

项目

广告算法

布尔召回

关键词召回

多级索引召回

协同过滤

向量召回

DSSM

Youtube DNN

TDM

搜索广告召回

关键词召回

商品定向召回

Retrieval

架构

系统逻辑

Index

物料索引概述

Index背景

流水、index和retrieval关系

流水系统

Index

Retrieval

网盘数据流转

Retrieval

检索过程漏斗模型

检索阶段

过滤阶段

调价阶段

预选阶段

粗排阶段

依赖知识

多级索引

gc_map

Magvir

现状

架构

整体架构

OP架构图

Beam Search

多叉树索引

难点

方案

动态确定聚类类别数K

聚类结束条件

多叉树平衡

索引树验证

SSG

参考
原理
 概要
 SSG
实现
效果
 参数
 数据集测试
 与其他算法比较
 上线
优点
不足
其他向量化检索技术
乘积量化
 索引构建
 检索
 最近邻搜索
IVFPQ-倒排乘积量化
 基本思想
 原理
HNSW
 朴素想法
 NSW算法
 跳表
 HNSW

项目

广告算法

布尔召回

关键词召回

多级索引召回

协同过滤

向量召回

DSSM

Youtube DNN

TDM

搜索广告召回

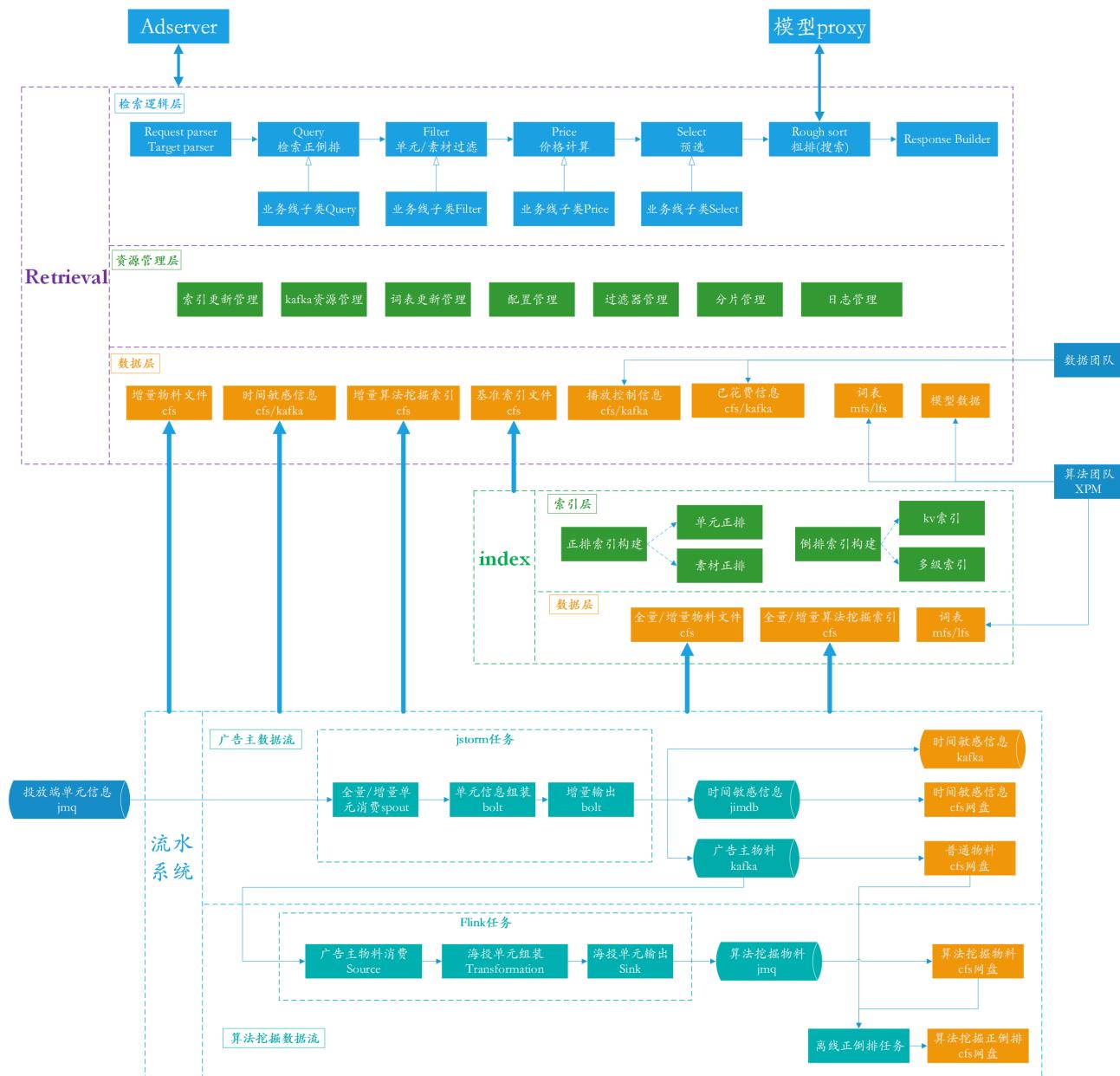
关键词召回

精确匹配、短语匹配、切词匹配、紧密变体、向量召回

商品定向召回

Retrieval

架构



系统逻辑

Index

物料索引概述

Retrieval 是广告播放索引服务，用来根据 AdServer 请求召回符合条件的广告，所有的广告都来自于广告主的投放（直接投放或者算法挖掘），内存中的状态需要实时（或者准实时，以下均称为实时）的反映投放端的广告库状态。

投放端的广告库状态是不断改变的（广告主开启关闭广告单元、修改价格等），这也就需要 Retrieval 的内存状态随之改变，并且当 Retrieval 服务重启时，需要尽可能迅速的将广告库状态加载到内存中。

