the learned representations provide much higher routing quality.

### 总结:

本文提出了一种可学习的图路由算法用于最近邻搜索 (NNS)。它通过学习节点表示 (embedding)，让搜索过程更容易沿着最优路径到达真正的最近邻。实验表明，在相同计算预算下，它比传统方法更不容易陷入局部最优、召回率更高。代价是需要离线训练一个神经网络，但训练完成后在线搜索阶段几乎不增加额外开销。

## 3. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node

现有 ANNS 索引需要全部放在内存中，因此成本高且无法扩展到十亿规模。DiskANN 利用 SSD + 内存混合设计，让十亿点搜索在普通单机上也能做到高召回、低延迟和高数据密度。在经典 SIFT1B 数据上，它的准确率和速度都显著超过 FAISS、HNSW 等主流方法，并引入了新的图索引结构 Vamana，使整体性能更强。

### 现有数据储存方案：

在亿级或十亿级的大规模数据上，当前工业界主要有两类方案：

第一类为“倒排索引 + 压缩”（如 FAISS、IVFOADC+G+P），通过聚类将数据划分为  $M$  个簇，只搜索最接近的  $m \ll M$  个簇；为了减少内存，这些方法使用 PQ 对向量进行有损压缩，虽然可在  $<5\text{ms}$  内检索，但由于压缩误差， $1\text{-recall}@1$  常仅约 0.5，只能在较弱的  $1\text{-recall}@100$  指标上取得高值。

第二类方法将数据分片，每片构建完整内存索引，如 NSG，但这需要极大的内存。例如 NSG 在  $100M \times 128$  维数据上需约 75GB 内存，因此 10 亿规模往往要用几十台服务器并行服务（如淘宝将 20 亿点划为 32 片部署）。

两类方法都依赖索引完全驻留内存，一旦索引放到 SSD 上，随机访问延迟会让搜索吞吐量急剧下降。FAISS 官方也指出“搜索必须在内存中进行，因为即使是 SSD 也慢几个数量级”。SSD 的随机读通常需要数百微秒，一块普通 SSD 每秒只能处理约 30 万次随机读，而在线搜索希望查询延迟控制在几毫秒以内。因此，要让 SSD 承载高性能 ANN 索引，必须：

- (a) 将每次查询的随机 SSD 访问次数压到几十次以内；
- (b) 将访问 SSD 的往返次数据控制在 5-10 次以内。

而若直接把 HNSW/NSG 等内存索引搬到 SSD 会导致每次查询数百次随机读，延迟不可接受。

作者提出方案：

DiskANN 利用新提出的 **Vamana 图索引** 实现了在 **64GB RAM** 上处理 **十亿级高维数据**，并达到 **95%+ 1-recall@1** 和 **<5ms** 延迟，证明消费级 SSD 也能支撑高性能 ANN 搜索。Vamana 生成的图直径更小，可显著减少 SSD 读取次数，并在内存中同样优于 HNSW/NSG。此外，大数据集可分块构建并高效合并，且系统通过“磁盘存图 + 全精度向量、内存存压缩向量”实现低成本与高精度兼得。

### 3.1 Robust Prune 算法：

---

#### 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  $\mathcal{V}$ 
```

---

输入：点  $p$ 、候选邻居集合  $\mathcal{V}$ 、出度上限  $R$  以及距离阈值。

核心步骤：(对于一个顶点  $v \in \mathcal{V}$ , 从  $v$  出发的边叫做 **出边 (out-edge)** )

1. 初始化  $\mathcal{V}$  为候选集合并加上  $p$  当前出边的邻居，清空  $p$  的出边。 ( $\setminus \{p\}$  表示从集合里面除去  $p$ )
2. 在  $\mathcal{V}$  中选择距离  $p$  最近的点  $p'$  添加为出边。
3. 更新  $\mathcal{V}$ ，删除所有距离比  $p'$  更远的点。
4. 重复直到  $\mathcal{V}$  为空或出边达到上限  $R$ 。

设计思想：

SNG 保证搜索能找到最近邻，但图直径可能很大（如一维线性排列的点，直径  $O(n)$ ），导致磁盘存储时搜索需要大量顺序读。

Robust Pruning 则通过引入“距离缩放因子  $> 1$ ”的约束：搜索路径上，每步距离查询点都缩短至少一个固定比例，从而使搜索对数级收敛。

### 3.2 Vamana算法:

结合Greedy Search和Robust Prune:

---

#### **Algorithm 1:** GreedySearch( $s, \mathbf{x}_q, k, L$ )

---

**Data:** Graph  $G$  with start node  $s$ , query  $\mathbf{x}_q$ , 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  $\mathcal{L} \leftarrow \{s\}$  and  $\mathcal{V} \leftarrow \emptyset$ 
while  $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$  do
    let  $p^* \leftarrow \arg \min_{p \in \mathcal{L} \setminus \mathcal{V}} \|\mathbf{x}_p - \mathbf{x}_q\|$ 
    update  $\mathcal{L} \leftarrow \mathcal{L} \cup N_{\text{out}}(p^*)$  and
         $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$ 
    if  $|\mathcal{L}| > L$  then
        update  $\mathcal{L}$  to retain closest  $L$ 
        points to  $\mathbf{x}_q$ 

```

**return** [closest  $k$  points from  $\mathcal{L}$ ;  $\mathcal{V}$ ]

---

Vamana 仅选择一个远小于  $n - 1$  的候选集合  $V$  来执行 Robust Prune，从而加速索引构建，同时保持图的高性能搜索特性。

---

#### **Algorithm 3:** Vamana Indexing algorithm

---

**Data:** Database  $P$  with  $n$  points where  $i$ -th point has coords  $\mathbf{x}_i$ , parameters  $\alpha, L, R$

**Result:** Directed graph  $G$  over  $P$  with out-degree  $\leq R$

**begin**

```

initialize  $G$  to a random  $R$ -regular directed graph
let  $s$  denote the medoid of dataset  $P$ 
let  $\sigma$  denote a random permutation of  $1..n$ 
for  $1 \leq i \leq n$  do
    let  $[\mathcal{L}; \mathcal{V}] \leftarrow \text{GreedySearch}(s, \mathbf{x}_{\sigma(i)}, 1, L)$ 
    run RobustPrune( $\sigma(i), \mathcal{V}, \alpha, R$ ) 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( $j, N_{\text{out}}(j) \cup \{\sigma(i)\}, \alpha, R$ ) to update out-neighbors of  $j$ 
        else
            update  $N_{\text{out}}(j) \leftarrow N_{\text{out}}(j) \cup \sigma(i)$ 

```

---

Vamana 索引算法通过迭代构建有向图  $G$  来支持高效的 GreedySearch:

1. 初始化图：

- 每个顶点随机选择  $R$  个出邻居 (out-neighbors) 。

## 2. 选择起点:

- 取数据集  $P$  的中位点 (medoid)  $s$  作为 GreedySearch 的起点。

## 3. 迭代优化图:

- 按随机顺序遍历每个点  $p \in P$ 。
- 对当前图  $G$  运行 `GreedySearch(s \to p)`, 得到经过的点集合  $V_p$ 。
- 使用 `RobustPrune(p, V_p, R)` 更新  $p$  的出边, 使其更适合收敛到  $p$ 。
- 为每个  $p' \in N_{\text{out}}(p)$  添加**反向边** ( $p' \rightarrow p$ ), 确保搜索路径上的节点与  $p$  相连。
- 如果添加反向边导致某节点出度超过  $R$ , 再次运行 `RobustPrune` 限制出度。
- 以一个点作为例子:

```
let [ $\mathcal{L}; \mathcal{V}$ ]  $\leftarrow$  GreedySearch( $s, x_{\sigma(i)}, 1, L$ )
```

先通过GreedySearch找到从中心点s出发到这个点坐标需要经过的所有节点集合为V

然后

```
run RobustPrune( $\sigma(i), \mathcal{V}, \alpha, R$ ) to update out-neighbors of  $\sigma(i)$ 
```

通过RobustPrune更新这个点与这些邻居节点的连接 (出边: 从这个点出去连到这些邻居)

然后

```
for all points j in  $N_{\text{out}}(\sigma(i))$  do
  if  $|N_{\text{out}}(j) \cup \{\sigma(i)\}| > R$  then
    run RobustPrune( $j, N_{\text{out}}(j) \cup \{\sigma(i)\}, \alpha, R$ ) to update out-neighbors of j
  else
    update  $N_{\text{out}}(j) \leftarrow N_{\text{out}}(j) \cup \sigma(i)$ 
```

对于被选择作为出边的点, 还需要为它们添加反向边, 使得搜索路径上的节点能互相连接

同时在这里添加出边的时候, 它们自己本身也会有出邻居, 因此再添加反向边之后, 出边个数可能会超出  $R$ , 需要再次进行剪枝

添加反向边时, **边的方向是从每个出邻居  $p' \in N_{\text{out}}(p)$  指向  $p$** , 即增加的是  $p'$  的出度, 而不是  $p$  的出度。

如果某个  $p'$  已经有接近  $R$  的出度, 新增  $p' \rightarrow p$  的反向边就可能使  $p'$  的出度超过  $R$ 。

## 4. 多轮迭代:

- 算法总共做两轮遍历:

- 第一轮使用  $\alpha = 1$
- 第二轮使用用户定义的  $\alpha_1$

a 用来保证搜索路径每步距离查询点都有“足够”下降。

如果节点  $p$  的候选邻居  $q$  满足

$$d(q, \text{query}) \leq \frac{d(p, \text{query})}{a}$$

才保留  $q$  作为出邻居。

换句话说,  $a$  控制 **搜索路径的收敛速度**:

- 第二轮可以生成更优的图, 但如果第一轮也用较大  $\alpha_1$ , 会产生平均更高出度的图, 导致构建速度变

慢。

**总体效果：**随着迭代进行，图结构越来越适合 GreedySearch，搜索速度和准确率不断提升。

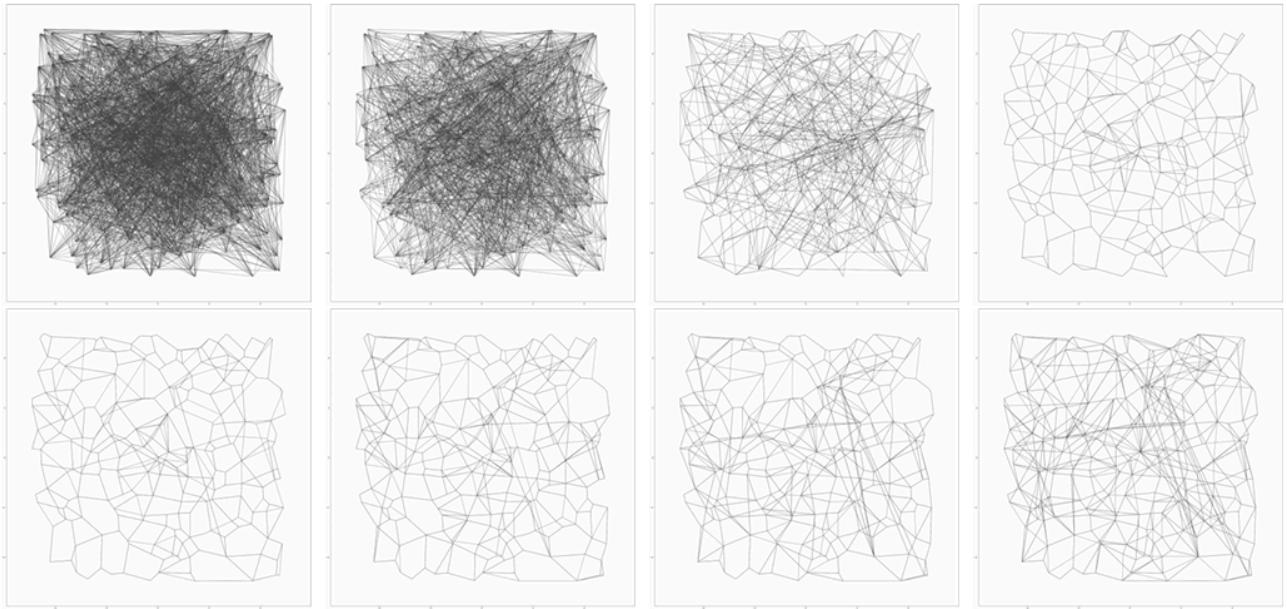


Figure 1: Progression of the graph generated by the Vamana indexing algorithm described in Algorithm 3 on a database with 200 points in 2 dimensions. Notice that the algorithm goes through the first pass with  $\alpha = 1$ , followed by the second pass where it introduces long range edges.

与 HNSW、NSG 的构建思路对比：

1. 可调参数  $\alpha$ :

- HNSW 和 NSG 没有可调  $\alpha$ , 默认  $\alpha = 1$ 。
- Vamana 可调  $\alpha$ , 使得在图的 **出度 (degree)** 与 **直径 (diameter)** 间实现更好的权衡。

2. 候选集合  $V$  的定义:

- HNSW:  $V$  是 GreedySearch 输出的最终  $L$  个候选点。
- Vamana / NSG:  $V$  是 GreedySearch 遍历过程中访问过的所有节点。
- 影响: Vamana / NSG 可以增加**长程边**, 而 HNSW 只能加局部边, 因此 HNSW 需要额外构建**多层图 (hierarchy)** 来弥补搜索范围。

3. 初始图构建:

- NSG: 从近似 KNN 图开始, 构建耗时且占内存。
- HNSW: 从空图开始。
- Vamana: 从**随机图**开始, 实验证明比空图得到更高质量的图。

4. 遍历次数:

- HNSW / NSG: 只遍历一次数据集。
- Vamana: 遍历两次, 第二轮进一步优化图质量, 提高搜索性能。

### 3.3 储存和取：

DiskANN 的目标是在单机 64GB 内存 + 普通 SSD 上完成十亿级向量的近似最近邻搜索。

主要困难有两个：

#### (1) 如何构建十亿点的图？内存放不下。

直接跑 Vamana 不可能，因为：

- 十亿点  $\times$  100 维的原向量数据远超 64GB 内存；
- Vamana 构图需要访问所有点向量。

在构建大规模 ANN 索引时，一个常见想法是：先用 k-means 把数据分成多个小簇（shards），每个簇分别构建自己的图索引。查询时，只把 query 发到和它最接近的几个簇中搜索。这样单个簇的索引可以放入内存，从而解决“10 亿点数据放不下”的问题。

但这种传统方式有两个明显缺点：

1. **延迟变高**：查询必须被路由到多个簇，意味着多次索引访问与多次搜索启动。
2. **吞吐量下降**：每个查询都要处理多个簇，因此整体系统压力增大，完成同样 QPS (Queries per second) 需要更多计算。

DiskANN 的核心创新是反过来思考：

与其在查询时把 query 发往多个簇，不如在**构建阶段**就让每个数据点属于多个簇，也就是说——**让簇之间重叠**。

#### DiskANN 的方案：重叠聚类 + 合并多个小图

步骤：

1. 用 k-means 将 10 亿数据分成 k 个簇， $k \approx 40$ 。
2. 每个点不只属于一个簇，而是属于 **r 个最近的簇中心**（通常  $r=2$ ）。  
这两个簇的交叠确保不同簇之间有足够连接。
3. 每个簇只有  $N/k$  点（例如  $10 \text{ 亿} / 40 = 2500 \text{ 万}$ ），可以在内存内跑 Vamana。
4. 最终把所有簇的图简单做 **边集合并**（union），得到一个统一大图。

#### 为什么可行？

因为重叠使得图有足够的跨簇的“桥”，GreedySearch 即使跨簇也能找到最短路径。

---

#### (2) 搜索时向量数据没法全部放进内存，怎么计算距离？

DiskANN 的做法：

- 原始向量（float）存 SSD。
- 内存中只保存 **压缩后的向量**（例如 PQ 压缩后 32 bytes）。
- 搜索的距离计算使用压缩向量，虽然不精确但**足够好**。
- 但 Vamana 构图使用的是**全精度**向量，因此图结构非常精确，能很好地引导 GreedySearch。

这就避免了“向量放不进内存”的问题，同时保持高 recall。

#### 具体储存方式：

DiskANN 将压缩向量存于内存，将全精度向量与图结构放在 SSD 上。为便于快速定位，磁盘中每个点采用定长记录：先存全精度向量，再存 R 个邻居 ID，不足部分以 0 填充。这样只需一次乘法即可计算任意点的磁盘偏移，避免在内存中维护额外指针，并使 SSD 能连续读取多个点记录。

由于 SSD 随机读取的延迟较高，若按传统 GreedySearch 方式逐点取邻居，会产生大量 round-trip，导致查询延迟不可接受。为此，DiskANN 采用 BeamSearch：在每轮迭代中利用内存中的 PQ 压缩向量快速选出候选集合 L 中距离最近的 W 个点，并一次性从 SSD 读取它们的邻居记录。因为读取多个相邻扇区几乎与读取一个扇区的成本相同，这种批量 I/O 能显著减少随机访问次数。所有新邻居再依据压缩距离加入 L 并重新筛选，只需少量迭代即可收敛。（W=1 的时候就相当于 GreedySearch）

W 的选择需要在“减少 I/O 往返”和“避免过多无效计算”之间取得平衡。虽然基于 NAND-flash 的 SSD 理论上每秒可以处理 50 万次以上的随机读取，但要达到这个最大吞吐量，必须让 SSD 的所有 I/O 队列长期处于“排队饱和”状态。这样虽然吞吐量高，但会带来显著的副作用：所有读请求都要在队列中等待，导致每次随机读取的延迟上升到毫秒级，明显拉高 ANN 查询延迟。

为了保持低延迟，DiskANN 不能把 SSD 压满运行，而需要让设备维持较低的负载因子（load factor）。实验显示，使用较小的 Beam Width（如 W = 2,4,8）可以让 SSD 的使用率稳定在约 30–40%，既不会造成 I/O 排队，也能维持较快的响应时间。在这种设置下，搜索线程的时间分布大致为：40–50% 用于 SSD I/O（因为仍需读取若干邻居记录），剩余时间用于压缩向量距离计算和候选更新。

DiskANN 为了减少查询过程中的 SSD 访问次数，引入了两个关键机制：**热点节点缓存（Caching）** 和 **基于全精度向量的隐式重排序（Implicit Re-ranking）**。

首先，在图搜索中，有些节点会被反复访问，例如距离起点 s（通常为 medoid）3~4 跳以内的节点，因为 Greedy/Beam 搜索在早期阶段总会经过这些区域。DiskANN 会将这些“频繁访问节点”的数据直接缓存到 DRAM 中，这样每次搜索都能在内存中完成前几跳的扩张，而不需要反复访问 SSD。

第二，DiskANN 解决 PQ（Product Quantization）带来的“距离误差”问题。PQ 是有损压缩，搜索时用 PQ 距离选出的候选点与真实欧氏距离并不完全一致，可能导致 Top-k 排序出现误差。DiskANN 的做法是：在 SSD 上将每个节点的“邻居列表 + 全精度向量”存放在同一个 4KB 扇区内。SSD 的随机读最小单位就是 4KB，因此读取节点邻居时顺带把完整向量也读入了内存，并不会增加额外的 I/O 成本。这样 BeamSearch 在扩展节点时就能逐步收集所有访问节点的全精度向量，最终使用真实距离重新排序 Top-k，确保结果精度接近内存型 ANN 系统。

**评估测试：**

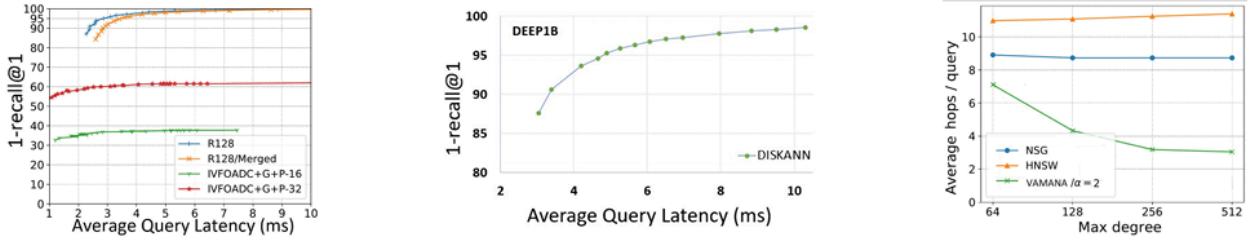


Figure 2: (a)1-recall@1 vs latency on SIFT bigann dataset. The R128 and R128/Merged series represent the one-shot and merged Vamana index constructions, respectively. (b)1-recall@1 vs latency on DEEP1B dataset. (c) Average number of hops vs maximum graph degree for achieving 98% 5-recall@5 on SIFT1M.

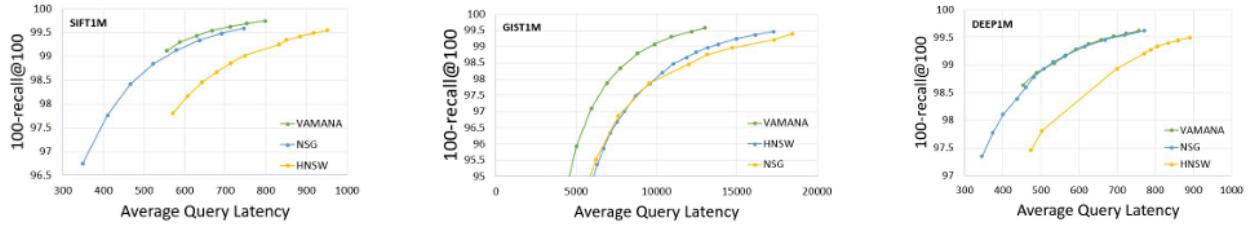


Figure 3: Latency (microseconds) vs recall plots comparing HNSW, NSG and Vamana.