物料

广告库的数据经流水系统从投放端传递过来，称之为“物料”，物料分为“全量”和“增量”两种

- 全量，可以理解为某个时间点的广告库的 snapshot (快照)，也就是某个时刻广告库里所有广告的状态，大概每天一次，内容较多
- 增量，可以理解为 patch (补丁)，代表每时每刻广告库的状态的改变，大概两分钟一次，也就是说每个增量包括这两分钟之内广告库的状态改变，内容较小

现阶段，全量和增量都是通过文件的形式下发的。

广告库的实时状态，由某次全量以及该次全量之后的所有增量组成。

索引

Retrieval 的内存状态基本上就是索引的状态，索引分为正排和倒排，基本可以理解为两种 map：

- 倒排，映射关系为：广告主的投放条件（以下称为倒排key） -> 广告单元ID（某些倒排直接映射到素材ID，为简化说明，使用单元ID代替）。广告主的投放条件在不同业务上有不同：搜索业务上，广告主通过购买关键词来投放广告，所以倒排key就是关键词；推荐业务上，广告主购买的可能是一个月内浏览过的某个类目，所以倒排key就是类目ID。
- 正排，映射关系为：广告单元ID -> 广告单元信息。广告单元信息就是广告主投放的内容，大概包括以下几类信息：
 - 素材信息，sku 相关的信息，title, cid 等
 - 出价信息，站内站外出价、人群溢价、无线溢价、抢位助手等
 - 播放控制信息，广告投放的广告位、地域、是否匀速播放等
 - 等等

目前可以暂时认为，倒排信息都是通过正排信息生成的。

所以如果 Retrieval 的索引能够反映广告库的实时状态，那也就意味着索引是根据某次全量及之后所有的增量得到的。

Index背景

物料索引处理逻辑不在retrieval上，原因如下

- 性能，如上文所说，全量文件因为包含了整个广告库的数据，所以非常庞大，根据这个文件来生成倒排会占用巨大的计算资源，如果由 Retrieval 来处理全量文件，会严重影响线上服务，并且同一份物料生成的倒排是完全一样的，也没有必要让线上所有的机器都做一次同样的工作

- 实时性，如上文所说，“广告库的状态由某次全量及该次全量之后的所有增量组成”，全量每天一次（假设每天0点），如果 Retrieval 在23:00重启，那需要在处理全量文件后，再加载一整天的增量文件才能提供正确的服务，这会导致较严重的线上问题

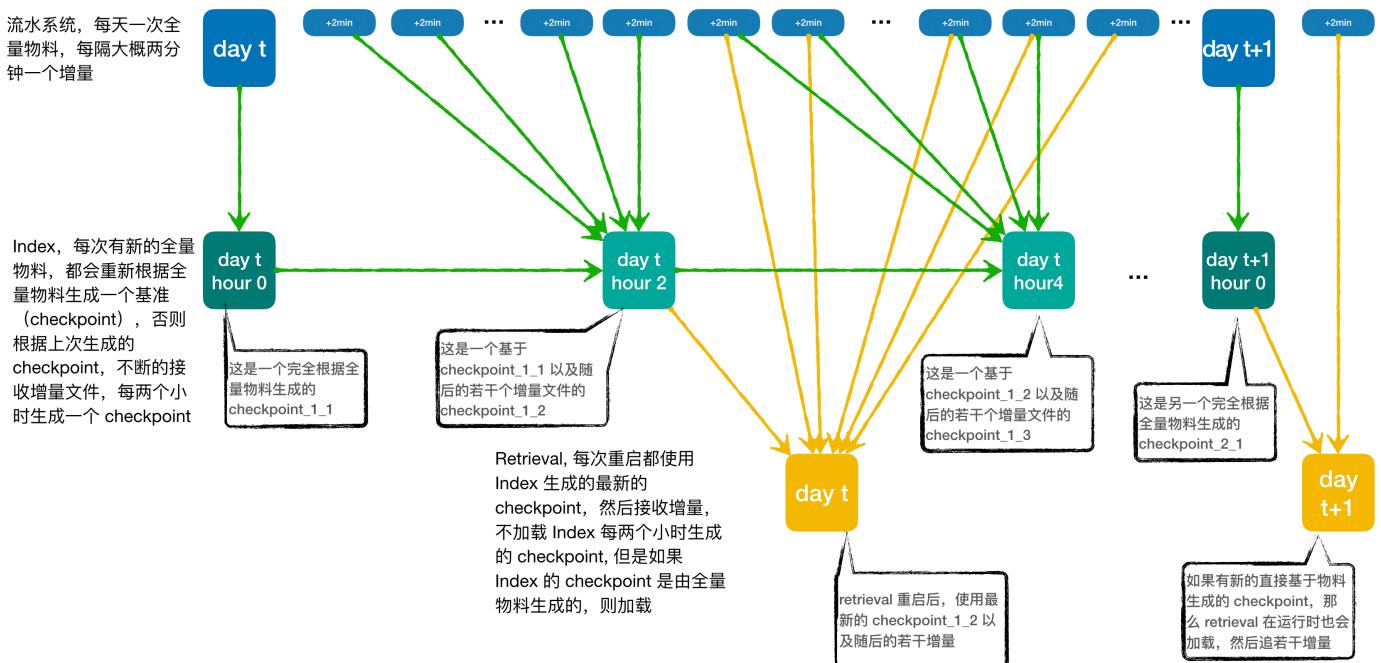
所以 Index 应运而生，Index 会根据物料信息生成正倒排存储到文件中，共供 Retrieval 直接加载使用，除此之外：

- Index 是离线服务，不需要担心处理大型全量文件带来的性能问题
- Index 每隔一段时间（目前是2个小时），生成一个 checkpoint (检查点，基准，dump)，这个 checkpoint 包括上一次全量物料以及截至该时刻的所有增量物料所生成的正倒排

使用 Index，Retrieval 不需要浪费大量计算资源处理全量文件，并且每次重启的时候，可以直接加载最新的 checkpoint，然后最多只需要再加载2个小时的增量文件就可以提供服务。

但是增量文件带来的倒排变化是需要 Retrieval 自己来处理的，当然，Index 也会处理。

流水、index和retrieval关系



流水系统

流水系统每天会生成一次全量物料文件，然后每两分钟生成一个增量文件。

Index

Index 重启时，分为两种情况：

- 重启时发现已经有一个之前的 checkpoint，那么就加载这个 checkpoint，然后接着追增量，然后每两个小时生成一个新的 checkpoint
- 重启时没有之前的 checkpoint，那么就使用上一个全量物料生成一个 checkpoint，然后每两个小时生成一个新的 checkpoint

当 Index 在运行时，流水推了一个新的全量物料时，Index 不再使用之前的 checkpoint，而是使用这个全量物料生成 checkpoint。

Retrieval

Retrieval 重启时，会使用 Index 生成的最新的 checkpoint，加载之后再继续追增量，将所有的增量都追完后，开始提供服务。接下来，Retrieval 就一直使用增量来维护正倒排，不再使用 Index 每隔两个小时生成的 checkpoint。

但是当 Index 生成的 checkpoint 是来自于全新的物料时，Retrieval 会去加载。

什么时候加载 Index 生成的 checkpoint：

可以认为 checkpoint 有两个版本号：

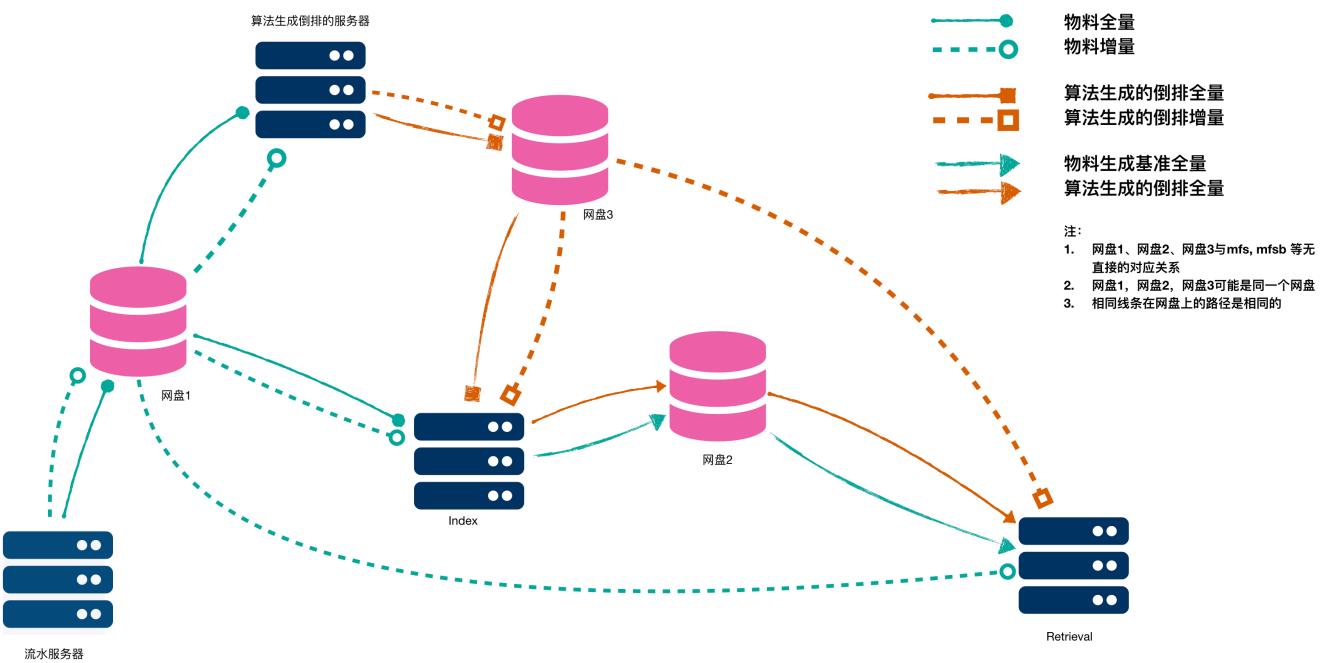
- 一个是大版本号，标记这个 checkpoint 是基于哪次全量物料
- 一个是小版本号，标记这个 checkpoint 在上次全量物料后，加载了多少增量