对十亿级数据集的实验主要比较了两种 Vamana 构建方式：**One-Shot Vamana (单次构建) \*与\*Merged Vamana (分片合并)**。在 One-Shot 模式中，作者直接在  $10^9$  条 SIFT 数据上构建单一图索引，参数设为  $L=125$ 、 $R=128$ 、 $\alpha=2$ 。这种方式需要极大的内存支持：在 M64-32ms (约 1.8 TB 内存) 上运行约 2 天，峰值内存达到 1100 GB，最终得到一个平均度数约 1139 的高密度图。相比之下，Merged Vamana 通过“分片 + 合并”降低构建压力：首先用 k-means 将数据划分成 40 个子集，并将每个点分配给其最邻近的两个簇；随后在每个子集中用较小的  $R=64$  构建局部图，最后将所有子图的边集并入一个全局索引。这种方法只需 64GB 内存，整个过程在 z840 上约 5 天即可完成，最终索引规模为 348GB、平均度数为 921，大幅降低了构建资源需求。

从查询性能来看，单索引的结构更完整，搜索路径更短，因此**在相同 recall 下延迟更低**。例如，在 bigann 的 10k-query 测试中，One-Shot Vamana 以 <5ms 的延迟达到了接近 98.68% 的 1-recall@1，刷新了当时的单机 SSD-based ANN 搜索性能记录。而 Merged Vamana 因为每个点的边集在合并后受到 shard 限制（大约只覆盖全体数据的 5%），导致访问邻域需要跨更多边，搜索会多走若干跳，因此整体延迟比单索引高约 20%。尽管如此，Merged Vamana 依旧明显优于当时其他最先进方法，而且能够在**仅 64GB 内存下完成十亿级数据集索引构建**，在资源受限场景中是一种非常实用的方案。

整体来看，One-Shot Vamana 是追求极致性能的最佳选择，而 Merged Vamana 则在资源需求与性能之间取得了非常实际的工程折中。

### 总结：

作者提出了新的图索引算法 Vamana，在高召回场景下，其内存搜索性能达到或超过现有最优方法。同时，作者利用仅 64GB 内存，在十亿级数据上构建了基于 SSD 的 DiskANN 索引，并实现了毫秒级延迟。整体而言，他们把图方法的高召回/低延迟优势与压缩方法的高可扩展性结合起来，达成了当前规模下的最新性能水平。

## 4. Unleashing Graph Partitioning for Large-Scale Nearest Neighbor Search

### 摘要：

研究怎样把超大规模的近邻搜索任务切成多个更小的“分片”，并希望每个分片都能保留邻居关系，这样查询时只需要查少数几个分片。论文提出了新的“路由算法”，在查询时快速判断该查哪些分片，而且这些路由方法是“模块化”的——可以跟任何分区方式搭配，不像以前的方法强依赖特定的分区结构。

### 问题及解决方案：

作者指出：图索引虽然召回最高，但无法分布式扩展，因为图的遍历依赖性强、跨机器通信大。因此分布式系统普遍采用“先分片再路由”的方式：把数据分成多个 shard；查询时先用轻量级路由判断要查哪些 shard，再在这些 shard 内做本地搜索（图索引或暴力搜索）。这其实就是 IVF 的思想，分片质量决定性能。

接着作者指出：现有路由方法往往绑定特定的分片方式，例如 k-means 的 routing 就是“把 query 距离最近的中心挑出来”。但如果换成别的分区方法，往往没有对应的 routing。

然后引出最近被发现的“平衡图划分 (Balanced Graph Partitioning) ”：先构建 kNN 图，再对图做平衡分区，尽可能减少跨 shard 的边，这样最多邻居会落在同一个 shard 里。实验证明这种分片质量远高于 k-means，但有两个致命缺点：

- 1) 这种 shard 没有几何形状，无法像 k-means 那样靠“中心”做路由，只能训练代价很高的神经网络路由模型。
- 2) 必须先构建 kNN 图才能做图划分，而 kNN 图本身又需要 ANNS，形成“鸡生蛋”问题。

作者针对前面提到的平衡图划分问题提出了解决方案，让它能用于十亿级别的 ANNS。主要贡献有四点：

1. **快速粗略建图**：用递归球切分 (dense ball-carving) 快速构建一个粗略的 kNN 图就够了，虽然近似，但能保证分片质量和查询性能。
2. **快速、精确、模块化的组合式路由**：设计了两种通用路由方法，可搭配任何分区，尤其是图划分。
  - **KRT**：先对每个 shard 做层次 k-means 子聚类，再用树或 HNSW 找最近中心点，把 query 路由到对应 shard。
  - **HRT**：用 LSH，把 query 放在 LSH hash 排序里，找附近点对应的 shard。
3. **路由的理论保证**：对 HRT 的两个变体进行了理论分析，证明每个 query 高概率会路由到包含近似最近邻的 shard，这是路由环节首次有严格理论保证。
4. **实验验证**：在十亿级数据上，使用平衡图划分的 shards 比最优的竞品分区快 1.19–2.14×，top-shard 的邻居集中度高 25%，KRT 的训练时间也快几个数量级（半小时搞定十亿点），比神经网络路由方法快很多，同时 recall 更好或相当。

### 4.1 快速近似 k-NN 图构建方法：

作者方法是递归划分 dense ball 簇：先随机选一些 pivots，把每个点分到最近的枢轴簇中，然后簇内递归继续划分或生成簇内两两边。直觉是 top-k 邻居大概率会落在同一簇。为了提升图质量，他们做了**独立重复**和**在第一次递归时把点分配给多个 pivots**，避免指数级开销。

关键点：

- 图不需要精确 top-k，只要捕捉粗略局部结构即可保证分片质量。
- 实验发现，即便图 recall 很低 (<0.3)，查询时 top-10 邻居仍有 96% 聚集在同一 shard：
- 增加每点边数 (degree) 有时会略微降低查询 recall，因为在低质量近似图中，多边会“污染”结构，但在精确

图中不会出现。

结论：构建稀疏、粗略的近似 k-NN 图就够用于分片，不会显著影响查询性能。

## 4.2 重叠分片减少边界点的召回损失：

当分片是互斥的 (disjoint) 时，边界点的 k-NN 可能跨 shard，导致查询丢邻居。作者提出一种贪心算法（受 Fiduccia & Mattheyses 图划分启发）：复制节点，把它放到包含大多数邻居的 shard，从而减少跨 shard 边 (cut edges)，也就减少召回损失。

### cut-edge 的来源

1. 你先有一张 k-NN 图：

- 每个点连接它的 k 个最近邻
- 这些连接构成图中的边

2. 然后你对所有点进行 分片 (sharding / partitioning) :

比如把数据分成 16 个 shard。

3. cut-edge 的定义：

如果一条 k-NN 边连接的两个点被分到了不同的 shard，这条边就叫 **cut-edge**。

也就是说：

- **cut-edge = 跨分片的 k-NN 边**
- **当一个点的邻居被划到别的 shard，查询时就可能 miss 掉它 → 召回下降**

关键细节：

- 引入参数  $o \geq 1$  限制节点复制量，同时保持 shard 最大小不变，通过增加 shard 数量实现。
- 算法每次选择能消除最多 cut edges 的节点进行复制，重复直到没有满足条件的节点。
- 可以批量并行处理：多个节点如果消除相同数量的 cut edges 可以同时处理。
- 只需少量轮次，因为每个节点至少消除一个 cut edge，k-NN 图的度数有限。

总结一句话：**重叠分片通过复制边界节点，把大部分邻居保存在同一 shard，提高查询 recall，同时保持 shard 尺寸可控且支持并行化。**

## 4.3 Routing策略

一、Routing 的核心目标是什么？

Routing 的任务是：

**给一个 query q，挑出很少几个 shard，但这些 shard 里要包含 q 的大部分 kNN。**

---

二、为什么传统的“一个中心代表一个 shard”在这里完全失效？

论文引用了经典 IVF 思想 (IVF = inverted file index)，其特点：

- IVF 通常把  $10^9$  点切成  $10^6$  个小 cluster
- 每个 cluster 大小 ~1000 点
- 这种小 cluster 形状简单，一个中心即可代表

但是本论文的场景完全不同：

- shard 数量非常小  $s \in [10, 100]$
- 但每个 shard 非常巨大  $|S_i| > 10^7$
- shard 不是凸区域，不是单峰结构，是由图分割得到，形状奇怪（不规则、断裂、洞、多支路）

这就导致一个重大的几何问题：

一个中心无法代表一个巨大的、非凸、内部有数十个 sub-clusters 的 shard。

论文给出的实际数据点：

- 用单中心 routing，MS Turing 数据集的 top-1 shard recall 只有 28%

这说明中心路由完全不能工作。

### 三、为什么 shard “不规则、不凸”会导致 routing 失败？

这是一个很关键但容易忽略的点。

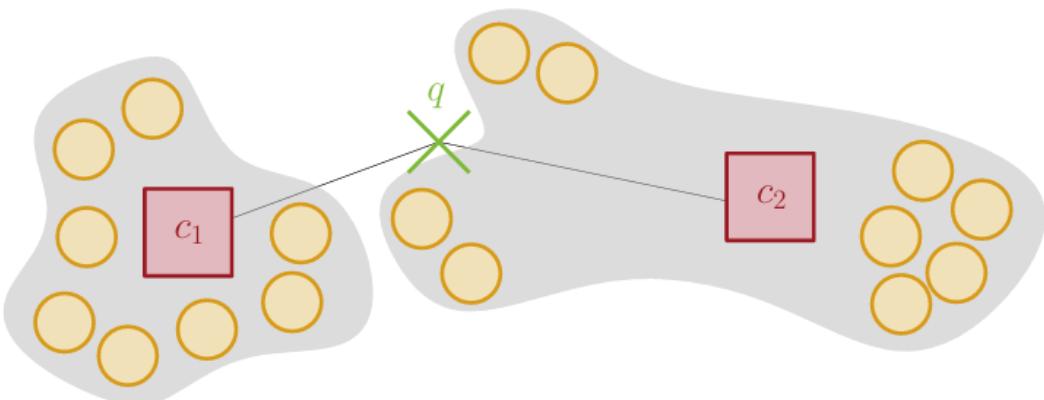
k-means 得到的是 convex clusters (凸区域)。

但 graph partitioning 得到的是基于图连通结构的 partition:

- shard 可能被图结构拉成“长条”“许多分支”“多个中心”
- shard 可能由若干个远离的 dense region 组成 (sub-clusters)

所以如果你用每个 shard 的“一个中心”进行 routing：

- query 可能离某个 sub-cluster 非常近，但离整体中心很远  
→ routing 会错误地跳过本该访问的 shard  
→ recall 下降



*Figure 2. Illustration of an example where routing using a single center per shard fails. The nearest neighbors of  $q$  are in the cluster of  $c_2$ , but  $d(q, c_1) < d(q, c_2)$ . If the hierarchical sub-clusters are represented with their own centers, the routing works correctly.*

### 四、他们的核心思想：每个 shard 不用一个中心，而是用多点表示 $R_i$

论文提出：

每个 shard  $S_i \subset P$ , 用一组点  $R_i$  来代表它的几何分布。 $R_i$  内含多个代表点, 可覆盖多个 sub-cluster。

所有 shard 的代表点集合:

$$\bigcup R_i = R \text{ (总共 } m \text{ 个点)}$$

Routing 时:

1. 给 query  $q$
2. 在  $R$  中找  $q$  的最近邻若干点, 得到  $Q$
3. 查看  $Q \in$  哪些 shard
4. 将 shard 按“ $Q \cap R_i$  中离  $q$  最近的距离”排序
5. 按顺序访问 shard

这叫做 **multi-center probabilistic routing**。

优势:

- 能捕捉 shard 内多个 cluster 的结构
- 即使 shard 的几何边界复杂, 仍能正确判断 query 靠近哪个 shard

#### 4.3.1 K-Means Tree Routing Index 训练与查询

建图:

一、KRT 的总体思想

目标: 为每个 shard 构建一个层次化 k-means 树, 用于 coarse routing。

每个 shard  $S_i$  会独立建一棵 tree:

- 树的所有节点是 k-means 聚类中心
- 所有层的中心点共同组成代表点集合  $R_i$
- 查询时通过这棵树找到与 query  $q$  最接近的若干 representative points  $Q$
- 进而决定优先访问哪些 shard

注意: 这个树不是用来存实际点的, 只存“代表点 (centroids) ”用于 routing  
——这是 KRT 与传统 k-means tree 的最大区别。

```
root
 / / ... \ \
c1 c2 ... c31 c32
/|\ \
...
...
```

训练过程:

---

**Algorithm 1:** KRT: Training

---

**Input:** Shards  $S_i$ , number of centroids  $l$ , index size  $m$ , cluster size  $\lambda$

**for**  $i = 1$  to  $s$  **do in parallel**

create root node  $v_i$

Build( $S_i, l, \frac{|S_i|(m-s)}{|P|}, \lambda, v_i$ )

**func** Build( $P, l, m, \lambda, v$ ):

**if**  $m \leq 1$  **return**

centroids( $v$ )  $\leftarrow$  K-Means( $P, l$ )

**for**  $c \in$  centroids( $v$ ) **do in parallel**

$P_c \leftarrow$  k-means cluster of  $c$

create new tree-node  $v_c$

add tree-edge from parent  $v$  to  $v_c$

**if**  $|P_c| > \lambda$

Build( $P_c, l, \frac{(m-l)|P_c|}{|P|}, \lambda, v_c$ )

---

$P$ : 当前递归处理的点集 (最开始是某个 shard  $S_i$ )。

每个节点  $v$  都有自己的点集  $P$ :

- 根节点的  $P$  是整个 shard ( $S_i$ )
- 第二层每个节点的  $P$  是对应的一个簇  $P_c$
- 第三层的  $P$  是  $P_c$  再次细分后的子簇
- ...
- 一直到  $P$  的大小小于  $\lambda$ , 就不再继续建树

$s$ : shard 数量。

$l$ : 每个非叶节点要产生的 centroid 数。

$m$ : 全局代表点 (centroid) 预算, 上限 (routing index 总大小)。

$\lambda$ : 停止递归的最小簇大小阈值。

$v_i$ : 第  $i$  个 shard 的 root 节点 (树根)。

Build( $P, l, m, \lambda, v$ ): 对点集  $P$  以预算  $m$  构造以  $v$  为根的子树并把 centroids 存入  $v$ 。

按照步骤解释:

1. 并行对每个 shard 创建根并分配初始预算

**for**  $i = 1$  to  $s$  **do in parallel**

create root node  $v_i$

Build( $S_i, l, \frac{|S_i|(m-s)}{|P|}, \lambda, v_i$ )

- 含义：对每个 shard 单独建一棵“k-means 树”。训练是按 shard 并行的（多线程/多机友好）。
- 为什么用  $|S_i| * (m-s)/|P|$ ：把全局预算  $m$  按点数比例分配给各 shard。减去  $s$  是因为根层（或每个 shard 至少会有一个代表）需要预留预算的一部分（实现上是为了避免把  $m$  全部分配给子树而忽略 root centroids）。

它决定子节点的“目标树大小”（即还能继续往下分多少层）。

## 2. Build 函数 — 递归终止条件

```
if m <= 1 return
```

- 含义：如果给当前子树分配的预算不足以再放一个有效 centroid（或不值得继续），就停止递归，不再在此子树创建新节点。

## 3. 在当前节点做一次 l-means

```
centroids(v) ← K-Means(P, 1)
```

- 含义：对当前子树的点集  $P$  做一次 k-means，生成  $l$  个中心向量并把它们存为当前节点  $v$  的 centroids。

## 4. 为每个 centroid 创建子簇与子节点（并行）

```
for c ∈ centroids(v) do in parallel
    Pc ← k-means cluster of c
    create new tree-node vc
    add tree-edge from parent v to vc
```

- 含义：把当前簇  $P$  按刚才的 centroids 分成  $l$  个子簇  $P_c$ ；为每个  $P_c$  创建子树根  $v_c$ （即树的下一层节点）。并行化能大幅加速构造。
- 注意：这里并没有把原始点存放到叶子（与传统 k-means tree 不同）；我们的树仅用于生成代表 centroids。

## 5. 子树是否继续细化：分配子树预算并递归

```
if |Pc| > λ
    Build(Pc, l,  $\frac{(m-l)|P_c|}{|P|}$ , λ, vc)
```

- 解释预算公式  $(m - l) * |P_c| / |P|$ ：
  - 当前节点消耗了  $l$  个预算（centroids），所以剩余预算是  $m - l$ 。
  - 这剩余预算按子簇大小比例分配给每个子簇：大子簇拿到的预算多，能继续更细化；小子簇拿到的预算少，可能很快终止。
- $|P_c| > \lambda$ ：当子簇大小小于等于  $\lambda$  时停止对该子簇继续递归（不再细分）。

查询：

---

**Algorithm 2: KRT: Routing**

---

**Input:** Search budget  $b$ , query point  $q$

$PQ \leftarrow \{(v_i, 0, i) \mid i \in [s]\} // min-heap with (tree node, key, shard ID) prioritized by key$

$\text{min-dist}[i] \leftarrow \infty \quad \forall i \in [s]$

**while**  $PQ$  not empty and  $b-- > 0$  **do**

$(v, d_v, s_v) \leftarrow PQ.\text{pop}()$

**for**  $c \in \text{centroids}(v)$  **do**

$\text{min-dist}[s_v] \leftarrow \min(\text{min-dist}[s_v], d(q, c))$

**if**  $c$  has a sub-tree

| add  $(v_c, d(q, c), s_v)$  to  $PQ$

**return** sort  $[s]$  by  $\text{min-dist}$

---

核心目标是：给定一个查询点  $q$ ，快速找到最可能包含其近邻的 shards（子图）。下面我帮你详细拆解。

---

1. 数据结构和初始化

```
PQ ← {(vi, 0, i) | i ∈ [s]}
min-dist[i] ← ∞ ∀i ∈ [s]
```

- **PQ**: 最小堆 (min-heap)，存储 (tree node  $v$ , key, shard ID)
    - key = 距离 query  $q$  的距离，用于排序
    - heap 优先访问距离最小的节点 (beam-search 样式)
  - **min-dist[i]**: 记录每个 shard  $i$  到查询点  $q$  的最小距离，用于最后对 shards 排序
  - 初始化：把每个 shard 的根节点  $vi$  放入堆中，初始距离 0
- 

2. 主循环 (类似 beam-search)

**while**  $PQ$  not empty and  $b-- > 0$  **do**

$(v, d_v, s_v) \leftarrow PQ.\text{pop}()$

- 每次从堆中弹出距离最小的节点  $v$
  - $s_v$  表示  $v$  所在的 shard ID
  - $b$  是搜索预算 (限制总迭代次数或弹出节点数)，防止搜索过慢
- 

3. 处理当前节点

```

for  $c \in \text{centroids}(v)$  do
     $\text{min-dist}[s_v] \leftarrow \min(\text{min-dist}[s_v], d(q, c))$ 
    if  $c$  has a sub-tree
        add  $(v_c, d(q, c), s_v)$  to PQ

```

- 遍历当前节点 v 的所有 centroids (树节点代表的簇中心)
- 更新当前 shard 的最小距离:  $\text{min-dist}[s_v]$
- 如果这个 centroid 有子树 (非叶节点) , 把子节点加入堆继续搜索, key 为距离 q 的距离

**核心思想:** 只对最可能包含近邻的子树继续搜索, 避免全量遍历。

---

#### 4. 终止条件

- PQ 为空 → 搜索完所有可能节点
  - 搜索预算 b 用完 → 防止搜索开销过大
- 

#### 5. 输出

```
return sort [s] by min-dist
```

- 对所有 shards 按 min-dist 排序
- 返回最可能包含近邻的 shards 列表 (probe order)

#### 4.3.2 HRT Routing

**HRT (Hash Routing)** , 一种基于 **Locality Sensitive Hashing (LSH)** 的路由索引方法, 用于快速确定查询点 q 应该访问哪些 shards。下面我给你详细拆解。

---

#### 1. LSH 的基本概念

- LSH 是一类哈希函数族  $H$ , 用于高维近似最近邻搜索
- 特性:
  - 相似的点  $x, y \rightarrow$  很大概率落到同一个桶:  $Pr[h(x) = h(y)]$  高
  - 不相似的点  $x, y \rightarrow$  很小概率落到同一个桶

LSH 的作用是 **把高维点映射到哈希空间, 让近邻点容易聚在一起。**

---

#### 2. 构建 SortingLSH Index

#### 3. 采样点

- 从数据集 P 中均匀随机采样 m 个点  $R \subset P$

#### 4. 多次哈希

- 对每个点  $x \in R$ , 使用  $t$  个独立 LSH 函数  $h_1, \dots, h_t$
- 得到复合哈希串:  $(h_1(x), \dots, h_t(x))$

## 5. 排序

- 按照复合哈希串进行字典序排序

每个点的复合哈希串是:

$$(h_1(x), h_2(x), \dots, h_t(x))$$

其中每个  $h_i(x) \in \{0, 1\}$ , 所以整个串就是一串 0/1 组成的长度  $t$  的位串。

排序规则:

**按从左到右逐位比较, 遇到第一个不同的比大小 ( $0 < 1$ ) , 完全一样则平级。**

也就是:

- 把复合哈希串当作二进制字符串
- 或当作长度  $t$  的数组
- 按字典序排序即可

例子:

点	复合哈希串	排序顺序
A	0 1 0 0	第 1
B	0 1 1 0	第 2
C	1 0 0 1	第 3

- 相似点更可能有更长的前缀 → 排序后靠得更近

## 6. 重复构建

- 为提高召回率, 重复  $r$  次 (例如  $r=24$ ), 得到多个排序索引

**每重复一次, 就创建一个新的、完全独立的 LSH 索引:**

对每次重复  $i$ :

1. 用新的随机 hash 函数集合  
(即新的  $h_1, h_2, \dots, h_t$ )
2. 对  $m$  个代表点  $R$  计算新的复合 hash 串
3. lexicographically 排序
4. 存成  $Q_i$  (第  $i$  个 Sorted-LSH index)