只有大版本号变化时，Retrieval 才会去加载。

网盘数据流转

目前数据分为两种

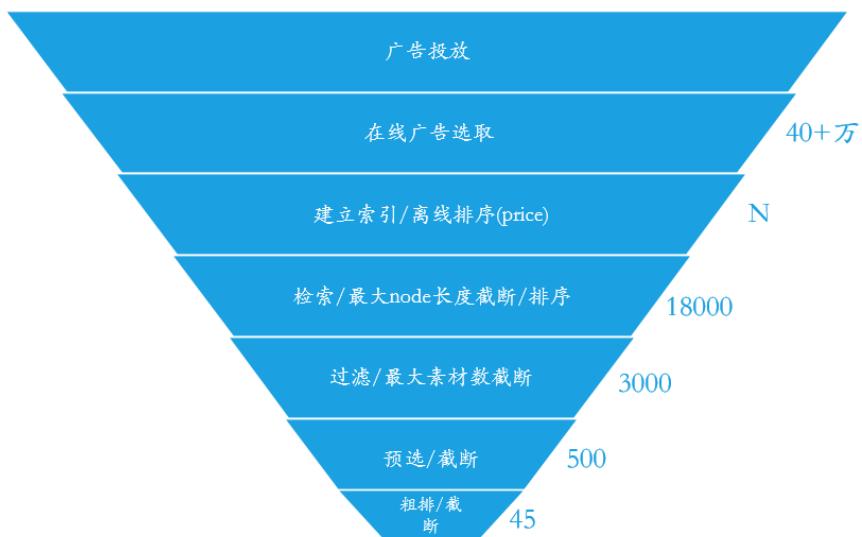
- 一种是倒排都是index和retrieval根据正排生成的
- 另一种是其他系统生成的倒排，比如搜索业务中的智能匹配倒排、海投倒排等。比如智能匹配倒排，是由其他服务生成，同样也分为全量、增量，Retrieval 可以直接加载使用，这个服务位于 Index 的上游，最重要的原因就是：生成这种倒排的服务自己不能提供 checkpoint 服务，所以这种倒排也要经过 Index，从而利用 Index 的第二个功能：定期生成 checkpoint。如果不经过 Index 利用这种特性，那么也会遇到重启后加载大量增量导致不能尽快提供服务的问题。



	input	output
智能匹配倒排服务	物料全量, 物料增量	智能匹配倒排全量, 智能匹配倒排增量
Index	物料全量, 物料增量, 智能匹配倒排全量, 智能匹配倒排增量, 海投倒排全量、海投倒排增量等	checkpoint, 智能匹配倒排 checkpoint
Retrieval	checkpoint, 物料增量, 智能匹配倒排 checkpoint, 智能匹配倒排增量	-

Retrieval

检索过程漏斗模型



检索阶段

1. 查询倒排

1. 普通kv倒排，比如搜索词倒排等，根据请求查询指定倒排。遍历结果链Posting，针对每个倒排节点，截断，构造AdUnit，智能调价（快车最大化转化、LiteGrowth最大化转化等）
2. 查询商品定向engine倒排，遍历结果链，构造AdUnit，过滤，加入到检索队列ad_unit_list中。填充商品定向信息，获取定向条件最高出价
3. 查询智能商品定向倒排

2. 排序检索队列：设置排序类型，包括随机ecpm、bid_price、price等，根据不同排序类型排序

3. 检索阶段

过滤阶段

1. 单元过滤

1. ad_group_state_filter 单元状态过滤（状态kafka）：通过group_id获取时间敏感正排中的时间敏感信息，判断单元状态，是否被过滤
2. time_range_filter 投放时间范围过滤：对计划豁免广告位不进行过滤，其他通过 AdGroupInfo::new_time_range_ 判断

3. group_area_filter投放地域过滤：到家单元不做过滤，通过RetrievalRequest::mixer::GeoInfo中的地域信息在AdGroupInfo::area字段中是否能找到进行判断
 4. vend_pin_filter广告主黑名单过滤：通过广告主id是否在词表vend_limit_dict_new_中判断（userpin黑名单）
 5. 等等，具体可看<https://cf.jd.com/pages/viewpage.action?pagetId=379934796>
2. 素材过滤
1. good_comments_rate_filter好评率过滤：好评率低于当前广告位相同主营类目cid广告的最低好评率时过滤，查词表min_good_comments_rate_dict_，目的是保证商品广告的好评率不能太低
 2. ware_quality_filter商品广告质量过滤：出价低于当前广告位相同主营类目cid广告的出价，查词表min_cid_price_dict_，保证广告质量
 3. query_sku_relevance_filter相关性过滤：计算关键词和sku的相关性，小于阈值则过滤。阈值通过访问配置文件的search_query_sku_relevance_threshold字段获取
 4. keyword_brand_filter搜索关键词品牌过滤：创意品牌id不在关键词预估的品牌id集合中则过滤
 5. space_info广告位过滤：图片请求只能出图片，商品定向只能出商品、店铺、频道
 6. creative_time_range_filter素材播放时间过滤：素材播放时间与素材创建时间无交集
 7. 等等，具体可看<https://cf.jd.com/pages/viewpage.action?pagetId=379934796>
3. 调价：搜索人群溢价，可看调价阶段
4. 创建预选

调价阶段

预选阶段

1. 排序预选阶段，获取截断数目，排序生成大顶堆

粗排阶段

1. 搜索业务访问粗排
2. 对广告队列分片访问粗排predictor

依赖知识

多级索引

gc_map

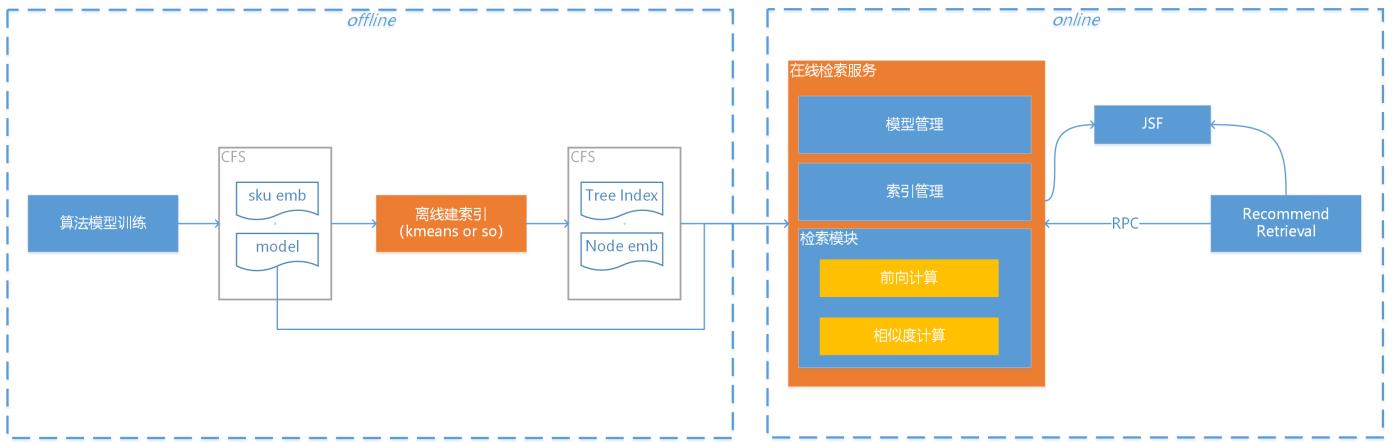
Magvir

现状

目前magvir分4个业务线，搜索magvir当前qps=1w2，机器数目16C+128G共24台，耗时20ms，压测时单机qps最高为800。未上相关性模型时，最高qps=3800。