最终得到:

```
Index_1: sorted by hash set #1
Index_2: sorted by hash set #2
...
Index_r: sorted by hash set #r
```

论文里  $r \approx 24$

也就是说你有**24 个不同的排序表**

## 查询时怎么用这些重复构造？

查询一个向量  $q$  时：

对每个  $index_i$ ：

1. 用该  $index$  的 hash 函数计算  $h(q)$
2. 找到  $h(q)$  在排序表里最接近的位置  $\tau$
3. 取窗口  $[\tau-W, \tau+W]$  作为候选

最后将所有  $index$  的候选合并，得到  $Q$ 。

## 直观理解：

排序后的列表类似“一维近邻序列”，查询点只需要查附近窗口，就能快速找到潜在近邻点所在的 shard。

---

### 3. Routing 查询流程

#### 4. 给定查询点 $q$

#### 5. 对每个 $r$ 次索引：

- 使用同样的  $t$  个 LSH 函数得到复合哈希串  $h(q)$
- 在排序列表中找到 **最接近  $h(q)$  的位置  $\tau$**
- 取窗口  $[\tau - W, \tau + W]$  中的点，计算实际距离，确定对应的 shards

**核心思想：**用 LSH 将 query 映射到排序位置，再在局部窗口搜索。

---

### 4. 理论保证

- HRT 的优点之一是可以提供 **形式化的理论保证**
  - 虽然路由只保证 **至少有一个近似最近邻所在的 shard 被访问**
  - 但在自然分区假设下，大多数近邻会和最近的点落在同一个 shard → 最终可以找到真实最近邻
- 

### 5. 总结对比 KRT

特性	KRT (k-means Tree Routing)	HRT (Hash Routing)
构建方式	分层 k-means 树	采样 + LSH 多次哈希 + 排序
数据结构	树	多个排序数组 (排序 LSH)
查询方式	beam-search 树遍历	哈希映射 + 排序窗口查找
理论保证	无明确理论保证 (实验效果好)	可证明至少路由到包含近似 NN 的 shard
优势	精度高，适合大 shard	构建快速，可提供理论保证

# Prefetching

## 1. Graph Reordering for Cache-Efficient Near Neighbor Search

摘要：

图搜索在近邻搜索中非常高效，但遍历图时容易出现缓存未命中，导致速度下降。作者通过图重排，将经常一起访问的节点在内存中靠近存放，从而优化内存布局，提高缓存命中率。实验表明，这种重排能将查询速度提升最多 40%，而重排本身耗时相比构建索引几乎可以忽略。

图搜索索引组成：

这部分主要在讲 图搜索类 ANN 索引（HNSW、PANNG、ONNG 等）内部都有哪些核心组件，以及它们为什么可以一并讨论。

核心思想：这些图索引虽然名字不同，但结构和搜索流程极其相似，所以图重排的效果可以推广到它们全部。

图搜索索引的三个核心组成

- 构图阶段的 **pruning / diversification**（剪枝和多样化）：把不必要的边删掉，把一些有用的长边补上，让图更“好走”。
- **搜索初始化**：给 beam search 找一个起点。可以是 HNSW 的上层结构、LSH、树结构或随机点。
- **Beam search 本身**：沿着图走，维护一个大小 M 的候选池，不断扩展更近的点。

作者强调：尽管不同算法细节不同，但访问节点的顺序和图结构很相似，所以图重排的适用性是通用的。

k-NN 图和  $\epsilon$ -NN 图的区别

- k-NN：每个节点连向距离最近的 k 个点（有向图）。
- $\epsilon$ -NN：距离小于  $\epsilon$  的节点全部连接（无向图）。

两者都可能产生不连通区域。

为什么真实近邻图不能直接构？因为太贵 ( $O(N^2)$ )

所以现代方法都用“bootstrapping”——边建边查：

1. 新点 q 查询目前的图（用近似搜索），得到近邻；
2. 更新这些邻居的边；
3. 最终成形的是 近似 k-NN 或 近似  $\epsilon$ -NN 图。

NN-Descent 是另一种常用方法（反复更新，几轮后接近真实图）。

现代最强的 ANN（如 HNSW 和 PANNG/ONNG）都不是用原始 k-NN 图，而是构图后进行剪枝与边补充：

- **HNSW**：分层结构，高层点更少，用于快速定位区域。底层是 pruned k-NN。其剪枝规则来自 Arya & Mount：如果邻居 B 比不上通过另一个邻居到达的路径，则删边。
- **PANNG / ONNG**：思想类似，基本都是“如果有更短路径就删掉长边”。ONNG 还会保证每个点的入度  $\geq k$ 。
- 其他如 FANNG，本质 prune 规则也差不多。

作者指出：虽然方法多，但因为剪枝原则接近，所以这些图的结构非常类似，这正是为什么重排效果可泛化。

## 搜索过程：本质是沿图步行

对每个访问的节点  $x$ , 计算  $d(q, x)$ , 然后从它的邻居中继续走, 看能否找到更近的点。直到  $M$  个候选都走不动, 就返回最好的几个。

**Beam search 是主要算法; greedy search 是它的  $M=1$  特例。**

---

## 初始化在搜索中占比很小, 而且不同初始化方式差别不明显

作者用 SIFT1M + HNSW 实验显示:

- 层级初始化只占 **约 3%** 的时间
- 把层级换成“随机选 50 个点取最近的”完全不影响效果

结论: 初始化方法对性能影响极小, 因此不会影响图重排实验的有效性。

## 1.1 图重排实现:

**把图的节点重新编号, 使得在内存中相邻的节点, 更可能在搜索时被连续访问, 从而降低 cache miss。**

---

### 一、图重排的目标是什么?

把每个节点  $v$  映射到一个新的编号  $P(v)$ , 并按照  $P(v)$  的顺序放入数组 = 内存布局。

目标是:

**如果两个节点经常一起访问, 就把它们的编号排得越接近越好。**

这样 CPU 访问内存时, 一旦把某段加载进 cache, 就能命中更多后续访问。

---

### 二、三类重排方法: 优化式、基于局部统计、基于图划分

论文把现有方法分成三大类。

---

#### (1) 优化式方法: 直接优化“标签接近 $\Leftrightarrow$ 关系强”。

这一类是论文最看重的, 也是实际效果最好的。

典型代表:

##### 1. Gorder

目标: 让连续的  $w$  个节点之间邻居尽量重叠。

含义: 把强相关节点放一起。

优点: 强, 但计算成本高 (NP-hard 的近似)。

##### 2. RCM

目标: 减少“两个相连节点的编号差”。

含义: 边尽量靠近对角线, 让图变“带宽小”。

优势: 老牌算法, 本质是 BFS 排序。

##### 3. MLOGA / MLINA

目标: 最小化所有边的编号差距 (总和或 log)。

含义: 尽量让所有边都“短”, 而不是只管最大差距。

共同特点:

**都是 NP-hard, 只能用启发式, 但能明显改善 cache 行为。**

---

## (2) 基于节点局部特征的方法：如 Degree Sorting

做法：按节点度从大到小排序。

直觉：

高度节点常常连接彼此（尤其在 power-law 图）。

所以排在前面的高程度节点很可能一起被访问。

但对于 ANNS 图（如 k-NN 图）：

- 度数都差不多（每个节点 k 条边）
- 不呈 power-law 分布

→ 这种方法在 ANNS 上基本无效。

---

## (3) 基于图划分的方法：把每个 partition 放一段连续内存

思想：

图划分算法会把 tightly-connected 的节点放到同一区域。

我们可以给每个 partition 一个连续的 label 区间。

缺点：

分区结果可能不适合 ANNS 的访问模式。

---

## 三、为什么优化式方法明显更好？

因为 ANNS 图的访问模式是：

- 查询时沿着图进行 local walk (beam search)
- 访问的是“局部邻域 + 一些分层入口节点”
- 每次跳转都依赖邻居布局

因此最关键的是：

**把邻接的节点尽可能放在相邻的内存地址。**

优化式方法正是直接围绕这个目标设计的，而 degree-sorting 之类的根本不关注“邻居访问顺序”。

这就是为什么论文说：

轻量算法在 ANNS 里没用，objective-based 才有效。

---

## 四、为什么重排不会让构建成本爆炸？

尽管这些 objective-based 算法看起来贵，但：

- 重排图的成本通常是  $O(E \log N)$  或  $O(E)$  级别
- 构建图（如 HNSW）成本本来就非常高
- 因此重排耗时反而比构建图小一个数量级

论文说：

**重排 << 建图成本**

**而加速可达 10-40%**

工业上很划算。

## 五、为什么 ANNS 图与社交网络不同？

大部分重排算法最初用于 social graph，这些图有：

- power-law degree (少数超级节点)
- 稀疏结构
- 社区结构明显

但 ANNS 图 (如 k-NN)：

- 度均匀，不是 power-law
- 较密
- 空间结构驱动，而不是社交结构

因此：

很多社交图能用的重排技巧，在 ANNS 上不一定有效。

## 实验设计

- 使用 HNSW 索引，对大规模 embedding 数据集 (SIFT100M/1B、DEEP100M/1B) 进行实验。
- 指标包括 平均查询时间、P99 延迟、缓存未命中率、top-100 recall。
- 控制单核运行、warm start、固定查询集合等外部因素。

## 实验结果

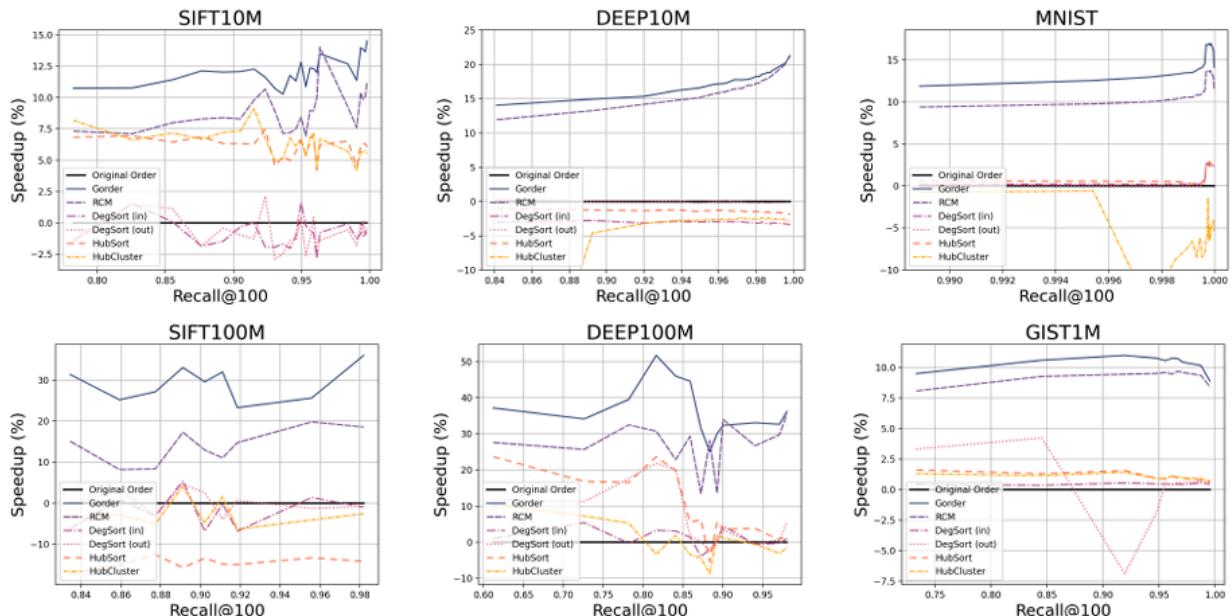


Figure 4. Effect of graph reordering on the query time vs. recall tradeoff. Reordering algorithms above the black line have faster query time than the original ordering. Algorithms below the line cause a slowdown. The speedup changes with the recall because the memory access pattern of beam search changes with the buffer size.

- Objective-based 重排 (Gorder/RCM) 可显著提高查询速度：小数据集 10% 提升，大数据集高 recall 下可达 40%。
- 重排时间远小于索引构建时间，长期查询收益远超一次性成本。
- 缓存命中率显著提高，降低 L1/L2/L3/TLB miss。

## 2. Graph Prefetching Using Data Structure Knowledge

### 1. 背景与问题

- 内存受限的工作负载 (memory-bound workloads) 通常通过 **预取 (prefetching)** 提升吞吐量：硬件根据访问模式预测未来内存访问，将数据提前加载到高速缓存。
- 对于图计算，传统硬件预取器效果很差，因为图访问模式是 **数据依赖 (data-dependent)** 且非连续的：
  - Stride prefetcher** 无法处理非规律的访问序列。
  - Correlation-based prefetcher** 存储开销大，且对一次性计算提升有限。
  - Pointer prefetcher** 无法有效预测间接数组访问。
- 类似 MapReduce 的框架对图计算也不高效，因为图计算通常 **不可天然并行**，迭代次数多，开销大。

### 2. 论文的核心观察

- 尽管硬件无法自动捕捉图结构，但在常见图应用中，**访问模式其实是可预测的**，例如 BFS、DFS、Best-First Search 等遍历模式。
- 因此，可以使用 **显式预取器 (explicit prefetcher)**：硬件预取器利用对数据结构和遍历方式的知识，提前知道需要哪些数据，减少学习访问模式的开销，并能提前做出预取决策。

### 3. 方法

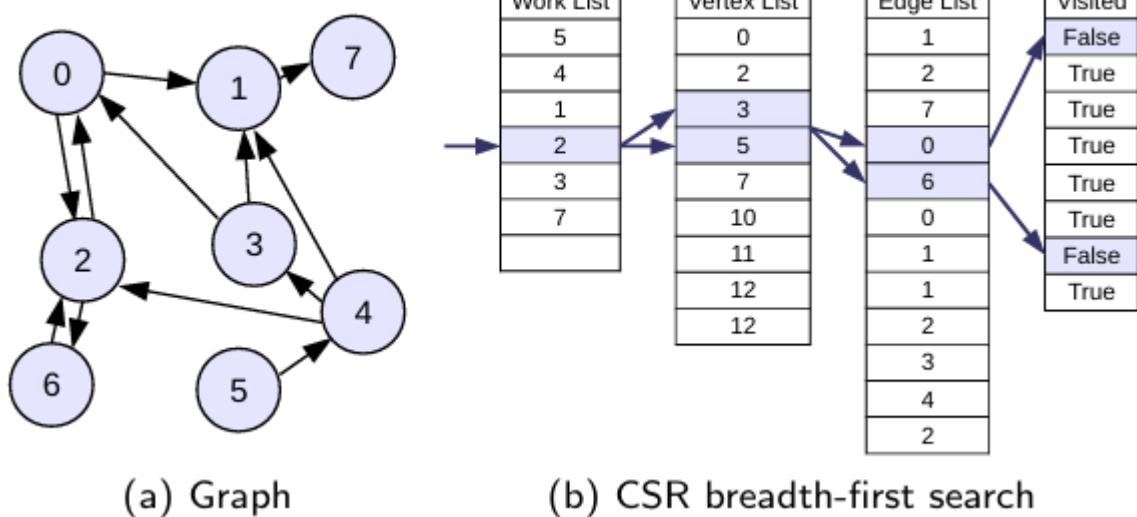
- 论文针对 **压缩稀疏行 (Compressed Sparse Row, CSR) 格式图**，设计了一个硬件预取器，专门优化 BFS 遍历。
- BFS 是社交网络分析、网页抓取、模型检测等领域常用的计算核，很多算法都基于 BFS。
- CSR 是稀疏图的标准存储格式，存储高效、被广泛采用；相比其他表示方法（如 GraphLab），CSR 减少了额外开销（如数据复制）。
- 预取器通过 **监控内存队列中的读取操作 (snoops reads)**，计算并调度边和顶点数据提前加载到 L1 缓存，从而保证数据在访问时已在高速缓存中。

相关方法介绍：

#### 第一段：Compressed Sparse Row (CSR) 数据结构

这段说明了 **CSR 是一种非常高效的稀疏矩阵/图存储格式**，常用于图处理，因为它既节省空间又性能好。

CSR 由两个核心数组组成：



### 1. edge list (边数组)

- 一维数组，顺序存储图中所有顶点的所有邻居。（从0开始，其邻居为1, 2；然后到1，其邻居为7；然后到2，其邻居为0, 6）

### 2. vertex list (顶点数组)

- 记录每个顶点的邻居在 edge list 中的起始位置。（从0开始，其邻居在edge list中从0开始；然后到1，其邻居从2开始；然后到3，其邻居从3开始）

CSR 图结构不包含指针，只有索引，因此：

- 更紧凑
- 更适合 CPU 缓存和内存顺序访问

论文图 1(a) 是一个示例图，而图 1(b) 则展示它的 CSR 两个数组。

## 第二段：关于 BFS 的背景

BFS (Breadth-First Search, 广度优先搜索) 是图计算中最常用、最重要的访问模式之一，广泛用于：

论文指出：

**由于 BFS 如此常见，优化 BFS 的内存访问对整个图计算性能具有巨大意义。**

## 第三段：BFS 的执行过程 (Algorithm 1与 Figure 1(b))

论文描述 BFS 工作方式：

```

1 Queue workList = {startNode}
2 Array visited[startNode] = true
3 while worklist ≠ ∅ do
4   Node N = workList.dequeue()
5   foreach Edge E ∈ N do
6     if visited[E.to] is false then
7       workList.enqueue(E.to)
8       visited[E.to] = true
9     end
10   end
11 end

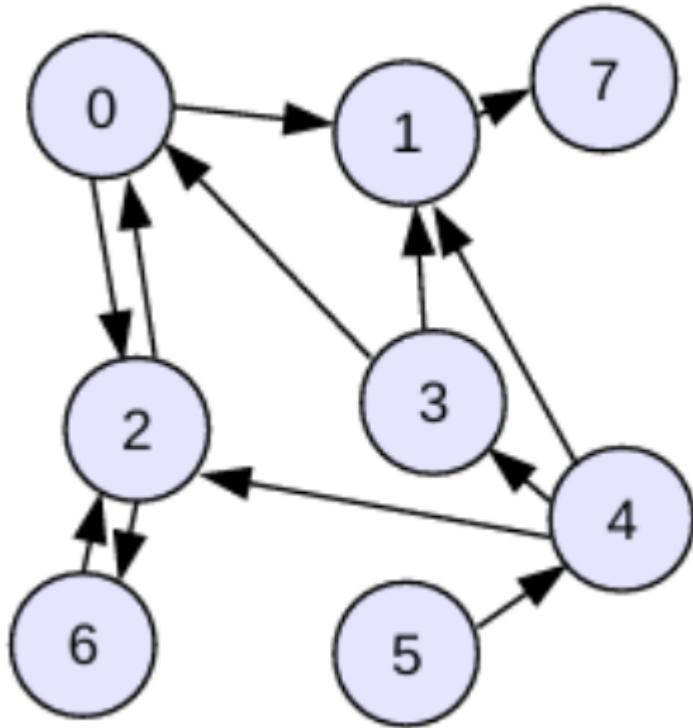
```

### Algorithm 1: Breadth-first search.

1. 从起始节点开始。
2. 将其加入一个 FIFO 队列 (work list queue) 。
3. 每次从队列取一个顶点，通过 CSR：
  - 用 vertex list 找到它的邻居起点
  - 用 edge list 遍历它的所有邻居
4. 每访问到一个新的邻居，就将它加入队列。

论文给了一个例子：

从顶点 5 开始遍历，访问顺序为： **(work-list)**



$5 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 0 \rightarrow 6$  (按照顺序来, 5->4; 然后遍历4的邻居, 1, 2, 3; 然后是1的邻居7; 然后是2的邻居0, 6)

这符合 BFS 的“逐层”访问方式。

BFS的缺陷:

**Stalling Behavior (停顿行为分析)** 的逐句清晰解释, 让你更容易理解论文为什么强调 BFS 的访存瓶颈。

## 1. BFS 的核心问题: 访存局部性极差

论文开头强调:

- **vertex list (顶点数组)** 访问没有时间局部性或空间局部性  
→ 每次访问不同的顶点, 跳得很随机, 连续性差。
- **edge list (边数组)** 只有“单个顶点内部”有局部性  
→ 访问一个顶点时会顺序读它的所有邻居, 但访问下一个顶点就跳走了。

因此:

一旦图比最后一级缓存 (LLC) 大, 缓存几乎帮不上忙 → 性能由内存延迟主导。

## 2. 实测: BFS 会造成极高的 stall (停顿)

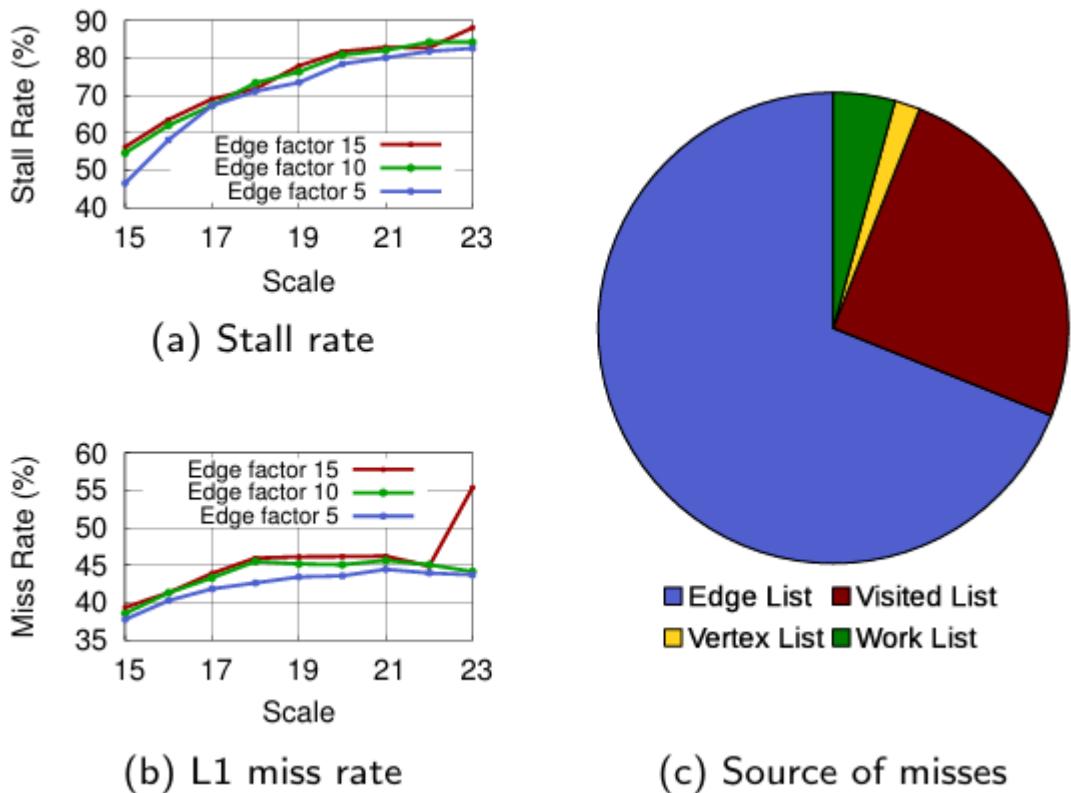
论文引用 Graph500 (图搜索标准 benchmark), 在 Intel Core i5-4570 上实验:

- **stall rate 接近 90%**  
→ CPU 有 90% 的时间都在等待数据从内存返回 (停顿), 不是在计算。
- **随着图规模变大, stall 率还在继续上升。**

这是图计算性能低下的核心原因。

### 3. 为什么 stall rate 这么高？因为 L1 cache miss 非常高

图 2(b) 显示：



- L1 cache miss rate 接近 50%

对 CPU 来说，L1 miss 非常昂贵，会导致 pipeline 停顿，一路等到 L2/L3 或内存返回。

BFS 的访问模式导致 cache 几乎无法命中，因为访问是随机散布的。

### 4. 更精细的数据（图 2(c)，gem5 仿真结果）

他们用 gem5 模拟器深入分析 miss 延迟来源 (scale 16, edge factor 10) :

- 69% 的额外时间来自 edge list misses
  - edge list 非常大 (edge list  $\approx$  vertex list 的 20 倍)，且访问跳跃大。
- 25% 来自 “visited” 数组 misses
  - visited[v] 标记顶点是否访问过
  - 类似 vertex list，访问顺序随机，每次访问几乎都是 L1 miss。

尽管 visited 的大小与 vertex list 相同，但：

- 每条边访问一次 visited
- 次数非常多 (几千万次)
- 且顺序完全随机
  - 导致大量 cache miss

这为后面提出的“硬件预取器”提供了直接动机：

- 如果能提前把 edge list / vertex list / visited 数据拉进 L1 cache

- 就能显著减少 stall
- 提升 BFS 性能 (论文中加速 2.3×-3.3×)

传统预取技术的局限性：

## 第一部分：Stride Prefetching (步幅预取) 为什么失效

现代 CPU 最常用的预取器是 **stride-based** (按固定步长推测未来访问)。

它的假设：

下一次访问地址  $\approx$  上一次地址 + 固定步长

这种模式适用于：

- 顺序扫描数组
- 遍历矩阵
- 连续内存结构

但 BFS 访问模式是：

- 每个顶点的邻居列表虽然连续
- 顶点之间跳跃极大且不可预测 (数据依赖)

因此：

没有可学习的 stride，预取器形同虚设。

---

## 第二部分：为什么不能预取 visited list (访问标记数组)？

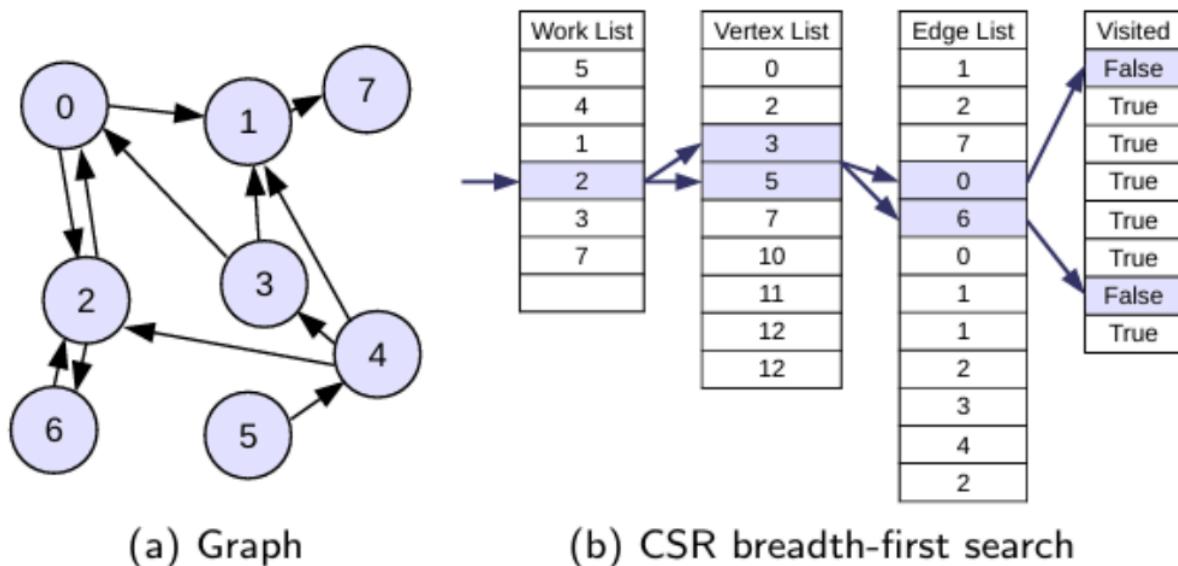
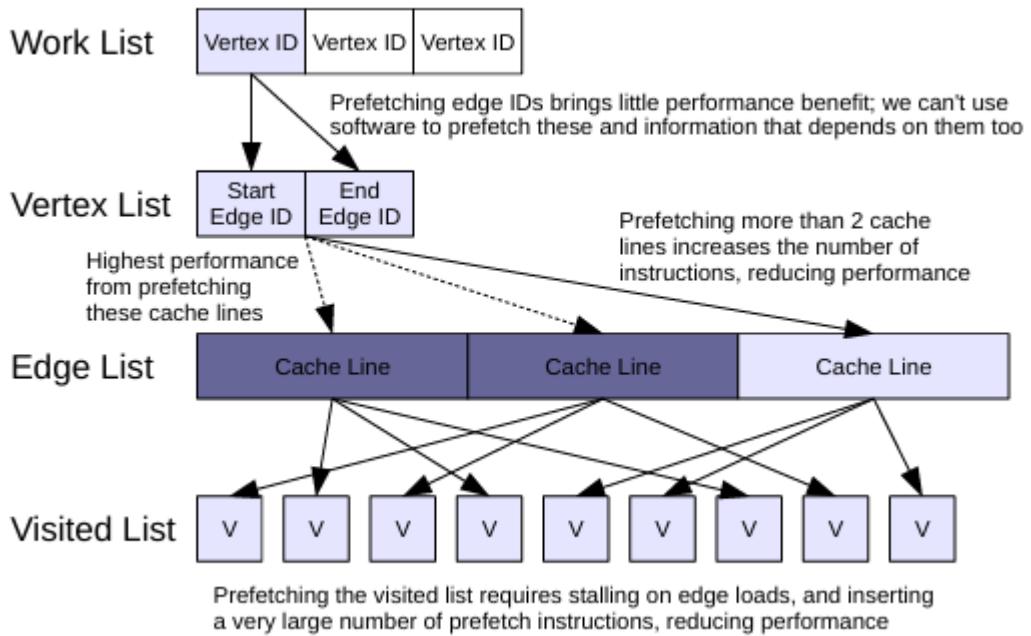
作者指出：

- visited[v] 属于随机访问
- 想预取 visited list，就必须先知道 v 的值  
然而 v 的值本身来自 edge list 的读取

意味着：

预取 visited 需要等待 edge list 加载 → 预取本身要 stall → 失去意义

图中示意：



- visited list 分布在多个 cache line (**内存不是按字节读的，而是按块读的**: 当 CPU 访问某个地址时，内存控制器会把“该地址所在的 64 字节块”整块加载进缓存，这个块就是 **cache line**)

### visited list 是什么？

通常是一个布尔数组：

```
visited[0], visited[1], visited[2], ..., visited[N-1]
```

N = 顶点数 (可能是百万、千万、甚至上亿级)

即使每个 visited 只占 1 byte 或 1 bit, 总大小也很容易超过:

- L1 cache (32 KB)
- L2 cache (256 KB)
- 甚至 L3 cache (几 MB)

## 原因 1：数组太大，必然跨多个 cache line

假设：

- cache line = 64 byte
- visited 数组 = 10M 个顶点（常见规模）

即使每个 visited 用 1 byte 存：

- 总大小 = 10 MB
- 需要数量 =  $10\text{MB} / 64\text{B} = \text{约 } 156,000 \text{ 个 cache line}$

所以显然会跨很多 cache line。

- BFS 会随机访问其中的单个标记
- 大量随机 prefetch 会把 CPU 流水线压爆，反而变慢

---

## 第三部分：Software Prefetching (软件预取) 也不好用

软件预取 = 程序员在代码里手动插入 prefetch 指令。

理论上 BFS 的访问模式已知，程序员可以：

- 预取 vertex list
- 预取 edge list
- 预取 visited list

但论文分析了实际问题：

---

### 问题 1：软件预取不能依赖前一个预取的结果

例如：

```
prefetch(vertex_list[work[i]])
```

要执行 prefetch，你要先读取 work[i] → 这就会 stall。

因此：

**任何软件预取都会引入额外 load → 造成 stall → 抵消预取的收益**

---

### 问题 2：预取距离 (prefetch distance) 是动态变化的

- BFS 队列的长度不断变化
- 不同图的度分布也不同

但软件预取必须设置 **固定距离**：

- 距离小 → 太晚，来不及
- 距离大 → 太早，会被 eviction 掉

软件无法动态适应 workload 行为。

---

### 问题 3：预取 edge list 量巨大，会产生大量指令

每个顶点有很多邻居：

- 若你对每个 neighbor 都 prefetch → 指令爆炸
- 指令开销 > 预取收益

所以不可行。

---

### 论文结论：最佳的软件预取策略仍然很有限

作者发现：

- 最好的策略是：  
在 BFS 算法中的第 4 与第 5 行之间  
对 **未来某个顶点的前两条 edge-list cache line** 做 prefetch

但是：

- 调不同 offset、不同 cacheline 数目都无提升
- 尝试预取 vertex list、work list 都让性能下降

最终结果 (Figure 3(b)) :

- 软件预取 + 硬件预取可提升约 **35% 性能**
- 但 CPU 仍然有 **80% 的时间在 stall**

说明：

**传统预取技术远远不够，依旧留下大量性能空间。**

## 2.1 Graph Prefetcher 设计与工作原理：

---

### 1. 预取器的总体思路

- 目标：BFS 遍历 **CSR 格式图**
- 方法：预取器 **监控 (snoop) CPU 和自身发出的缓存访问**，根据访问情况发起新的预取请求
- 放置位置：紧挨 **L1 cache**，而不是 L2，因为 L1 预取能最大化减少 miss 延迟
- 支持虚拟地址访问，需要 DTLB (虚拟地址到物理地址转换)

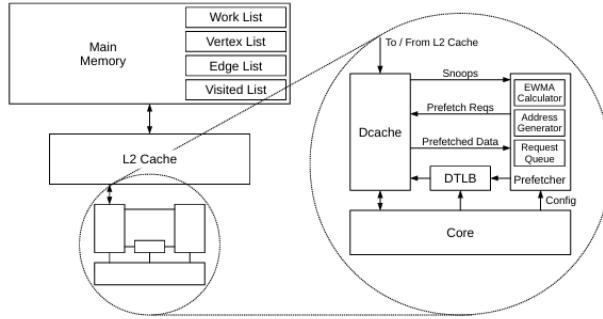
核心数据结构：

- **Edge list** (边数组)
- **Vertex list** (顶点数组)
- **Visited list** (访问标记数组)
- **Work list** (待访问队列)

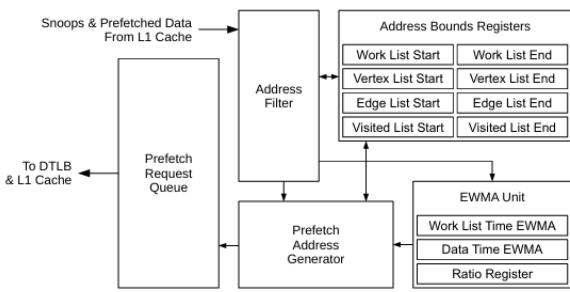
预取器知道它们的地址范围，能把 **索引 → 地址** 转换，从而发起针对性预取。

---

## 2. 基本操作 (Basic Operation)



(a) System overview



(b) Prefetcher microarchitecture detail

假设 CPU 正在处理 `workList[n]` :

### 1. 计算预取目标顶点

- 目标顶点 = `workList[n + o]`
- $o = \text{offset}$ , 表示“提前多少步”的预取距离（基于 fetch 和 traversal 延迟比例来选择）

### 2. 依次预取相关数据

对每个目标顶点  $v$ :

- Step 1: 预取 `workList[n+o]` → 得到 vertex ID  $v$
- Step 2: 预取 `vertexList[v]` → 得到该顶点的 edge list 起始索引
  - `vertexList[v+1]` 通常在同一 cache line, 如果不在, 就假设该顶点对应 2 个 cache line
- Step 3: 对每条边  $e$ , 预取 `edgeList[e]` → 得到邻居顶点 ID
  - 邻居 ID 也用作 visited list 的索引
- Step 4: 预取 `visited[v]`

### 3. 链式依赖触发

- CPU 访问 `workList[n]` → 预取器触发
- 依赖关系链条生成新的预取请求
- 当 L1 cache 的 MSHR 可用时发出预取
- 预取器 snoop 总线, 检测数据返回 → 再计算 `vertexList`、`edgeList`、`visited list` 的地址 → 发起下一轮预取  
(预取器“看”CPU访问什么 → 等数据回来 → 根据它计算下一步要预取的所有数据 → 发起新的预取”, 这样保证数据提前到达 cache, 减少 CPU stall)

**Graph Prefetcher 如何调度 (Scheduling) 预取**, 也就是 **决定预取哪些数据、什么时候预取**。我帮你梳理成逻辑清晰的理解:

### 1. 核心目标: just-in-time 预取

理想状态:

```
work list time = data time
```

- **work list time** = CPU 处理当前顶点所需的平均时间

- **data time** = CPU 从内存加载该顶点所有相关数据所需的时间

如果两者匹配：

- 预取的数据在 CPU 需要之前刚好到达 L1 cache (这样可以直接无缝把数据提供给CPU处理)
- 类似 Mowry 等人的静态编译器方法，但这里是 **动态实现**，根据运行时情况调整。

## 2. 动态估算时间：EWMA (指数加权移动平均)

由于图顶点度数不均、处理时间波动大：

- 预取器使用 **EWMA** 动态估算 work list time 和 data time
- 公式：

$$\text{avg\_time\_new} = \alpha * \text{new\_time} + (1 - \alpha) * \text{avg\_time\_old}$$

- $\alpha = 8$  → 用于 work list time
- $\alpha = 16$  → 用于 data time (更稳定，避免偶然边导致过低估计)

效果：适应不同图规模和顶点度数差异。

## 3. Vertex-Offset Mode (顶点偏移模式)

条件：`data time > work list time`

- 意味着加载一个顶点数据比 CPU 处理当前顶点还慢
- 解决方法：提前几个顶点发起预取
- 计算偏移量公式：

$$o = 1 + k * \text{data\_time} / \text{work\_list\_time}$$

- $k = 2$  (论文模拟结果)
- 意图：确保数据在使用前到达 L1 cache
- 这种模式针对 **小顶点处理快、数据量大** 的情况

## 4. Large-Vertex Mode (大顶点模式)

条件：`data time < work list time`

- 当前顶点处理时间长，加载下一个顶点的数据很快
- 问题：如果只预取 offset=1，可能 **数据在 cache 中被替换掉**
- 解决策略：
  - 基于当前顶点 edge list 的处理进度
  - 连续预取 21 个 cache line
  - 然后以 stride 14 进行后续预取：

```
firstLine = edgeList[idx + 14*lineSize]
```

- idx = 当前 edge list 索引
- lineSize = cache line 大小
- 对于下一个顶点：当距离当前顶点 edge list 尾部 4 cache line 时，发起预取
- 这种方式 **fetch distance 保持小且保守**，避免数据过早被淘汰

## 5. 总结调度策略

- **目标**: CPU 访问数据时已经在 L1 cache
- **动态适应**: 根据运行时 work list time 和 data time 调整偏移量
- **两种模式**:
  1. Vertex-Offset Mode → 数据加载慢，提前若干顶点发起预取
  2. Large-Vertex Mode → 当前顶点处理慢，边块预取 + 小步长 stride

## Graph Prefetcher 的具体实现细节 (Implementation) :

### 1. 预取器用状态机实现

- 预取器有两种操作模式 (section 3.2 的 Vertex-Offset Mode 和 Large-Vertex Mode)
- 实现方式是 **多个有限状态机 (FSM)**
  - 每个 FSM 对 L1 cache 的活动作出反应
  - 根据不同事件触发不同的预取动作
- **Table 1** (论文中) 列出了：

Vertex-Offset Mode

Observation	Action
Load from workList[n]	Prefetch workList[n+o]
Prefetch vid = workList[n]	Prefetch vertexList[vid]
Prefetch from vertexList[vid]	Prefetch edgeList[vertexList[vid]] to edgeList[vertexList[vid+1]] (12 lines max)
Prefetch vid = edgeList[eid]	Prefetch visited[vid]

Large-Vertex Mode

Observation	Action
Prefetch vid = workList[n]	Prefetch vertexList[vid]
Prefetch eid = vertexList[vid]	Prefetch edgeList[eid] to edgeList[eid + 8*lineSize - 1]
Load from edgeList[eid] where (eid % (4*lineSize)) == 0	Prefetch edgeList[eid + 4*lineSize] to edgeList[eid + 8*lineSize - 1]
Prefetch vid = edgeList[eid]	Prefetch visited[vid]
Prefetch edgeList[vertexList[vid+1]]	Prefetch workList[n+1]

- 预取器观察的事件 (例如 CPU load / prefetch)

## 现代 CPU 自带多级预取器

- L1、L2 都有硬件预取器

- 会自动根据访问历史或 stride 模式提前加载数据到缓存

### CPU 预取属于普通 load 的扩展

- CPU 发出的预取会产生内存访问，总线上的动作和普通 load 类似
- Graph Prefetcher 监听总线 (snoop)，能捕捉到这些事件

### 利用 CPU 预取提高效率

- 看到 CPU 预取的数据返回后，Graph Prefetcher 可以立刻利用这些信息去计算下一步要预取的 vertex/edge/visited
- 对应触发的动作（例如预取 vertex list / edge list / visited list）

---

## 2. 配置 (Configuration)

- 预取器不能自动学习每个列表的内存地址范围（太复杂）
- **应用程序必须显式告诉预取器**: work list、vertex list、edge list、visited list 的地址范围
- 优点：功能可封装到库里，用户只需调用库即可
- 缺点：需要重新编译，但对高性能应用影响不大

---

## 3. 运行机制 (Operation)

- 每次预取器观察到一个 load 或 prefetch 地址：
  1. 检查该地址是否属于某个列表
  2. 如果是 → 发起相应预取，加载将来会用的数据

举例：

- Vertex-Offset Mode: CPU 访问 `workList[n]` → 触发预取 `workList[n+o]`，并预取该顶点相关的 vertex list、edge list
- 如果预取器观察到 **work list 的预取**，就可以直接读数据，然后继续预取 vertex list

---

## 4. 简化假设: vertex list 索引在同一 cache line

- 预取器假设 `vertexList[v]` 和 `vertexList[v+1]` 在同一 cache line
- 优点：状态机设计简单，不用同时处理跨多 cache line 的情况
- 缺点：如果实际索引跨 cache line，预取器能力会降低
  - 解决方法：假设顶点对应 edge list 在 2 个 cache line 内
  - 如果进入 Large-Vertex Mode → 在应用程序真正加载数据后再修正

---

### 一句话总结：

Graph Prefetcher 是一组有限状态机实现的动态预取器，需要预先配置数据结构地址范围，通过 snoop L1 cache 事件触发链式预取，简化假设 (vertex list 同 cache line) 降低 FSM 复杂度，同时在特殊情况下进行修正。

## 硬件结构与资源需求：

- **核心结构 (5 个) :**
    1. **Address Filter:** 检查 snoop/预取地址是否属于 4 个列表 (work/vertex/edge/visited)
    2. **Prefetch Address Generator:** 生成最多 2 个预取地址 (使用 2 个加法器)
    3. **EWMA Unit:** 更新 work list 和 data 时间的指数加权移动平均 (2 个加法器 + 2 个乘法器)
    4. **Ratio Register:** 存储比例，用于偏移计算 (使用一个除法器)
    5. **Prefetch Request Queue:** 存储待发出的预取请求 (200 条 64-bit entry)
  - **存储需求:** 约 1.6 KB (比 stride prefetcher 少，也比历史/Markov 基预取器小得多)
  - **计算单元:** 4 个加法器、2 个乘法器、1 个除法器
- 

## 2. 可扩展性：支持并行和不同访问模式

- **并行 BFS:**
  - 多核心共享图数据，每个线程用自己的预取器
  - FIFO work list 换成“bag”，每线程访问独立子队列
  - 多线程仍可按顺序访问，实现链式预取
- **顺序迭代访问 (Sequential Iteration) :**
  - 例如 PageRank 遍历 vertex/edge
  - CPU 访问边列表时，预取器按 large-vertex mode 逻辑加载依赖的 vertex-indexed 数据
- **其他访问模式:**
  - 例如 best-first search，可监听二叉堆 array 的访问
  - 其他数据格式 (hash-indexed array) 也可用类似方法，只需调整地址生成逻辑
  - 核心硬件 (prefetch request queue) 可复用

## 3. Adaptive Granularity Based Last-Level Cache Prefetching Method with eDRAMPrefetch Buffer for Graph Processing Applications

### Introduction部分：

1. **问题背景:** 主存延迟是现代计算系统性能瓶颈，尤其是大数据和 AI 应用中，由于内存访问模式高度不规律，传统预取方法难以准确预测，容易造成缓存污染。
2. **现有方法局限:** L1/L2 缓存容量小，低级缓存预取容易误触发；而 LLC (最后一级缓存) 虽然容量大，但多核环境下访问模式极不规律，准确预取更具挑战。
3. **本文贡献:** 提出了一种 **自适应粒度的 LLC 预取方法**，使用三种预取粒度：
  - **Cache line 粒度**
  - **Correlated line 粒度** (8条 cache line)
  - **Page 粒度**通过三个模块分别分析内存访问流，实现保守和激进的预取策略平衡**精度与覆盖率**。

### 4. 实现方式：

- 使用 **eDRAM 预取缓冲区**，位于 LLC 与主存之间，减小内存访问延迟。
- 模拟器基于 Intel Pin 框架生成内存 trace。

## 5. 实验结果：

- 执行时间平均提升约 15% (对比 GHB、BO 预取器)
- 能耗降低约 18% (对比 GHB、BO)
- 相对于 Markov 预取器和 VLDP，性能和能效也有小幅提升。

### Background and Related Work:

#### 缓存与内存预取基础

- Stride-based 预取：记录最近的地址增量 (delta)，若模式重复则触发预取。
- Markov predictor 预取：用历史表分析地址间相关性，适合处理时间不规则模式，但需要大存储。
- Locality-based 预取 (FDP)：根据预取精度、及时性和缓存污染程度控制保守/激进策略，单核环境下有效，但检测固定 stride 和固定反馈机制有限。

#### 先进的预取方法

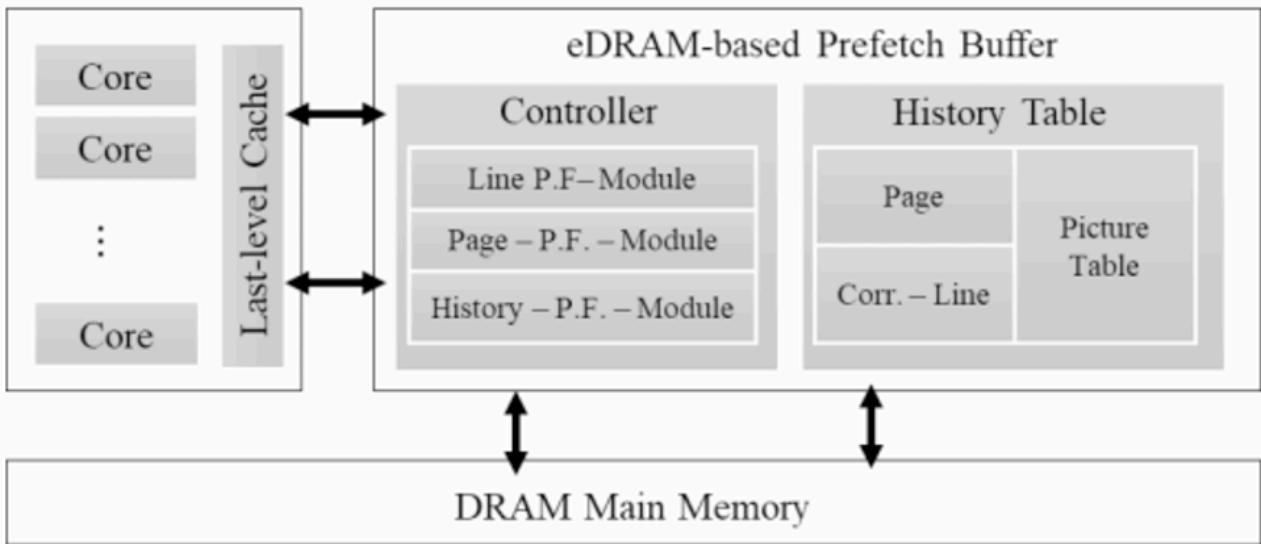
- **GHB (Global History Buffer)**：通过索引表 + 全局历史缓冲记录最近访问地址，FIFO 管理历史表，用最近请求触发预取。
- **VLDP (Variable Length Delta Prefetcher)**：使用 delta 历史表，匹配最近访问流的 delta 链，多层表寻找最匹配链条，实现更精准的预测。
- **组合方法** (Kondguli 等)：三个模块分别处理 stride 流、指针遍历和高空间局部性流，由控制器选择最适合的模块进行预取。
- 商用处理器的硬件预取器：简单 stride、next line、adjacent line；Bingo 空间数据预取器与 Domino 时间数据预取器优化了覆盖率与精度。
- **领域专用预取器**：例如针对图处理的 L1-L2 预取，利用图数据结构的顶点/边历史优化地址边界预取。

#### 问题与挑战

- 现代大数据和 AI 应用需要大工作集和高通用性预取器，传统方法难以覆盖大容量主存访问且复杂性受限。

### Methodology: Proposed Prefetcher

#### 总体架构



**Figure 1.** Overall Architecture.

- eDRAM 预取缓冲区位于 LLC 与主存之间，利用自适应预取粒度预测不规则内存流。
- 三种粒度分析内存访问：cache line (64 B)、correlated line (8 个 cache line)、page (4 KB)。
- History table + Prefetch engine：管理不规则访问，权衡预取精度与覆盖率。

### History Table (历史表)

- 由 Page Table、Correlated Line Table、Picture Table 组成。
- **Page Table**: 64 条记录，按页存储访问地址，限制 delta 范围 [-63, +63]，使用 LRU 替换。
- **Correlated Line Table**: 每页 8 个 correlated line，每个 correlated line 8 个 cache block，用于检测局部访问模式；当页表被替换，相关行表也清空。
- **Picture Table**: 64 条记录 (32 页粒度 + 32 correlated line 粒度)，抽象表示访问模式；只有当同一页或同一 correlated line 被访问  $\geq 4$  次时更新，用于触发预取。

### 3.1 工作机制：

**History Table = Page Table + Correlated Line Table + Picture Table**

你可以把整个 History Table 想成一个“预取记忆系统”，由三层结构组成：

#### 1. Page Table: 存“访问了哪些页”

用来回答：

“我最近访问过哪些 4KB 的页面？”

结构很简单：

- 一共有 64 个 entry
- 每个 entry 存一项 Page\_Adr (页面地址 = 原地址  $\gg 12$ )
- 同时还挂着这个页面的 CL table (下一层)

为什么只存页面地址？

- 因为这样连续两次访问的 delta 就变成 -63 ~ +63 (只看页内偏移)

作用：

- ▶ 判断两个地址是否在同一页
- ▶ 给后面的 CL Table 提供基础窗口

## 2. Correlated Line Table (CL Table) : 存“一个页里访问了哪些 8-line 小区间”

用来回答：

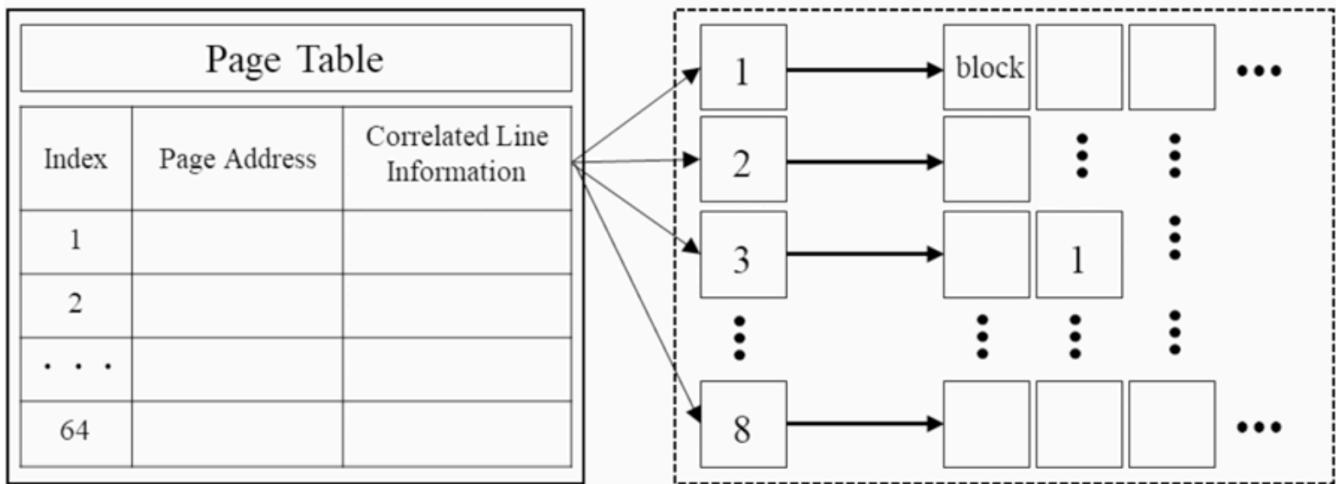
“在同一个页面里，我访问的是哪一段段 8 个 cache lines 组成的小区间？”

页面大小 4KB → 共有 64 个 cache line

论文把每 8 个 line 合成 1 个 correlated line (CL)

所以：

一个页共有 8 个 correlated line



每个 CLTable entry 记录：

- CL 编号 (0~7)
- 访问过的 cache line index
- 访问顺序 / 时间戳
- 访问次数
- 连续访问 delta (如 +3, -2, +1 等)

作用：

- ▶ 把乱七八糟的 irregular 访问行为放进更大粒度 (8行) 里观察规律
- ▶ 判断这个 CL 是否有稳定的 delta 模式

通俗比喻：

- page table 是“我进入哪一栋楼”
- CL table 是“我在这栋楼里走过哪些楼层区间 (每 8 层一组)”

## 3. Picture Table: 存“某个 CL 的访问图案”

用来回答：

“这个 correlated line 的访问模式到底是什么形状？是否可预测？”

作用非常关键：

- 当某个 CL 的访问次数  $\geq 4$
- 才把它的访问 pattern 抽象成图案（类似压缩的路径图）存入 picture table
- 分为两类（各 32 entries）：

类型	数量	用来记录什么？
Page-level picture table	32 entries	整个 page 内的访问 pattern
Correlated-line-level picture table	32 entries	某个 CL 内更细粒度的 pattern

### Picture Table 存的是什么？

关键来了：

它存的是 **访问的 block index 的相对位置变化** (delta 图案)。

例如某 CL 里访问序列为：

```
block 5 → 7 → 6 → 4
```

将它转成：

```
第一个 block 当作 0  
后面的减去第一个 block
```

就变成：

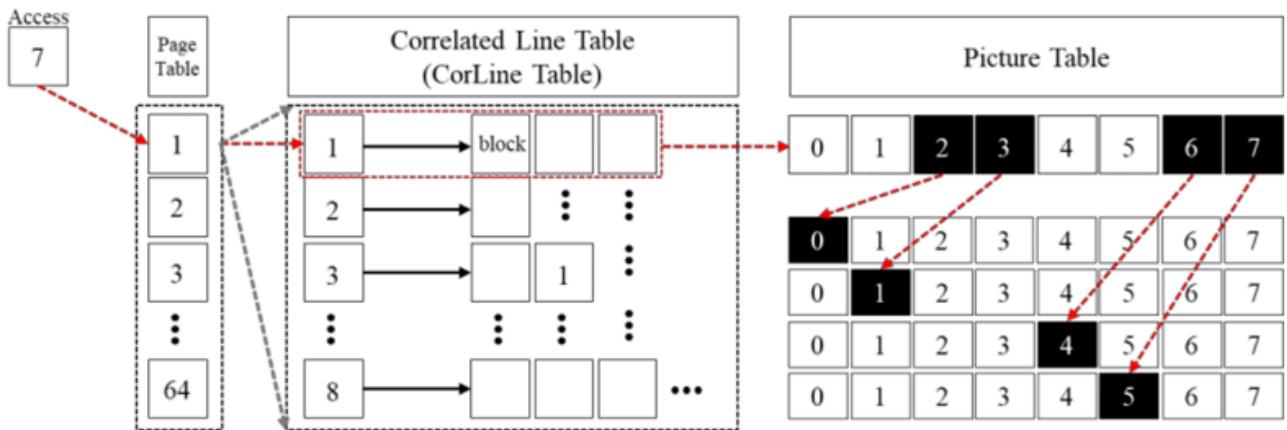
```
0 → +2 → +1 → -1
```

然后再做进一步压缩：

- 全部变成 0~7 范围（模 8）
- 存成小小的 3-bit 数字序列

于是一个访问序列能被压缩成类似：

```
[0, 2, 1, 7]
```



**Figure 3.** Update flow of picture table.

三个表的关系

一次访问到来时的流程：

1. **Page Table**: 看这个地址属于哪一页
  2. **CL Table**: 更新该页的 correlated line 信息
  3. **Picture Table**: 当某个 CL 的访问够多 ( $\geq 4$  次) 就升级成图案
  4. **Prefetch Engine**: 用 picture table 的图案预测下一步的位置, 放进 eDRAM prefetch buffer

一句话总结：

- ▶ Page Table 管“在哪栋楼”
  - ▶ CL Table 管“在楼里的哪些区间移动”
  - ▶ Picture Table 管“这些移动的规律是什么”

### 3.2 Prefetch Module:

## Prefetch Engine 的核心思想

它做的事其实只有一句话：

- 用 3 个不同“粒度”的预测器，从“非常准确但覆盖低”到“非常激进但覆盖高”，依次尝试预测未来访问。

这 3 个预测器就像三道关卡：

1. **Correlated-line** 模块 (最准确, 最保守)
  2. **Page** 模块 (较准确, 中等覆盖)
  3. **Global delta** 模块 (最激进, 最广覆盖)

prefetch controller 会决定用哪个模块。

## Prefetch Engine 的执行顺序

prefetch engine 每次都按以下顺序运行：

1. 先试 Correlated-Line 模块  
若 CL 模块发现可用的 delta pattern → 直接预取

## 2. CL 不行，再试 Page 模块

若 Page 中可找到跨 CL delta → 预取

## 3. 两个都不行，最后用 Global delta

即使只有一个 delta 也敢预取 → 激进预测

每个Prefetch模块的触发条件：

### 3.2.1 Correlated Line-Based Prefetching Module

#### 1. 模块作用

这是三个 prefetch 模块中的第一个，也是最 **保守、精准** 的模块。

它的核心思想：

▣ 只在同一个 correlated line 内发现稳定的访问模式时才预取。

- Correlated line = 8 个连续的 cache line
- 只分析小范围的访问，保证准确率高

---

#### 2. 触发条件

预取触发需要满足条件：

##### 1. 同一 CL 至少有 3 次访问

- 为什么 3 次?
  - 两次 delta → pattern 不够可靠
  - 三次 delta → 模式可靠且覆盖率合理

---

#### 3. 执行流程

当 LLC 发出访问请求，prefetch engine 先检查：

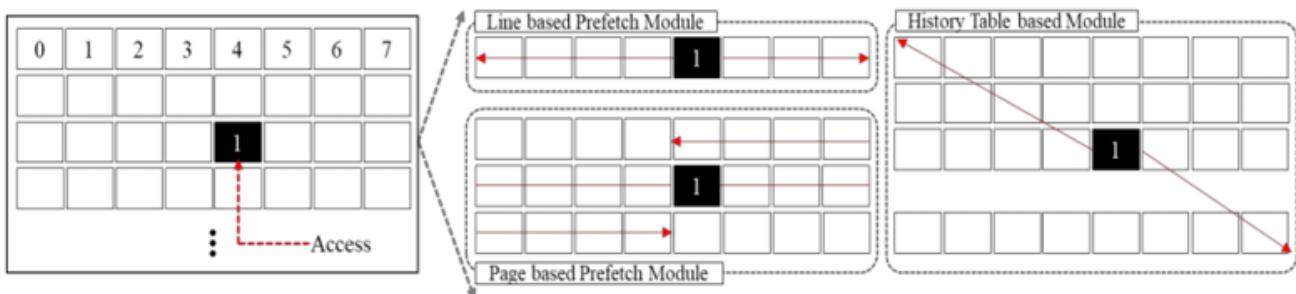


Figure 4. Selection criteria for the prefetching module.

##### 1. 看 prefetch buffer 里有没有数据

- 已存在 → 不用做
- 不存在 → 继续下面步骤

##### 2. 计算当前 CL 已访问次数 (通过 History Table)

- 小于 3 → 不做预取
- $\geq 3$  → 触发 Pattern Matching

### 3. Pattern Matching (匹配模式)

- 从 Correlated-Line Picture Table 中取最近三个访问 block 的 index
- 看这个 pattern 是否存在
  - 存在 → 按这个 pattern 进行预取
  - 不存在 → 去 Page-level Picture Table 查
    - 如果在 page-level table 出现次数  $\geq 2$  → 也可以触发
    - 否则 → 不做预取

#### 3.2.2 Page-Based Prefetching Module

##### 1. 模块核心思想

- 不再按 correlated line 分组，而是 **直接按整个 page (4 KB) 分析**
  - Page 内的访问模式可能更 **不规则**，但通过 **分表管理 picture table**，依然可以提取规律
  - 主要目标：增加覆盖率，抓住跨 CL 的访问模式
- 

##### 2. 触发条件

- 当 **同一页发生  $\geq 3$  次连续访问**，触发 pattern matching
  - 对应 Page-Level Picture Table：
    - 找到匹配 pattern → 根据表里的 delta 生成 prefetch address
    - 匹配不到 → 不触发
- 

##### 3. 执行流程

1. 从物理地址中提取 **page number** → 定位 Page Table entry
2. 在 Page-Level Picture Table 中查找匹配 pattern
3. 如果匹配成功：
  - 取 delta → 加到当前地址 → 发起预取
4. 如果匹配失败 → 不预取

**关键点：**delta 可能是跨 correlated line 的，因此可以捕捉更多不规则模式

#### 3.2.3 History Table-Based Prefetching Module

##### 1. 核心思想

- 不再依赖 picture table
- 直接用 **History Table 中的 delta 相关性** 做预测
- 也就是看最近的访问差值 (delta)，预测下一步可能访问的地址

特点：

- **覆盖率高** → 可以预测稀有或不规则访问
  - **准确率略低** → 因为 pattern 不稳定，可能预取错
-

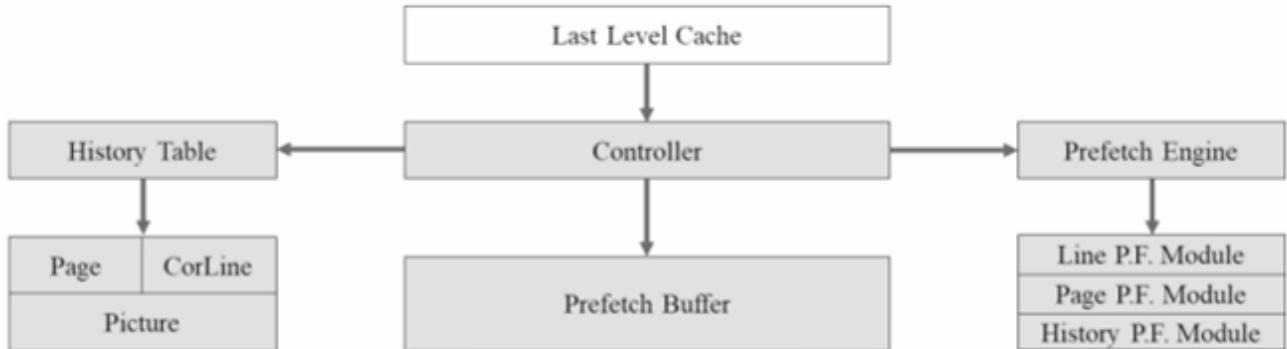
## 2. 触发条件

- 当连续访问跨越 **不同 correlated lines** 时触发
  - 为什么?
    - 如果连续访问在同一个 CL → 前两个模块够精确, 不用第三模块
    - 如果访问在不同 CL → 说明 pattern 可能很不规则, 需要激进预取
- 第三模块只在 **CL 变化时** 才使用 → 代表“进入不规则访问阶段”

## 3. 执行流程

1. 识别最近的访问 delta (在 History Table 中)
2. 分析 delta 间的相关性
3. 根据 delta 生成下一步可能访问的地址 → 发起 prefetch!