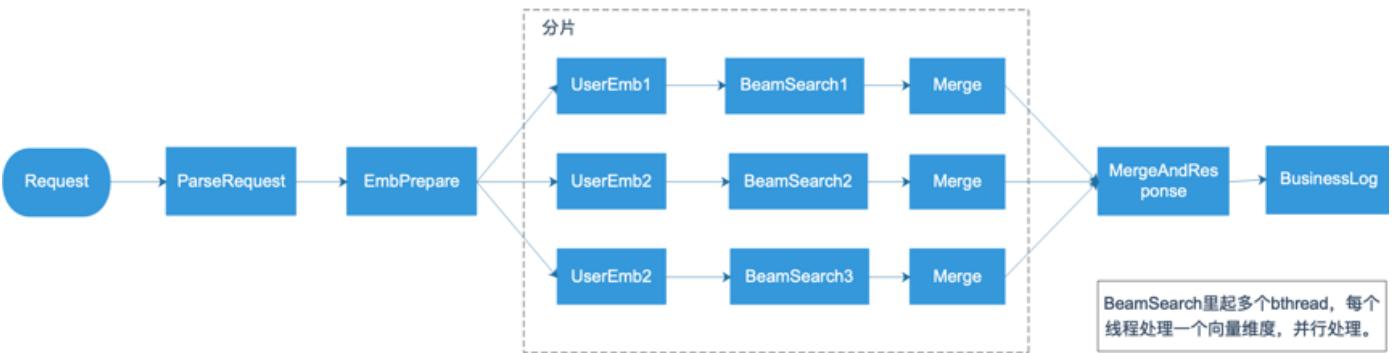
架构

整体架构



1. 算法模型训练，产出embedding数据，推给index
2. index构建索引，推给serve
3. serve加载模型、索引，提供serving能力

OP架构图



玄圃项目使用 brpc 作为对外服务框架。内部执行流使用执行图框架，通过将主要过程抽象为 op 予以串联组成有向无环图。

目前抽象出来的 op 有：

- 请求解析
- 用户向量化
- 检索过程
- 检索结果合并
- 结果构造

其中 request context 中存储 session 级别的数据，global context 存储全局共享数据，两种 context 中的数据可以被 op 按需访问。

Beam Search

beam search检索过程：逐层向下检索，每层保留K个节点并往下扩展，兼具效率和精度

多叉树索引

难点

将索引树升级为多叉索引树，目前存在的难点主要为以下几方面

1. 无法针对数据集明确指定聚类的类别数K
2. 无法确定某节点的结束聚类条件
3. 无法有效平衡多叉树，避免出现链表树，即节点全在左子树的情况

方案

目前想法是根据用户指定的索引树高度Height和某节点中包含的数据数目来解决上述问题

动态确定聚类类别数K

根据用户指定的索引树高度Height和数据集数目node_numbers。通过公式

$$k^{\text{height-level}} \geq \text{node numbers}$$

可计算出该数据在某高度下的聚类类别数目。

比如针对root节点，树高度Height=4，当前level = 1，node_numbers = 100w，可得K为100；则将数据聚为100类左右；

同理类似，假如root节点下某孩子所包含数据数目为81w，其level = 2，则可计算出该孩子的聚类类别数K为900。

聚类结束条件

判断某节点不再进行聚类的条件如下

1. 某节点level达到阈值即Height - 1，则只需向其包含的sku挂到该节点即可
2. 设定簇包含的节点数目阈值，若某节点所包含的sku数目低于该阈值，停止聚类，将其sku挂载到该节点，成为叶子节点（待定，可能会加剧树间高度差）
3. 对某节点进行类别数为K的聚类，发现聚类后的标签数目低于K，则此时有两个选择：a. 结束聚类，将sku挂载到该节点即可；b. 调整类别数K，重新聚类