### 3.3 Prefetch Controller:



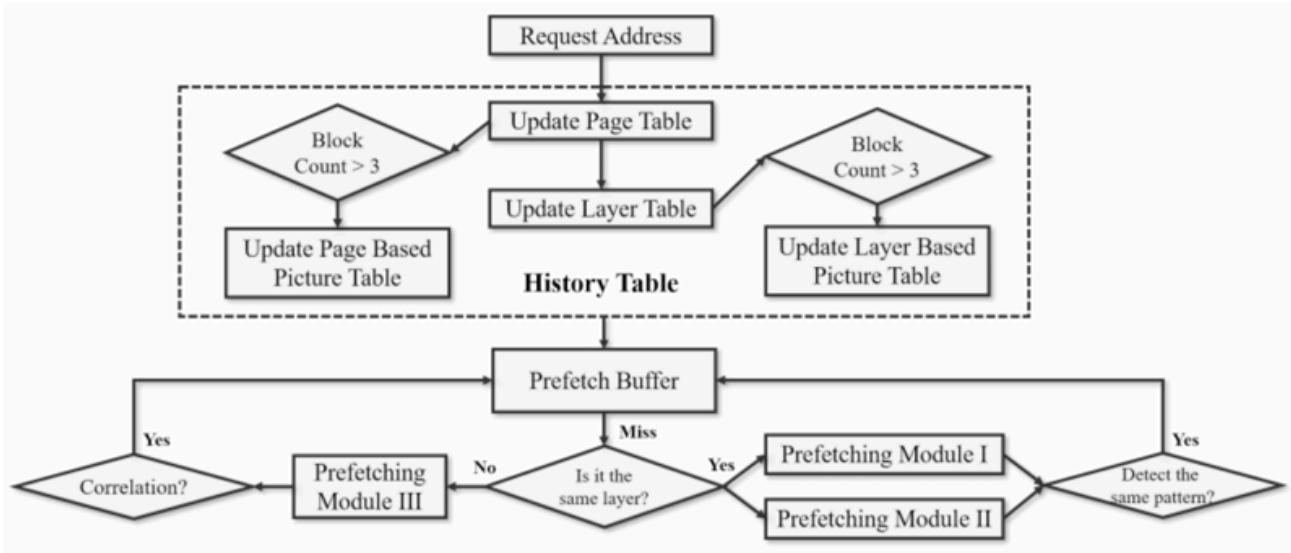
**Figure 5.** Controller.

#### 1. Controller 的核心作用

Controller 是整个 prefetch 系统的大脑, 它管理三个主要功能:

1. **History Table 更新**
2. **选择合适的 Prefetch 模块** (CL / Page / History)
3. **管理 Prefetch Buffer** (只存 DRAM 的预取数据, 并应用 LRU 替换策略)

#### 2. History Table 更新流程



**Figure 6.** Operational flow of prefetch controller.

当 LLC 发起访问请求时，Controller 会：

### 1. 检查 Page Table

- 如果当前 page 已存在 → 更新对应 Correlated Line Table
- 如果 page 不存在 → 在 Page Table 中插入或替换最久未使用的页面 (LRU)

### 2. 更新 Picture Table

- Correlated Line Picture Table → 当同一个 CL 被访问  $\geq 4$  次时更新
- Page Picture Table → 当同一页被访问  $\geq 4$  次时更新

这保证了 picture table 存储的都是“稳定的模式”，便于后续模块做准确预测

### 3. Prefetch 模块选择

- Controller 会根据 **当前访问地址与前一次访问的关系** 来选择模块：

情况	触发模块	原因
当前访问在同一 Correlated Line	Module I (CL-based) 和 Module II (Page-based)	pattern 稳定，优先精准模块
当前访问跨 Correlated Line	Module III (History Table-based)	pattern 不规则，需要激进预取

- Controller 会把预测出的地址发送给 **Prefetch Engine** → 将数据从 DRAM 取到 **Prefetch Buffer**

### 4. Prefetch Buffer 管理

- Controller 管理 Prefetch Buffer，只存储 DRAM 预取的数据
- 使用 **LRU 替换策略** → 保证常用的预取数据优先保留
- 这样可以减少 cache pollution，同时提高系统命中率

### 3.4 Prefetch Buffer:

#### 1. 为什么用独立的 eDRAM buffer?

- 目的是激进预取，而不会：
  - 污染 LLC cache
  - 消耗昂贵的 on-chip SRAM 空间
- eDRAM 优势：
  1. **低延迟 + 高密度** → 存储更多数据且访问快
  2. **比 DRAM 快** → 介于 LLC 和主存之间
  3. **比 SRAM 高性价比** → 省芯片面积

核心思路：用一块大容量、访问快、成本低的 buffer 来存放预取数据，而不是占用 CPU 内部缓存

#### 2. Prefetch Buffer 的组织

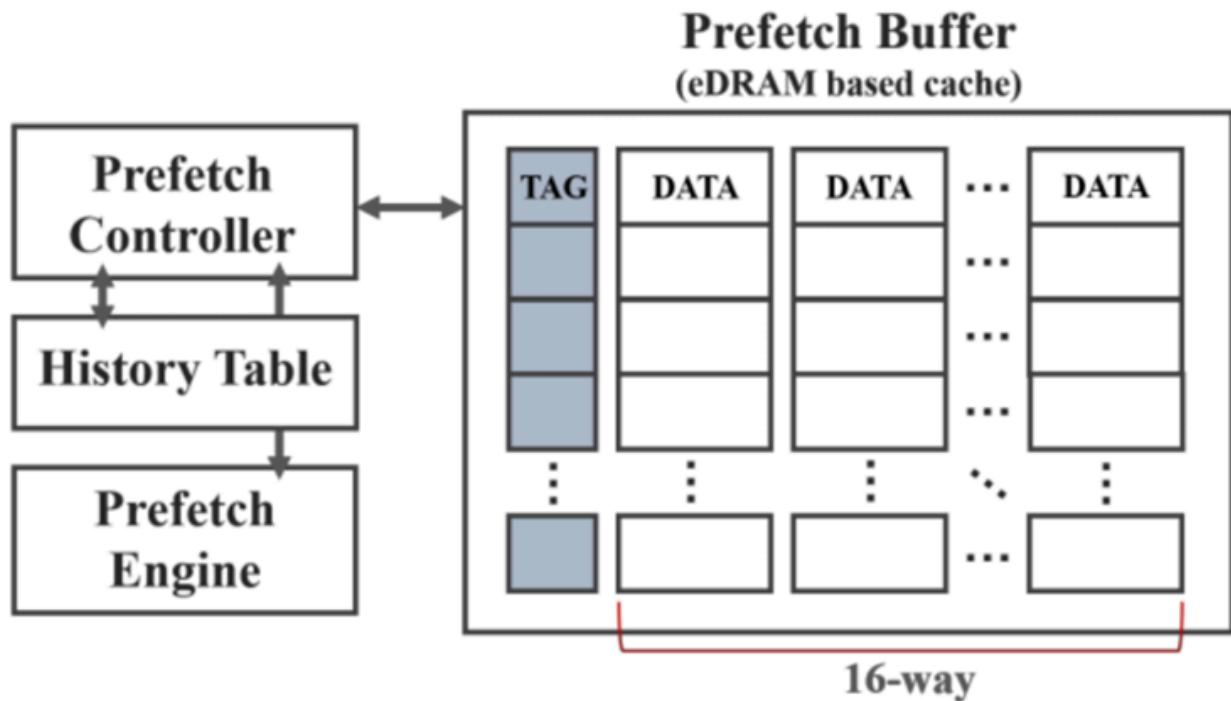


Figure 7. Internal structure of Prefetch Buffer.

- **位置**: LLC 和主存之间
- **大小**: 8 MB
- **块大小**: 64 B
- **结构**: 16-way set associative (类似 LLC cache)
- **替换策略**: LRU
- 与传统 LLC 类似，但 **只存预取数据**，不存实际的 demand request

## 4. GoVector: An I/O-Efficient Caching Strategy for High-Dimensional Vector Nearest Neighbor Search

该论文针对**基于图的高维向量索引**在大规模近似最近邻搜索 (ANNS) 中的存储和访问效率问题进行研究。传统图索引占用内存大，需要存储在二级设备上，而查询过程中频繁的图节点与向量数据加载导致 I/O 成为主要瓶颈，占查询延迟的 90% 以上。现有的静态缓存策略仅在搜索初期通过预加载入口点和多跳邻居缓解 I/O 压力，但在后续依赖查询动态访问的阶段效果有限。

论文提出了 **GoVector** 缓存策略：

1. **静态缓存**: 存储入口点和访问频繁的邻居节点。
2. **动态缓存**: 在第二搜索阶段自适应捕获高空间局部性的节点。
3. **磁盘节点重排**: 将相似向量存放在相邻页上，以提升空间局部性并减少 I/O。

### Related Works:

主流向量索引方法主要分为三类：

1. **基于哈希的索引** (如 LSH)：通过多哈希函数将向量映射到不同桶中，仅在同桶内搜索。计算开销低，但高召回率需要大量哈希表，增加内存和维护成本，限制了可扩展性。
2. **基于量化的索引** (如 IVF PQ、SCANN)：通过向量压缩或空间划分减少计算和存储开销，广泛用于工业系统。但高精度搜索任务中，量化误差可能导致相似向量被分配到不同单元，降低召回率。
3. **基于图的索引** (如 HNSW、NSG、DiskANN)：构建稀疏的可导航邻居图，采用启发式遍历策略进行高效近似搜索，因其高准确率和良好可扩展性，近年来成为大规模向量检索的主流方案。

图索引优化方法包括：

- **索引存储优化**：磁盘图索引的物理布局直接影响 I/O 效率。主流策略包括按插入顺序存储和哈希分布存储，但前者忽略向量相似性，后者破坏邻居物理局部性，均导致随机 I/O 增多。为此，Faiss-IVFOPQ 通过倒排索引聚类相似向量并进行乘积量化压缩，减少磁盘寻道和 I/O 数据量；LENS 系统从查询日志挖掘热点并主动预加载内存，提高缓存命中率约 25%。
- **缓存优化**：现代 ANNS 系统通过缓存关键节点数据降低查询磁盘访问延迟。DiskANN 预加载高频入口节点及其邻居；Starling 构建内存导航图，为查询提供更接近的入口点，缩短磁盘搜索路径，有效减少 I/O 开销。

总体而言，现有工作在图索引精度、存储布局和缓存策略上均有优化，但仍面临 I/O 瓶颈和查询路径动态性带来的挑战，这也正是 GoVector 提出混合缓存与向量相似度驱动索引重排的出发点。

### GoVector概述：

GoVector 是一种基于向量相似性的 I/O 高效缓存策略，用于提升磁盘驻留图索引在向量搜索中的性能。其整体框架由内存层的混合缓存机制与磁盘层的相似度驱动数据重排两部分组成。

在内存端，GoVector 采用静态 + 动态的混合缓存设计。静态缓存预加载入口节点及其多跳邻居，便于查询在初始阶段快速接近目标区域；动态缓存则在查询过程中自适应保留被访问节点所在的磁盘页，以及与这些节点相邻的页，以适应 top-k 相似性搜索阶段的访问模式，从而显著提高缓存命中率。

在磁盘端，GoVector 根据向量相似性对图索引进行重排，将相似向量聚集到同一或相邻磁盘页中。这样可以增强查询时的数据局部性、减少跨页访问，并有效降低随机 I/O 开销。

整个搜索流程可概括为：

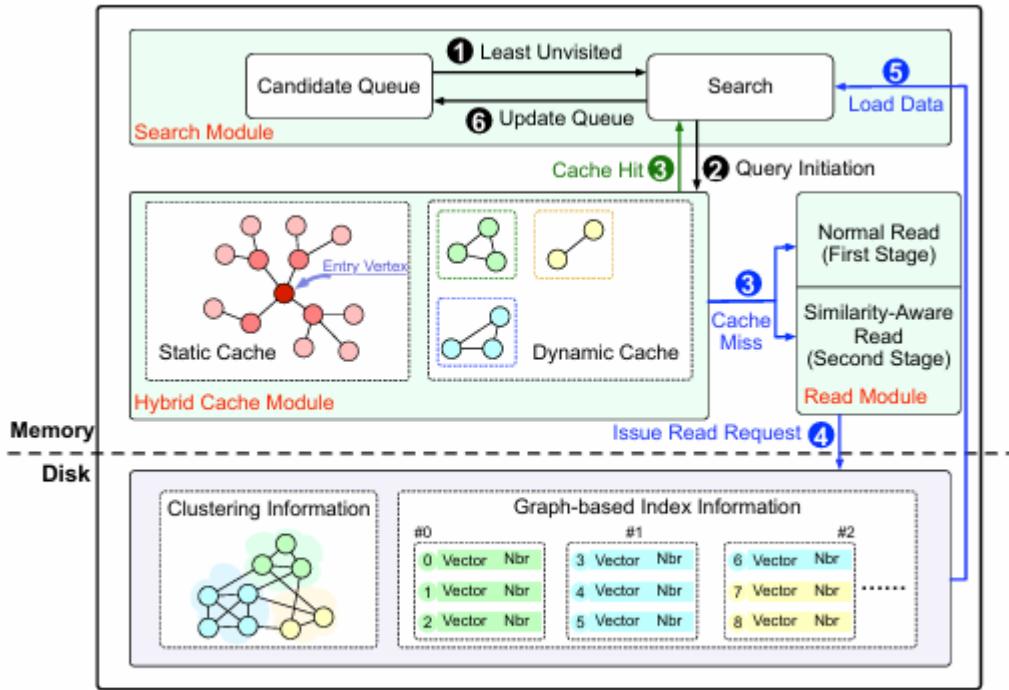


Fig. 4: System architecture of GoVector.

1. 从候选队列中选取当前最接近且未访问的节点进行扩展；
2. 将该节点的 ID 发送至混合缓存模块；
3. 若缓存命中，则直接返回向量和邻接信息，否则向磁盘发起读取；
4. 在初始阶段直接读取对应页，在相似性扩展阶段则基于相似度加载多页以提高命中率；
5. 将读取到的向量与邻接信息载入内存，并在第二阶段写入动态缓存；
6. 计算扩展节点与查询向量的精确距离，并按优先级将其邻居加入候选队列。

#### 4.1 Static-Dynamic Hybrid Caching Strategy:

##### Limitations:

传统的基于 Beam Search 的图索引系统通常采用静态缓存策略，即在系统初始化时预加载入口节点及其多跳邻居，整个搜索过程中缓存内容不发生变化。这种方法在搜索初期能够有效提高缓存命中率，因为查询从入口节点开始，前几跳的节点被频繁访问。然而，随着搜索进入后续阶段，静态缓存无法适应实际查询路径的扩展行为，其命中率迅速下降，导致内存访问效率大幅降低。

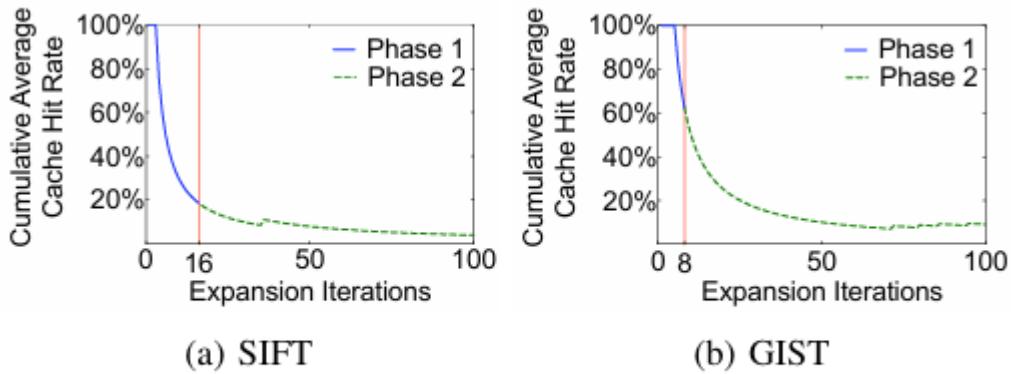


Fig. 5: Static cache hit rate variation in DiskANN over 100 expansion steps ( $k = 10$ ).

在对 DiskANN 的 SIFT 和 GIST 数据集实验中，静态缓存命中率在第一阶段分别为 19% 和 63%，进入第二阶段后迅速下降至仅 4% 和 9%，而第二阶段的搜索时间通常占总查询时间的 80% 以上，成为性能瓶颈。

进一步分析显示，第二阶段扩展节点集中在查询向量附近的环状区域内，节点距离变化范围较小，具有明显空间局部性。如果查询能集中在该局部区域内，并对向量进行顺序重排与物理存储优化，则每次磁盘 I/O 读取的向量更可能被访问，从而提高缓存命中率并减少冗余 I/O。

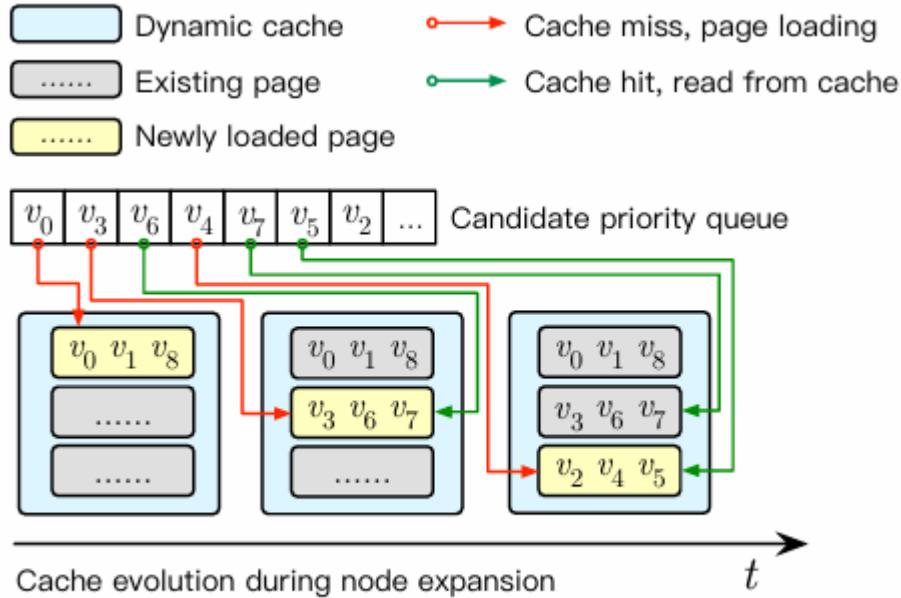
然而，现有磁盘图索引方法（如 DiskANN）仍采用逐点加载策略，每次仅读取包含当前扩展节点的磁盘页，读取后立即从内存中清除，忽略了第二阶段节点的空间聚集性，导致重复 I/O 操作频繁。因此，第一阶段可继续使用传统静态缓存，而第二阶段则需要利用环状局部区域的空间局部性，采用增强缓存策略以高效支持查询路径上的热点访问。

#### Design of Query-Aware Hybrid Caching:

基于第二阶段查询路径的空间局部性分析，GoVector 提出了一种**查询感知的混合缓存机制**，结合静态缓存与动态缓存策略。

在第一阶段，仍采用传统的静态缓存策略，加速入口节点及其邻居的频繁访问；当进入第二阶段时，系统切换到动态缓存机制，重点优化局部访问。具体而言，当扩展节点在缓存中未命中时，GoVector 根据节点在向量空间中的位置触发**批量读取**，将多个相邻向量从磁盘加载到动态缓存中，利用扩展节点的空间相似性提高后续缓存命中率，从而显著降低随机 I/O 的性能开销。

为了支持动态缓存，GoVector 引入**相似度感知读取策略**：



(b) Vertex expansion process and dynamic cache updates

- 在索引构建阶段，将相似向量聚类并按物理顺序存储，以保证磁盘访问的局部性；
- 在读取阶段，系统根据扩展节点的类别及在类别内的位置计算最优顺序读取区间，实现集中加载目标节点及其邻近向量；
- 加载策略遵循三条原则：以目标节点为中心，优先加载同类相邻向量；若类别不足以填充缓存页，则补充相邻类别向量；确保读取区间不超出边界。

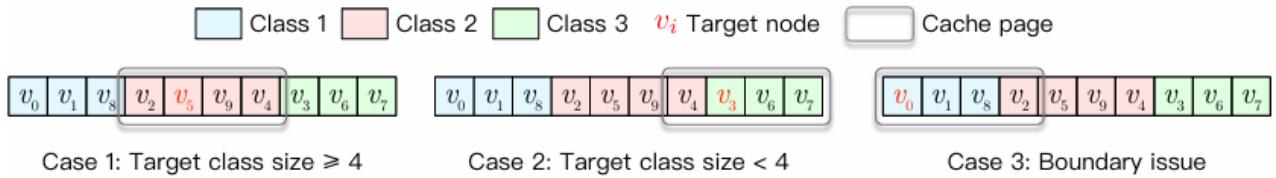


Fig. 7: Illustration of the similarity-aware reading mechanism.