多叉树平衡

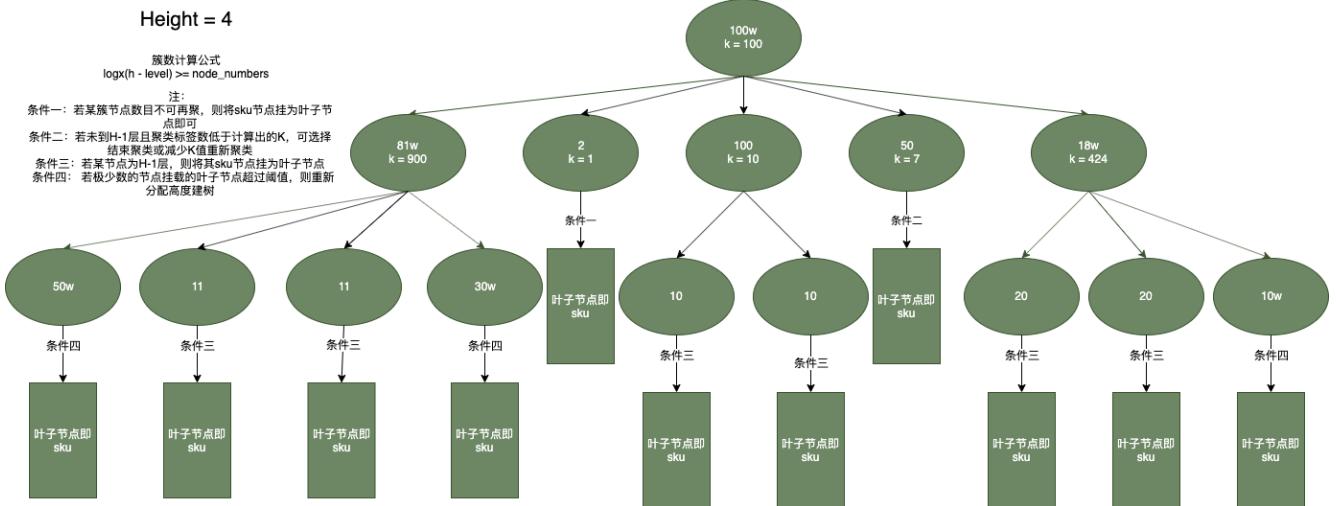
由上所知，多叉树的建树过程通过以下几点保证平衡性

1. 由用户指定高度Height
2. 针对包含不同数据量的节点，保证数据量大的节点的聚类类别数K1 大于数据量小的节点的聚类类别数K2，如图中第二层 81w节点的类别数为900、18w节点的类别数为424以及其余数据量小的98个节点的类别数大约为10左右

索引树验证

索引树正确合理性的验证通过以下机制保证

1. 自验证：当在用户指定的高度下，完成索引树构建后。若存在较少的簇内包含过多的sku节点，即视为该索引树不合理；比如图中50w节点、30w节点以及10w节点，此三个节点包含了90w数据，而剩下的数千个类别仅包含10w个sku数据，则视为该索引树构建不合理，可重新指定高度进行构建
2. diff工具评测召回率；在每次构建索引树后，通过diff工具进行暴力检索和索引召回检索的diff，保证召回率在指定阈值以上，才视为该索引树构建合理



SSG

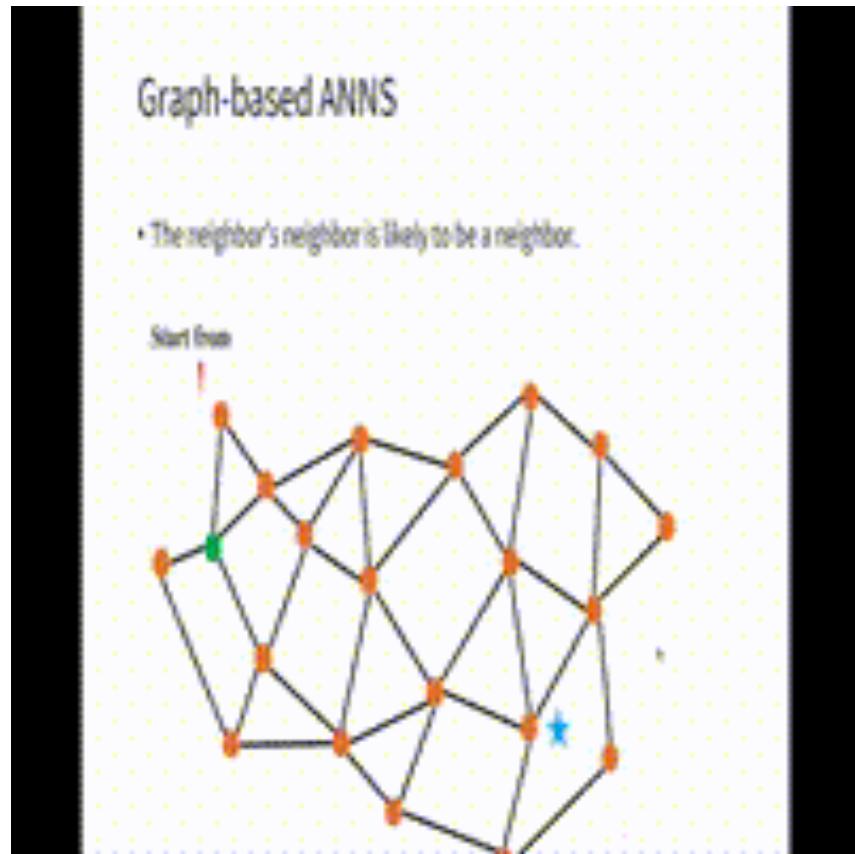
参考

- [efanna_graph](#)
- [Graph Search Engine: A Deeper Dive](#)
- [SSG](#)
- [极度快速的近似最近邻搜索算法\(EFANNA\)学习笔记](#)：强推，降了efanna_graph构建knn图的原理
- [余弦定理](#)

原理

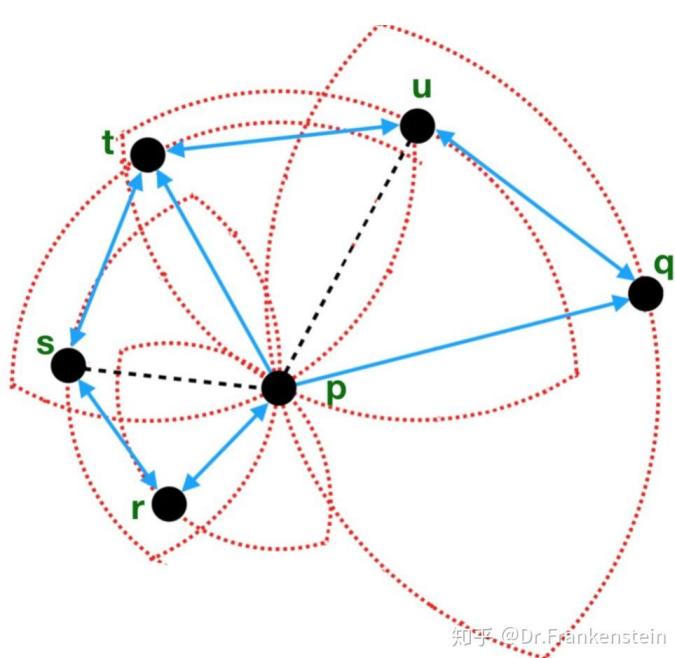
概要

- **朴素图搜索算法：**假设图中蓝色点是待检索的query，在给定有向图中，随机选择一点作为起点（比如图中绿色原点），迭代重复一下过程：将当前点（node）以及它在图上的邻居点与query进行距离计算，从中选择距离query最近的点，作为下一次迭代的核心点，直到找到目标点。具体如下图