Case3:(在动态缓存批量加载向量时，系统通常会以目标节点为中心向两边扩展，把相邻的向量一起加载到缓存里。但有时候，如果目标节点靠近索引文件的开头（即最左边），按照常规“以目标节点为中心”的加载策略向左扩展，就会超出索引文件的实际范围——也就是不存在的数据地址。)

随着查询进行，动态缓存数据量不断增长，达到容量上限时需要进行替换。GoVector 借鉴操作系统虚拟内存管理，实现了 FIFO、Random 和 LFU 三种替换策略，并基于实验选择 LFU 作为默认策略。

**阶段划分背景：** GoVector 的搜索分为两阶段——

- 第一阶段：从入口节点开始，快速接近目标邻域，静态缓存有效；
- 第二阶段：搜索目标节点附近的局部区域，需要动态缓存和批量读取。

为了准确识别第一、二阶段的切换点，GoVector 引入可调参数  $\theta$ ，第二阶段的核心是“进入目标节点附近的局部区域”，而  $\text{top-}\theta\cdot k$  候选节点的访问情况能够 提前反映搜索已经接近目标区域。当这些关键候选节点大部分访问过，就说明搜索路径已经进入了需要动态缓存和相似度批量加载的阶段

## 4.2 Vector-Similarity-Based Reordering:

### Analysis of I/O Efficiency Issues:

在大规模向量搜索中，**磁盘 I/O 成为性能瓶颈**，主要原因之一是每次 I/O 的利用率低。由于磁盘访问以页为单位，当系统读取某节点的向量或邻居信息时，需要加载整个包含该节点的页。如果该页中大部分向量在本次查询中未被使用，就造成了带宽浪费。为了继续搜索其他相关向量，系统必须重复加载新的磁盘页，增加了额外 I/O 开销。

以 DiskANN 为例，约 94% 的页内向量未被访问，92.5% 的查询时间用于磁盘 I/O，这一问题在 ANNS 第二阶段尤其明显。理想情况下，如果扩展节点及其相邻向量被存放在同一磁盘页中，一次 I/O 就可以加载多个潜在有用向量，从而提高每次 I/O 的效率，减少冗余访问。

然而，目前主流的索引布局策略多按照图结构邻接关系组织页，例如 Starling 使用 BNF 算法将图上邻接节点聚集，但这种方式忽略了**图拓扑与向量空间相似性的差异**，不能保证相似向量的物理局部性，因此 I/O 局部性提升有限。

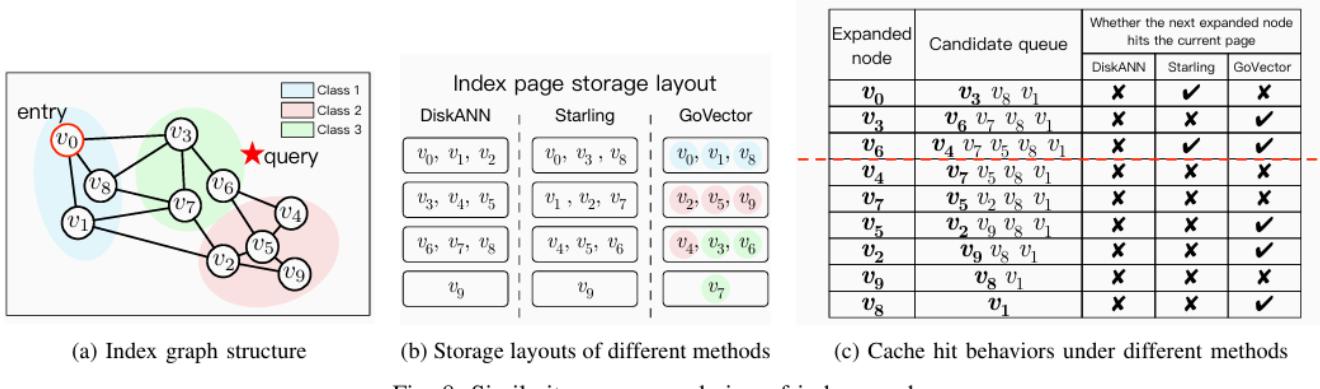


Fig. 8: Similarity-aware reordering of index graphs.

红线上半部分为第一个阶段：从 entry  $v_0$  到离 query 最近的点  $v_6$

红线下半部分为第二个阶段：从  $v_6$  开始探索  $v_6$  近邻，寻找是否有更近的点（每次从候选队列中选择最接近查询向量的未访问节点进行扩展，计算该节点与查询向量的精确距离，并将其邻居加入候选队列。）

实验（图 8）显示，DiskANN 和 Starling 的布局在第二阶段搜索中经常需要多次 I/O 访问相邻扩展节点，导致每次 I/O 利用率低。尽管增加缓存容量或采用异步预取可以缓解问题，但更有效且成本更低的方式是设计**基于向量相似性的索引重排布局**，通过提升数据局部性直接改善 I/O 效率。

GoVector 为了提升磁盘访问效率，对图索引进行了**基于向量相似度的重排**，整个过程分两步：

1. **相似度聚类**：使用 k-means 将所有向量按照欧氏距离划分为多个高相似度的簇，每个簇内的向量在向量空间中彼此接近。
2. **局部布局优化**：在每个簇内，根据图结构进一步重排向量，并尽量把簇内向量放到同一磁盘页或相邻页，减少跨页访问。

这样做的效果是：在第二阶段搜索时，扩展的相似向量很可能已经在同一页里，从而**一次 I/O 可以利用更多有用向量**。相比传统的 DiskANN（按插入顺序或随机存储）或 Starling（按图拓扑邻居存储），GoVector 更符合第二阶段“相似度驱动”的访问模式，大幅提升 I/O 利用率，减少磁盘访问次数，提高查询性能。

# 5. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment

## 设计理念:

在分析现有基于磁盘的图索引方法时，I/O 时间主要受两个因素影响：块内顶点利用率和搜索路径长度。顶点利用率越高，每次磁盘读入的数据越多为有效信息，可减少不必要的 I/O；搜索路径越短，访问磁盘的次数越少。然而，现有方法存在两大问题：

1. **数据局部性差**：传统方法按 ID 顺序存储顶点，每块包含多个顶点。实际查询中，大部分顶点与目标无关，导致块内顶点利用率低。例如 BIGANN 数据集上，约 94% 的磁盘读取数据被浪费。
2. **搜索路径长**：入口点随机或固定，可能远离查询目标，导致路径冗长，需要多次磁盘访问才能完成搜索。

为解决上述问题，提出 **Starling** 框架，其设计理念包括：

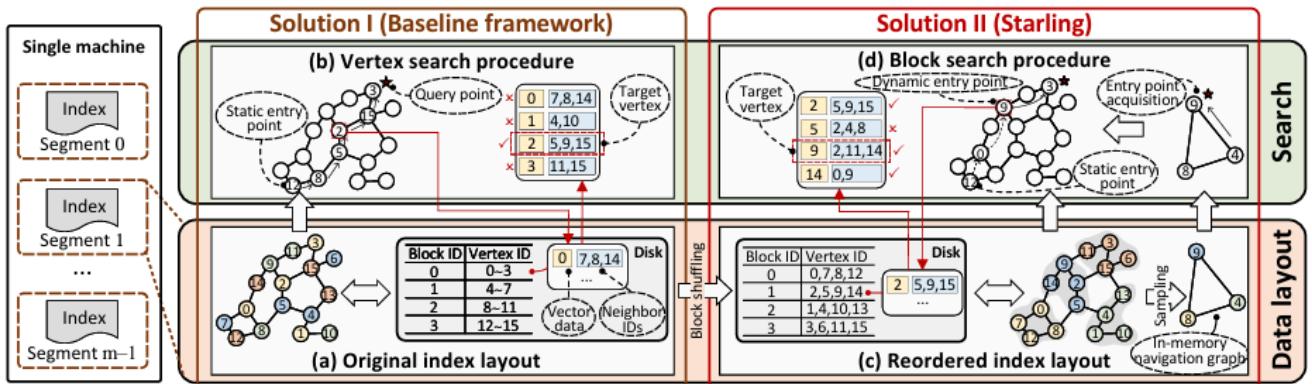


Fig. 2. Illustration of the data layouts and search strategies for the baseline and Starling, respectively.

- **磁盘数据布局优化**：根据图拓扑重排数据块，将顶点及其邻居存放在同一块中，提高块内顶点利用率，减少磁盘访问次数。（比如现在0, 7, 8, 12就变成一个block）
- **内存导航图**：在内存中构建小型导航图，为查询选取接近目标的入口点，从而缩短磁盘搜索路径。
- **块级搜索策略**：每次按块访问顶点，计算块内所有顶点与查询的距离，并选择合适的顶点继续搜索。该策略充分利用优化后的数据布局，降低磁盘 I/O，但增加计算量，通过专门优化提升整体性能。

**示例说明：**原方法从顶点 12 到顶点 3 需要至少 6 次磁盘 I/O，而 Starling 通过重排索引和内存导航图，仅需 2-3 次 I/O 即可完成搜索，显著提升大规模场景下的检索效率。

## 5.1 Block Shuffling on the Disk:

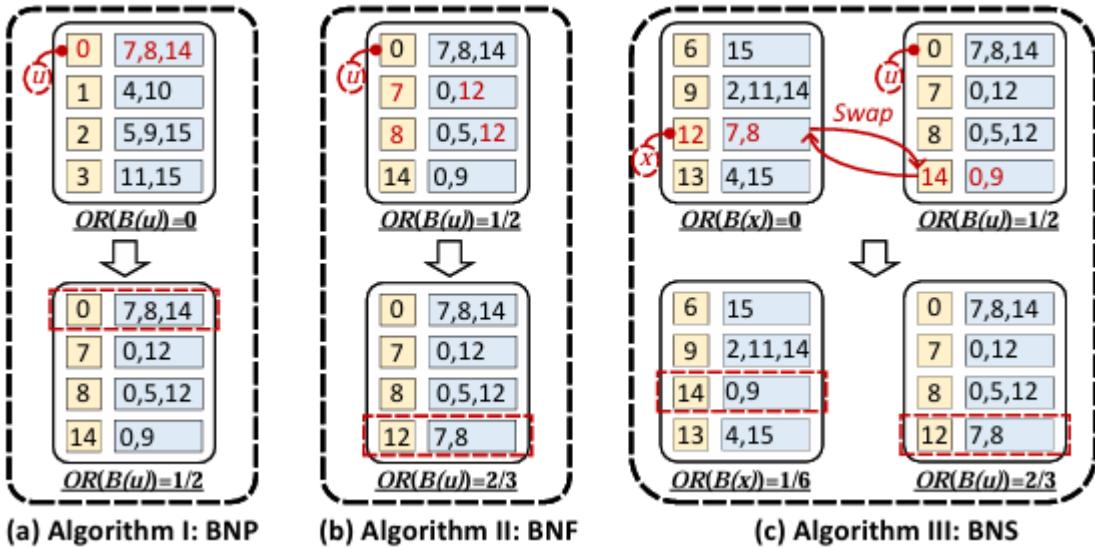
- 磁盘块重排 (Block Shuffling)

为了提升基于磁盘的图索引的数据局部性，Starling 提出了块级重排策略。设磁盘图索引为  $G = (V, E)$ ，每个顶点存储向量数据、邻居数量  $\lambda$  和邻居 ID 列表（最大长度  $\Lambda$ ），每个磁盘块大小为  $\eta$  KB，每块最多容纳  $\epsilon = \lfloor \eta / \gamma \rfloor$  个顶点，总块数为  $\rho = \lceil |V| / \epsilon \rceil$ 。

**图布局定义**：块级图布局是将  $|V|$  个顶点分配到  $\rho$  个块的方案。数据局部性用重叠率  $OR(G)$  衡量：对顶点  $u$ ， $OR(u)$  为其邻居在同块顶点中占比；块重叠率为块内顶点  $OR(u)$  的平均值，图的重叠率为所有顶点的平均。重叠率越高，表示数据局部性越好，块内顶点利用率越高。

- 块重排启发式算法

针对块重排问题，Starling 设计了三种启发式算法，用于提高块内顶点重叠率  $OR(G)$ ：



### 算法 I：块邻居填充 (BNP, Block Neighbor Padding)

- 按顶点 ID 顺序依次填充磁盘块，尝试将顶点及其邻居放入同一块。
- 时间复杂度为  $O(|V|)$ ，能够提高块内邻居重叠率，但受限于邻居分散或已被分配到其他块，提升有限。
- 示例：BNP 将顶点 0 及邻居 7、8、14 放入同一块，使  $OR(B(u))$  从 0 提升到  $1/2$ ，但顶点 12 无法与邻居 7、8 同块，仍为 0。

### 算法 II：块邻居频率 (BNF, Block Neighbor Frequency)

BNF 以 BNP 的布局作为初始状态，并通过迭代优化块分配。算法流程如下：

(1) **初始化阶段**：保存当前“顶点 → 块”的映射（记录每个顶点目前属于哪个磁盘块，比如 0, 7, 8, 14 属于同一块），并清空所有块，以便重新分配（对应 Algorithm 1 的 lines 3-5）。

(2) **基于邻居频率的分配阶段**：对每个顶点  $u \in V$ ，BNF 会统计其邻居在各块中的数量，并按邻居数量从高到低依次尝试分配（lines 6-14）。

- 若某块有空位，则将  $u$  放入该块；
- 若所有块均已满，则将  $u$  放入当前的空块中。

(3) **迭代收敛阶段**：上述步骤重复进行，直到达到迭代次数上限  $\beta$ ，或连续两次迭代的  $OR(G)$  提升低于阈值（lines 15-16）。最终输出新的块布局。

原来一个块  $B(u)$  里是：

{0, 7, 8, 14} (同一个颜色才是同一块，横着的代表的是这个块里面这个点的邻居)

但 14 在这个块里没什么邻居（关系弱）。

而 12 在这个块里有两个邻居（7 和 8）——关系强。

于是 BNF 会做一件事：

把 14 换成 12，让这一块关系更紧密。

换完变成：

{0, 7, 8, 12}

### 算法 III：块邻居交换 (BNS, Block Neighbor Swap)

- 灵感来自 NN-Descent，通过在不同块的邻居顶点间交换重叠率最低的顶点，迭代提升  $OR(G)$ 。

- 时间复杂度为  $O(\beta \cdot o^3 \cdot \epsilon \cdot |V|)$ , 保证迭代过程中  $OR(G)$  单调不减 (Lemma 4.2)。
- 示例：通过交换顶点 12 与 14，进一步提升块重叠率。
- BNS 的效果最好，但计算成本最高，可通过并行化加速。

## 性能分析

- 三种算法均显著提高  $OR(G)$ ，进而提升块内顶点利用率和搜索性能。
- BNP 最快但效果有限；BNF 兼顾效率与效果；BNS 提升最大但时间成本高。
- 实践中推荐 BNF 作为默认选择，兼顾效率和性能。

## 时间与空间开销

- 块重排仅涉及顶点扫描和简单统计，不需要向量计算，额外时间成本低 (BNF 约占图索引构建时间的 3%-10%)。
- 空间成本保持不变，仅调整顶点顺序，不增加额外存储。

## 5.2 In-Memory Navigation Graph:

### (1) 随机采样一部分数据点 (不到整体的 10%)

只取小部分数据放进内存，因为内存有限。

### (2) 用与磁盘图同样的算法 (HNSW、NSG 等) 构建内存导航图

这张图很小，驻留在内存里，可以很快定位“更接近 query 的入口点”。

## 5.3 Search Strategy:

Starling 的搜索策略分为四步：

- (1) 内存导航图上找“更接近 query 的入口点”(不访问磁盘)
- (2) 从这些入口点开始，在磁盘图上做 block search (访问磁盘)
- (3) ANNS查询
- (4) Range Search

第二步是核心，因为磁盘 I/O 最慢。

### 什么是 Block Search (块搜索) ?

Starling 在离线阶段把“互相是邻居的顶点”尽量放进同一个磁盘块。

这样：

**一次磁盘读 I/O 不是只读到 1 个顶点，而是一次读到一群相关的顶点 (例如 64 个)。**

块搜索做两件事：

- 计算 block 内所有顶点到 query 的距离
- 对距离近的那些顶点，继续取它们的邻居做下一步搜索

下面是块搜索的三个关键优化。

### (1) Block Pruning (块剪枝)

现实中 OR(G) 很难达到 1，往往只有 0.3~0.6，也就是：

**一个 block 里有部分顶点和这次搜索其实没关系，是“无效顶点”。**

所以 Starling：

- 把一个 block 内所有顶点按“距离 query 的远近”排序
- 只对前  $\sigma(\epsilon-1)$  个靠近 query 的顶点检查他们的邻居
  - $\epsilon$ : 一个 block 的顶点数
  - $\sigma$ : 剪枝比例 (最佳经验值 0.3)

效果：

**远距离顶点的邻居被丢掉 → 少算、少扩展 → 更快**

---

## (2) I/O 与计算并行 Pipeline (边读边算)

朴素方法：

- 先做一次 disk read (DR)
- 再对 block 里的所有点算距离 (DC)  
→ DR 等 DC, DC 等 DR, CPU 和磁盘互相空转

Starling 改成 pipeline：

- DR 与 DC 并行
- while 计算当前 block, 其实下一次 DR 已经在读了

好处：

**磁盘与 CPU 同时满载使用，查询延迟明显下降。**

---

## (3) PQ (Product Quantization) 近似距离

问题：

图索引中某个节点的邻居很多，比如一个节点有 200 个邻居，但一个 disk block 可能只能放 50 个向量。

→ **很多邻居向量不在当前 block 里，要从磁盘读。**

Starling 的解决方案：

- 先把所有向量 PQ 编码 (几 bytes 一个)。PQ code 全在内存中。

当你需要判断下一步往哪个邻居跳，但那个邻居的真实向量不在内存：

**不用去磁盘读真实向量**

→ **直接用 PQ code 和 query 的 PQ code 算一个近似距离。**

这个近似距离非常快 (只查表)，能支持快速排序邻居。

然后只从磁盘 **读最可能有用的那几个 block**，而不是读所有候选邻居的真实向量。

**ANNS查询：**

Starling 在执行近似最近邻搜索 (ANNS) 时采用候选集合与结果集合两个有序结构来组织搜索过程。系统首先通过内存中的导航图获取若干查询的入口点，并将这些入口点加入候选集合与结果集合。候选集合维持固定大小以避免产生过多候选节点，并按照基于 PQ 短码的近似距离排序；结果集合不设大小限制，仅在最终阶段进行精确排序。搜索过程中每次从候选集合中选择与查询最接近的未访问节点，读取该节点所在的磁盘块，并将其邻居加入候选集合，同时

将节点本身加入结果集合。为了减少不必要的 I/O，系统会在块内进行基于查询的剪枝，只保留与查询较近的子集用于后续并行扩展。算法持续执行直到候选集合中不存在未访问节点，最后再依据真实向量计算精确距离并返回最终的 top-k 结果。

### Range查询：

在范围查询 (Range Search) 中，由于相同半径可能产生从零到数千不等的结果数量，Starling 需要动态适应结果规模变化。其方法是在 ANNS 的基础上引入“踢出集合”，用于存储从候选集合中移除但仍可能有价值的节点。系统首先初始化候选集合与结果集合，并开始常规扩展。当候选集合全部访问完后，系统计算结果集合规模与候选集合规模的比例；若该比例超过阈值  $\phi$ （实验表明  $\phi=0.5$  最优），说明许多候选实际上是有效结果，因而系统会扩大候选集合容量并重新启动搜索。新的搜索会加入上一轮被踢出的、更接近查询的节点，从而避免重复访问磁盘。该机制使范围查询能够在结果规模大的情况下继续发现新结果，同时避免重复 I/O 开销；当比例低于阈值时则终止搜索并返回结果。

## 6. Accelerating Graph Indexing for ANNS on Modern CPUs

HNSW 图索引因性能突出而成为主流，但其索引构建耗时极长，在千万到十亿规模数据上甚至需要数小时到数天，严重制约实际系统的更新与服务部署。作者分析发现，HNSW 构建过程的主要瓶颈来自距离计算，其占据超过 90% 的时间，问题根源包括频繁的随机内存访问导致的高延迟，以及高维浮点数据使 SIMD 指令难以高效发挥。基于“构建阶段只需比较、无需精确距离”的特性，他们研究了紧凑编码，并最终提出面向 CPU 的 Flash 编码策略，通过减少随机访存并充分利用 SIMD 并行，有效提升缓存命中率并降低算术开销。实验结果表明，Flash 在千万至十亿规模数据集上可实现数量级的索引构建加速，同时保持甚至提升搜索性能，显著突破了 CPU 上构建效率的瓶颈。

### 问题分析：

先形式化了 HNSW 的构建流程，指出每次插入向量都需要执行候选获取和邻居选择两个步骤，而二者都大量依赖距离比较。由于向量数据与邻居 ID 分离存储，在遍历邻居列表时需要频繁随机访存才能取到完整向量，导致 CPU 等待内存的时间远大于计算时间。更重要的是，高维向量使 SIMD 很难高效利用，一个距离计算需要重复加载几十到上百次 128-bit 寄存器，而这些加载本身在乱序访存下非常低效。

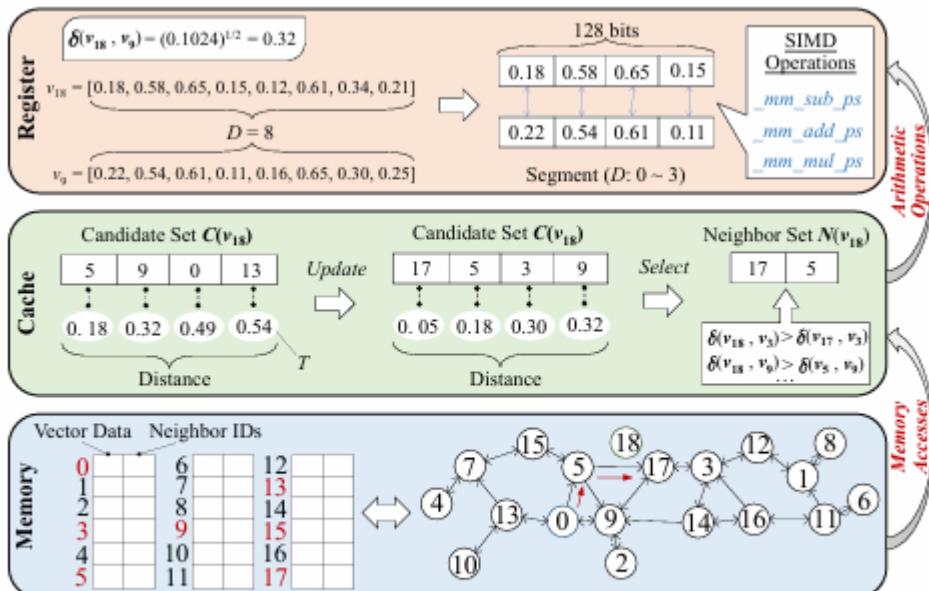


Figure 2: Illustrating memory accesses and arithmetic operations during the HNSW construction process (base layer).