可以发现，这是中启发式贪婪算法，每一步仅追求当前最接近query的点，期望从局部最优达到全局最优。但该算法，在一般的图结构上无法保证一定可以搜索到query或query的紧邻点。该算法仅适用于 **MSNET**图结构

- **MSNET理论**
- **NSG**
 - 理论概述



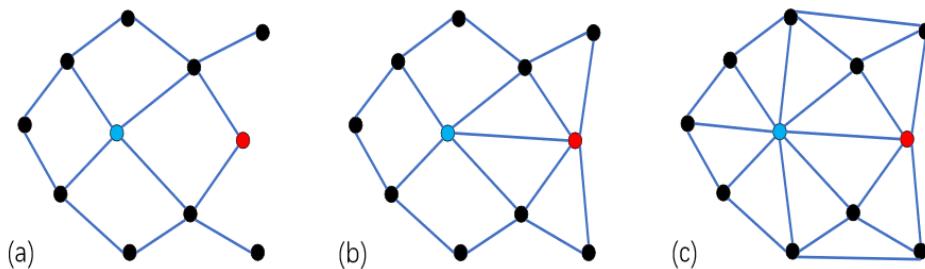
图上的r、s、t、u、q是点p最相似的5个邻居。在K近邻图中，它们都会与p相连。但在我们提出的NSG中，我们优先保留距离p最近的r。对于次近邻s，我们发现边sp是三角形spr的最长边，因此s与p不连接。对于点t，pt虽然是三角形spt的最长边，但由于sp没有连接，那么pt就可以连接。以此规律，我们可以将K近邻图中的较多“冗余边”去除。

被去掉的边为什么都是“冗余”的呢？因为在留下边构成的这个图上，我们能证明接近 $\log N$ 的平均检索复杂度。而且这个图刚好也是保证这一属性的最小图。因为在A* search的每一次迭代，我们都要检查当前点的*所有邻居*来找到离query最近的那个，因此，图的平均出度（out-degree）越低，检索越快。

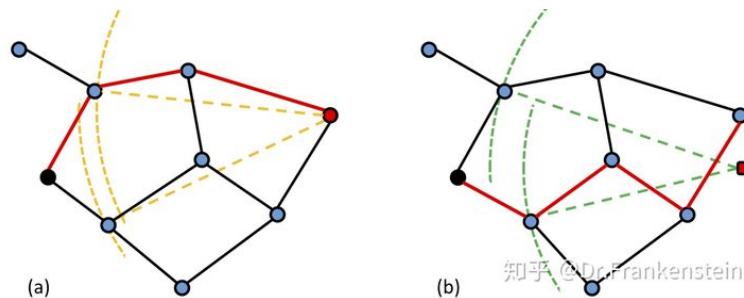
- 优点：重振了MSNET理论，性能最先进

- 缺点：

- NSG图过于稀疏，图的平均出入度很小，搜索算法需要绕路，到了搜索时间更长，性能偏离了潜在的最佳位置；下图计算次数分别为7, 5, 8



- NSG图索引构建时间复杂度高，限制了方法效率和可伸缩性
- NSG图理论存在漏洞，其前提是被搜索的点是存在于数据库中的。而在相似性检索领域，被搜索的点可能不在于数据集中，比如以图搜图等。而这样，就会导致搜索效率下降。如图



- 左边 (a) 表示搜索数据库内某点（红色圆点），因为是存在于数据库的点，建立索引时会与其他点有边连接。假设搜索起点是黑色点，根据上述贪婪搜索算法，上方邻居离红点近，搜索路径会是上方红色边构成的路径。
- 右边 (b) 表示搜索数据库外某点（红色方块），不存在与数据库中，不存在与其相连的点，同时注意到与红色方块最近的点也是右上方的原点，即 (a)、(b) 搜索目标点是同一个点。但在 (b) 图中，选择路径是下方红色边所组成的路径。
- (a)、(b) 对比，起始点、目标点相同，但路径不同。(a) 是当前格局最优路径，路程短，距离计算次数少，计算量小。总结来说 NSG 的理论保证仅仅对 (a) 情况有效，对 (b) 无效