展示了 HNSW 在基础层插入一个新向量 ( $v_{18}$ ) 的完整构建过程：

在候选获取阶段， $v_{18}$  被当作查询点，从  $v_0$  开始进行贪心搜索，不断访问邻居并计算距离以更新候选集合。每当找到比当前最远候选更近的顶点时，候选集合就会更新，例如：一开始是  $v_0$  的邻居集合  $\{v_0, v_5, v_9, v_{13}\}$ ，然后继续访问二跳节点， $v_5$  邻居  $v_{15}$  离  $v_{18}$  相比  $v_{13}$  更近，因此  $v_{13}$  被  $v_{15}$  替换，随后又被  $v_{17}$  和  $v_3$  替换，最终候选集合为  $\{v_{17}, v_5, v_3, v_9\}$ 。由于这些顶点在内存中分布随机，访问它们需要大量随机内存读操作。

在邻居选择阶段，算法从候选集合中选出最终邻居，通过距离比较剔除不合适的选择，最后  $v_{17}$  和  $v_5$  被加入邻居列表，并将  $v_{18}$  反向加入它们的邻居列表，同时确保邻居数不超过上限  $R$ 。向量被分成多个四维段以利用 128-bit SIMD 寄存器加速计算，但由于需要多次加载寄存器，SIMD 的利用效率受到限制，整个过程的瓶颈仍然在大量的距离计算和随机访存。

因此，想要真正加速 HNSW 构建，就必须从距离计算和数据布局入手，减少随机访存并提升 SIMD 并行效率。

### 向量压缩：

HNSW 构建的核心瓶颈是距离比较（DC1 用于候选更新，DC2 用于邻居选择）。通过 Lemma 1 可知，两向量距离的大小关系可以表示为向量在某个超平面上的位置判断，即  $\delta(u, v) > \delta(u, w) \iff e \cdot u - b > 0$ 。

Theorem 1 进一步说明，即使向量经过压缩（如 PQ、SQ、PCA），只要压缩误差  $E$  足够小，使得  $|e \cdot u - b| \geq |E|$ ，距离比较的符号不变，DC1 和 DC2 的结果仍然正确。这意味着通过调整压缩参数，可以在保证距离比较正确的前提下大幅减少向量存储大小，提升内存访问效率和 SIMD 利用率。

具体操作是：对数据集随机抽样生成向量三元组  $(u, v, w)$ ，为每个向量生成压缩向量  $u', v', w'$ ，计算  $|e \cdot u - b|$  和  $|E|$ ，调整压缩参数以最大化满足条件的三元组比例，同时最小化存储开销，然后将压缩向量整合进 HNSW 构建中验证效果。

三种向量压缩方法在 HNSW 构建中的应用及局限性：

- HNSW-PQ (Product Quantization)**：将向量分成子空间，每个子空间用代码本压缩。候选获取阶段用 ADC 查表加速距离计算，邻居选择阶段用 SDC 估算候选距离。调整子空间数  $M_{PQ}$  和码长  $L_{PQ}$  可以权衡压缩误差和构建效率，但随机内存访问模式未改变，SIMD 利用率仍不高。
- HNSW-SQ (Scalar Quantization)**：每个维度独立量化为整数，解码后计算距离。通过选择合适的比特数  $L_{SQ}$  可兼顾压缩误差和效率。相比 PQ，减少了寄存器加载，但距离计算仍顺序执行，SIMD 利用率不充分。
- HNSW-PCA (Principal Component Analysis)**：通过投影将高维向量降到低维，保留主要成分形成压缩向量。距离计算在低维空间完成，减少存储和计算量。调整保留维度  $d_{PCA}$  可平衡索引质量与构建时间，但随机访问模式和顺序计算仍存在。

## 6.1 Flash Method:

### 1. 设计概览：

Flash 结合 PQ 的子空间分割思想，将高维向量分成  $M_F$  个子空间，每个子空间在 SIMD 寄存器中并行处理。CA 阶段使用非对称距离表（ADT），存储在寄存器中快速计算部分距离；NS 阶段使用对称距离表（SDT），存储在缓存中以避免重复随机访问。邻居 ID 与对应的 codewords 聚合存储，批量加载以减少随机访问。

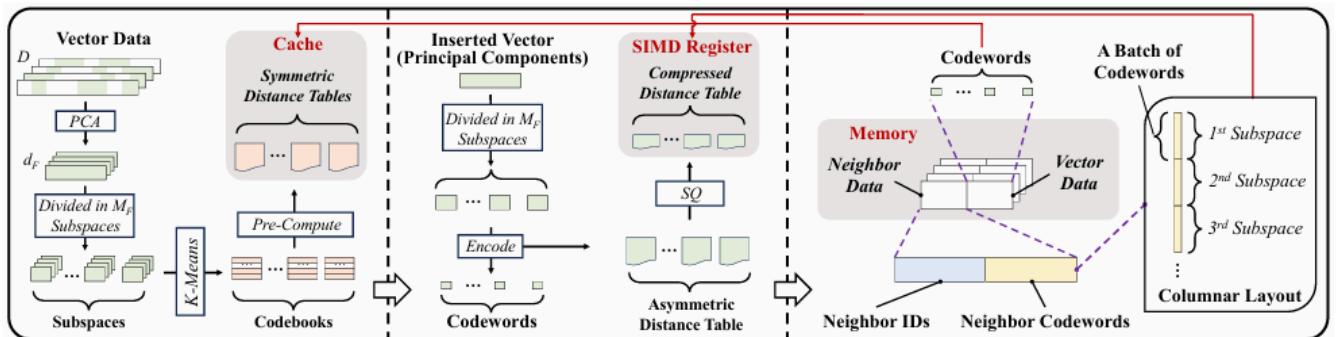


Figure 5: Illustrating the coding pipeline and data layout of Flash.

## 2. 主成分提取 (PCA)

Flash 首先对高维向量进行 PCA 降维：

- 对数据集  $\mathcal{S}$  中的  $n$  个向量，每个维度为  $D$ ：

1. 计算均值向量  $\bar{\mathbf{u}} = \frac{1}{n} \sum_{i=1}^n \mathbf{u}_i$ , 将数据中心化:  $\mathbf{S} = \mathbf{S} - \mathbf{1}\bar{\mathbf{u}}^\top$ 。
2. 计算协方差矩阵  $\Sigma = \frac{1}{n} \mathbf{S}^\top \mathbf{S}$ , 并提取特征值  $\lambda_1, \dots, \lambda_D$  和特征向量  $\mathbf{a}_1, \dots, \mathbf{a}_D$ 。
3. 根据累积方差比例  $\alpha$  选择前  $d_F$  个主成分, 形成投影矩阵  $A_{1:d_F} = [\mathbf{a}_1 \dots \mathbf{a}_{d_F}]$ 。

- 投影得到低维向量  $\tilde{\mathbf{u}}_i = A_{1:d_F}^\top \mathbf{u}_i$ 。

**作用：**优先保留高方差维度，使有限的位数编码能够承载更多有效信息，降低量化误差。

---

## 3. 子空间划分与编码

Flash 将降维后的向量  $\tilde{\mathbf{u}}$  分为  $M_F$  个子空间  $[\tilde{\mathbf{u}}_1, \dots, \tilde{\mathbf{u}}_{M_F}]$ :

- 每个子空间生成一个代码本  $C_i = \{\mathbf{c}_{i1}, \dots, \mathbf{c}_{iK}\}$ , 即 PQ 的思想。
- 对每个子空间，量化为最近中心点的索引 (codeword) :

$$\psi(\tilde{\mathbf{u}}_i) = \arg \min_{\mathbf{c}_{ij} \in C_i} \|\tilde{\mathbf{u}}_i - \mathbf{c}_{ij}\|$$

- 每个向量最终由  $M_F$  个 codeword 表示。

**作用：**每个子空间的 codeword 只需少量位表示，可直接存入 SIMD 寄存器。

---

## 4. 非对称/对称距离表 (ADT/SDT)

Flash 将距离计算表优化为寄存器友好形式：

- **ADT (Asymmetric Distance Table)** : 用于 CA 阶段，存储插入向量  $\mathbf{u}$  与子空间中心点的距离。
  - 通过 SQ 压缩距离到离散值 ( $\eta(\text{dist})$ )，使整个表能驻留在 SIMD 寄存器中。
  - 每次批量处理  $B$  个邻居，使用 shuffle 操作并行提取部分距离并求和。
- **SDT (Symmetric Distance Table)** : 用于 NS 阶段，存储同一子空间内所有 centroid 之间的距离。

### NS 阶段的任务

- 在 HNSW 构建过程中，当一个新向量  $u$  插入后，CA 阶段先找到候选邻居集合。
- NS 阶段要在这些候选邻居里 **挑选最终的近邻**，这时需要计算 **候选节点之间的距离**，保证邻居集合的多样性（避免过于集中在同一簇）。
- 注意：这个阶段计算的是 **候选节点之间的距离**，而不是 query 与候选节点的距离
- 所有向量共享同一 SDT，减少随机访问。
- 同样通过量化压缩以优化缓存使用。

---

## 5. 内存布局优化

Flash 对图索引中的邻居列表做了访问感知布局：

- 对每个顶点，邻居 ID 和对应 codewords 分开连续存储。
- 批量访问  $B$  个邻居的 codewords，利用 SIMD 并行加载和部分距离计算。
- 避免了传统 HNSW 需要从散布向量位置读取数据的随机访问问题。

**效果：**大幅提高缓存命中率，同时保证 SIMD 并行化执行。

## 6. SIMD 加速策略

- 每个 SIMD 寄存器存储 ADT 的部分距离信息。
- 利用 shuffle 操作将  $B$  个邻居的 codewords 作为索引，同时提取部分距离。
- 将不同子空间的部分距离累加，得到最终的距离结果。
- 整个流程无需循环加载单个向量或条件分支，从而最大化寄存器和 SIMD 利用率。

在HNSW上实现：

Flash 在 HNSW 上的实现可以理解为一个从数据预处理到 SIMD 加速的完整流水线，下面详细说明每个环节及其优化效果：

### 1. 数据预处理与 PCA 降维

- 对整个数据集先执行 PCA，提取主成分，得到低维向量  $\tilde{\mathbf{u}}$ （第 3.3.2 节）。
- 选取前  $d_F$  个维度，保留主要信息，并为 SIMD 寄存器分配合适的子空间维度。

### 2. 子空间划分与代码本生成

- 将  $\tilde{\mathbf{u}}$  划分为  $M_F$  个子空间，每个子空间生成 PQ 风格的代码本  $C_i$ 。
- 每个子空间向量通过最近中心点量化得到 codeword，形成向量的紧凑表示（第 3.3.3 节）。
- 对每个插入向量同时生成 ADT（用于 CA 阶段）和 codewords，避免重复计算。

### 3. ADT 和 SDT 的作用

- **ADT (非对称距离表) :**

- 存储插入向量与子空间中心点的距离。
- 通过量化 (SQ) 压缩到 SIMD 可处理的位数。
- 存储在寄存器中，计算距离时可直接通过 shuffle 操作并行处理  $B$  个邻居的部分距离。

#### ADT (Asymmetric Distance Table)

- 用于 **query 或新向量 vs 已有向量** 的距离计算。
- 假设 query 向量 Q 的子空间向量是：
  - 子空间 1:  $[q_1]$
  - 子空间 2:  $[q_2]$
- 预先计算 Q 与每个中心点的距离表：

子空间 1 (ADT1):

centroid	c0	c1	c2	c3
dist(Q)	2	5	1	4

子空间 2 (ADT2):

centroid	c0	c1	c2	c3
dist(Q)	3	2	4	1

- 某个候选向量 A 的 codewords = [2, 3] (子空间 1 对应 c2, 子空间 2 对应 c3)

- 距离 Q 与 A = ADT1[c2] + ADT2[c3] = 1 + 1 = 2

ADT 直接存 query 到中心点距离, 可以 SIMD 批量算 query vs 多个候选向量。

- **SDT (对称距离表) :**

- 存储每个子空间内所有中心点之间的距离。
- 所有向量共享 SDT, 避免随机内存访问, 全部保存在 L1 cache 中。
- 用于 NS 阶段计算候选节点间距离。

### SDT (Symmetric Distance Table)

- 用于 **候选节点之间的距离计算** (NS 阶段)。
- 假设候选节点 A 和 B 的 codewords:
  - A = [2, 3]
  - B = [1, 0]
- SDT 预存每个子空间中心点之间的距离:
- SDT 存的是 **中心点之间的对称距离** (所以叫 Symmetric Distance Table)。也就是说:

$$SDT1[A_1][B_1] = SDT1[B_1][A_1]$$

子空间 1 (SDT1):

	c0	c1	c2	c3
c0	0	5	2	3
c1	5	0	4	1
c2	2	4	0	6
c3	3	1	6	0

子空间 2 (SDT2):

	c0	c1	c2	c3
c0	0	3	5	1
c1	3	0	2	4
c2	5	2	0	6
c3	1	4	6	0

- 候选 A 与 B 的距离 =  $SDT1[A_1][B_1] + SDT2[A_2][B_2]$   
 $= SDT1[2][1] + SDT2[3][0] = 4 + 1 = 5$

SDT 避免了访问完整向量，直接通过 codeword 查表得到候选之间距离。

#### 4. 内存布局优化

- Flash 将邻居 ID 与其 codewords 分别连续存储。
- 批量处理 B 个邻居，使 SIMD 加载一次即可完成多向量距离计算，减少随机访问。
- 结构支持当邻居数 R > B 时，将邻居分成多个块，每块按 B 大小处理。

#### 5. SIMD 加速计算

- ADT 驻留在寄存器，B 个邻居的 codewords 通过 shuffle 操作作为索引提取部分距离。Shuffle 指令可以根据给定的索引，把寄存器里的元素重新排列，或者选择其中某些元素
- 各子空间的部分距离累加得到最终距离。
- 距离计算仅使用简单加法，避免浮点乘法/减法等复杂操作，提高计算效率（第 3.3.5 节）。

操作流程如下：

1. **寄存器载入**：将该子空间的 ADT 加载到一个 SIMD 寄存器里。寄存器里保存了  $u_i$  到每个中心点的距离。
2. **邻居 codewords 作为索引**：B 个邻居的 codewords 表示它们在这个子空间对应的中心点编号，例如  $[3, 7, 1, \dots]$
3. **Shuffle 操作**：利用 SIMD 的 shuffle 指令，将寄存器里的距离按照 codewords 索引重新排列，从而快速提取 B 个邻居的部分距离。
4. **并行求和**：提取出的 B 个部分距离可以直接用 SIMD 并行加到各子空间的部分距离上，得到最终距离。

举例：

- ADT 寄存器里存放了 8 个中心点到 query 向量的部分距离（每个 subspace 一个寄存器）：

```
ADT = [d0, d1, d2, d3, d4, d5, d6, d7]
```

- 有 4 个邻居，它们在这个 subspace 的 codewords（即索引）为：

```
B_neighbors_codewords = [1, 4, 3, 0]
```

shuffle 操作会把 ADT 里的距离按照这些索引提取出来：

```
shuffle(ADT, B_neighbors_codewords) → [d1, d4, d3, d0]
```

这就对应了每个邻居在该 subspace 的部分距离。（提取出了这 4 个邻居在这个子空间里面离 query 的距离）

- 后续可以再对多个 subspace 的部分距离求和，得到每个邻居的完整距离。
- 整个过程**不访问内存**，全部在寄存器里完成，SIMD 可以同时处理 B 个邻居。

#### 6. 参数调节与平衡

- 可调参数包括：
  - $M_F$ : 子空间数

- $d_F$ : 主成分维度
- $L_F$ : 每个 codeword 位数
- $B$ : SIMD 批量大小
- 这些参数控制压缩误差，同时不会破坏 SIMD 并行或访问模式优化。

## 7. 成本分析与优化效果

### • 内存访问:

- 原 HNSW CA 阶段需  $O(R \log n)$  次访问向量数据，Flash 仅需  $O(\log n)$  次，SDT 存在 L1 cache，NS 阶段无需访问原向量。

### • 寄存器加载:

- 原 HNSW: 每次距离计算需  $N_{RL}^{orig} = 32 \cdot D/U$  次寄存器加载 (D=向量维度, U=寄存器位宽)。
- Flash: 每次只需  $N_{RL}^{ours} = M_F \cdot H/U$  次加载，极大减少寄存器开销。
- 示例: D=768, U=128, M\_F=16, H=8 → 原 HNSW 需要 192 次加载，Flash 仅需 1 次。

### • 算术运算: 仅需加法，无需复杂减乘操作。

Evaluation:

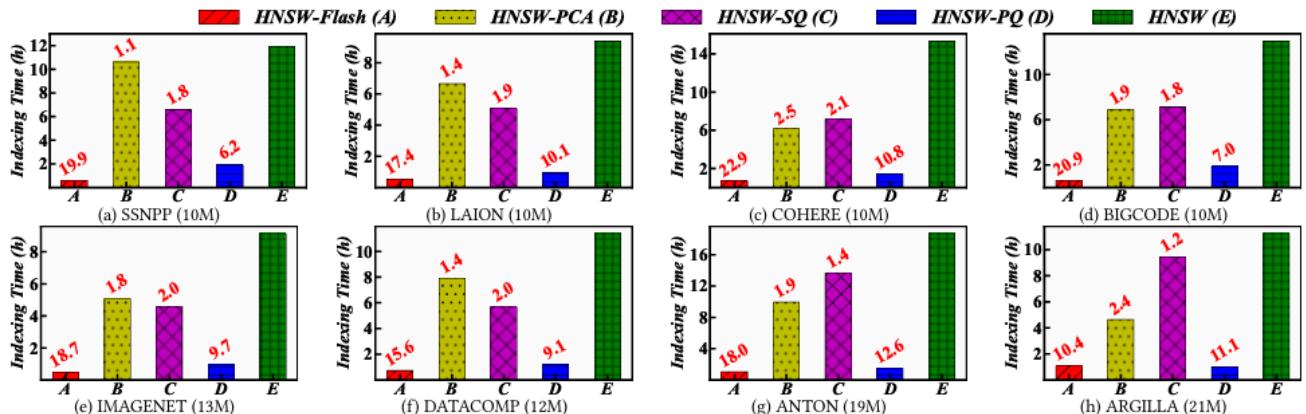


Figure 6: Indexing times for all methods across eight datasets (red values at the top of each bar indicate speedup ratios).

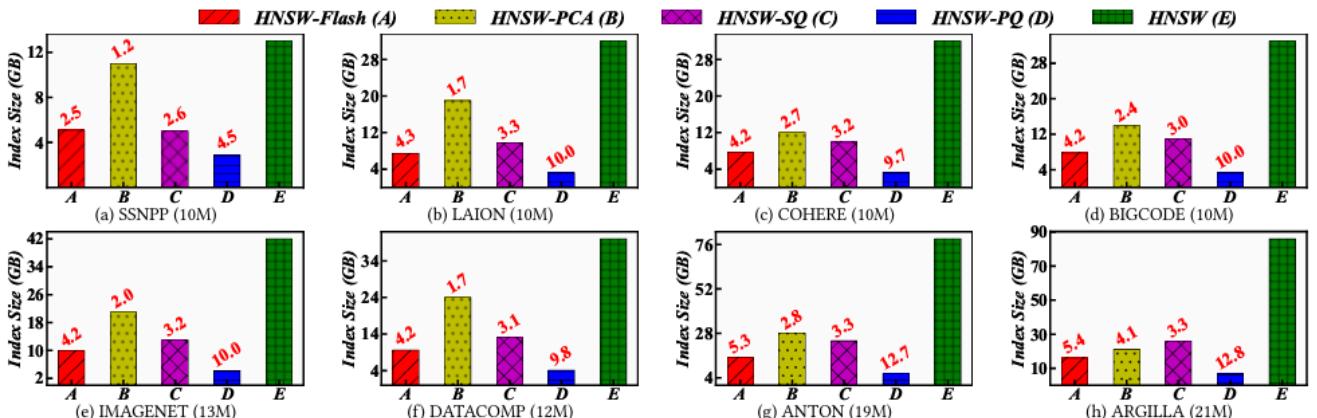


Figure 7: Index sizes for all methods across eight datasets (red values at the top of each bar indicate compression ratios).

构建时间和大小都显著降低

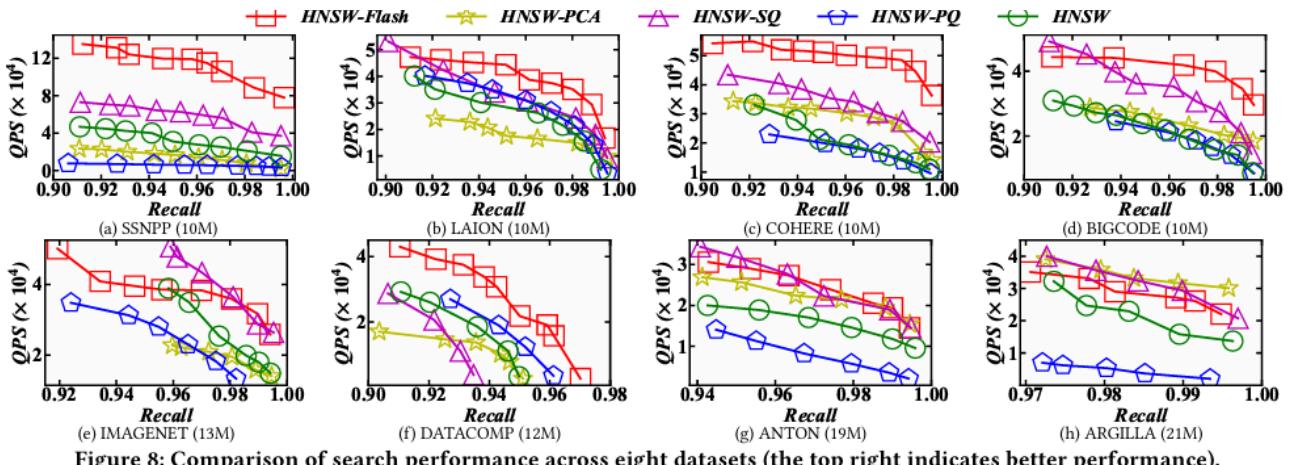


Figure 8: Comparison of search performance across eight datasets (the top right indicates better performance).

同时Recall还保持高水平