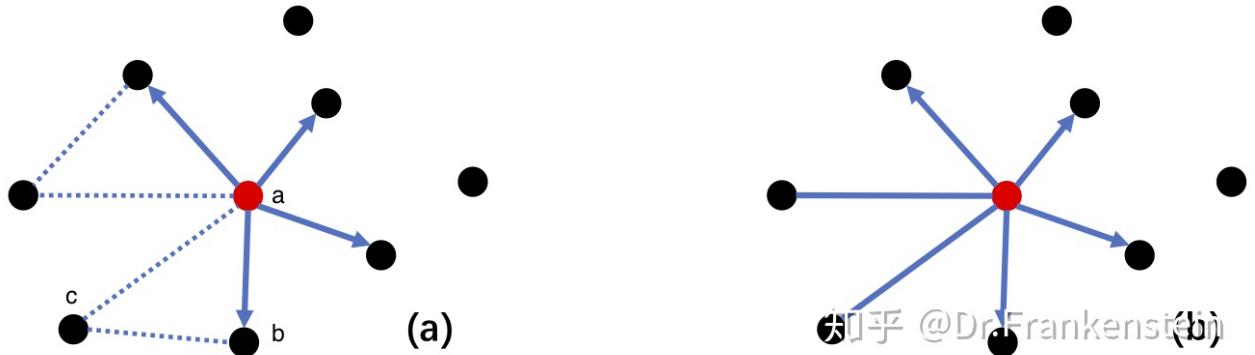
- SSG**：来自于通信卫星系统消息传输机制，发明了新的MSENT系列SSG，设计了一种边选择策略（通过在每个节点的任意两个邻边之间强制最小约束），充分继承了MSNET卓越的ANNS属性，保持了均匀分布的邻域，可以使得每个节点和其邻域的连接有效，确保了图的有效搜索路由

SSG

- NSG缺点**：NSG的缺点在上文提到了，那么SSG是如何解决这些问题的呢？

- 问题一：NSG图过于稀疏，导致搜索效率下降。针对此问题，SSG的图形理论定义发生了变化，SSG通过参数（描述邻边间的最小角度）来控制图形的稀疏度，并且能够探索图的最佳出度
- 问题二：NSG图索引构建时间复杂度高。针对此问题，SSG设计了一种新型的索引算法NSSG，以近似于SSG（构建精确的SSG非常耗时），NSSG的索引算法时间复杂度低，有利于大型应用

- 问题三：NSG图选边策略没有考虑角度约束，无法保证Not-In-DataBase情况搜索路径最优。如下图所示，NSG中边选择策略是：以左图为例，三角形abc中，ab已经连接的情况下，ac因为是最长边，因此不能被连接。如此一来，以a为中心点的邻域，左下方就出现了巨大缺口。假设某搜索路径达到了a，期望向左下方前进时，就需要从别的边绕路，导致搜索路径延长。要注意，这只是二维情况，在上百维的高位空间中，NSG会出现各种缺口。



如何避免呢？自然是使用更合理的选边策略：使用角度，让边与边之间按一定角度均匀分布在邻域各个方向，类似于卫星图。

- 定义：

- 单调图：**假设存在图G，其节点集合为S，节点数量n，数据维数d。基于图的ANNS使用 算法一 从开始点 p 搜索，直到目标点 q 。若在搜索过程中的每次迭代中，算法1越来越接近目标点 q ，那么就说图是 单调的。并且 所有单调图都属于MSNET系列
- 定义

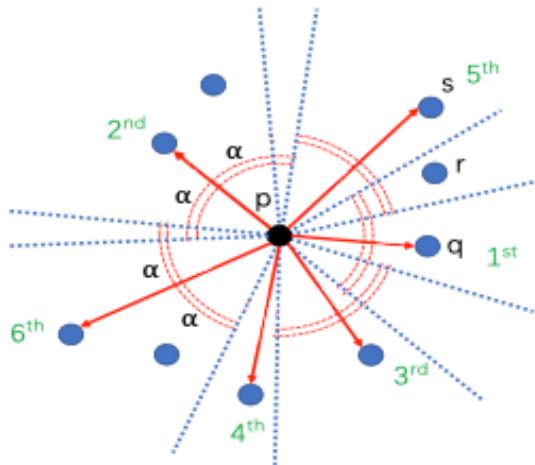
for any edge pq , $pq \in SSG$ if and only if $Cone(pq, \alpha) \cap B(p, \delta(p, q)) \cap S = \emptyset$ or $\forall r \in Cone(pq, \alpha) \cap B(p, \delta(p, q)) \cap S$, $pr \notin SSG$, where $B(p, \delta(p, q))$ is defined as the open sphere centered at p with radius $\delta(p, q)$, and $\delta(p, q)$ is the distance between p and q . $\alpha \leq 60^\circ$

$pq \in SSG$ 当且仅当以 pq 为轴， α 为角直径的圆锥和以 p 为圆心， pq 为半径的球体和有向边集 S 的交集为空或者该交集中存在点 r ，但 $pr \notin SSG$ ，

$\delta(p, q)$ 表示 pq 之间的距离， $B(p, \delta(p, q))$ 表示以 p 为圆心， pq 为半径的开球， $\alpha \leq 60^\circ$ ，
 $Cone(pq, \alpha)$ 表示在欧几里得空间中，对于任意

- NSSG步骤：**遍历 S 中的每个点， $p \in S$ ，执行如下操作，构建 SSG

- 对于每个点 $q \in S - \{p\}$ ，计算 pq 的长度
- 以长度 pq 的升序对集合 $q \in S - \{p\}$ 进行排序，从而形成列表 C 。
- 对于 $pq \in C$ 中的每个边，检查是否可以根据 SSG 定义按顺序将 pq 添加到 SSG 。



- 距离分析，在2维空间中，首先计算与p点的距离，并升序排序。按顺序检查有向边是否可以加入SSG。
 - 首先看q点， $Cone(pq, \alpha) \cap B(p, \delta(p, q)) \cap S = \emptyset$ ，所以有向边pq加入SSG。其他的以此类推
 - 再看r点，检查它的顺序肯定在q之后，s之前。 $q \in Cone(pr, \alpha) \cap B(p, \delta(p, r)) \cap S \neq \emptyset$ ，且属于SSG，所以不能加入SSG。
 - 看s点， $r \in Cone(ps, \alpha) \cap B(p, \delta(p, s)) \cap S \neq \emptyset$ ，但，所以s可以加入SSG。

● 效果

○ In-DataBase Search

实验证明，虽然NSG平均出度较小，但平均搜索路径更长。SSG可通过边夹角选边，使得边邻域分布更加均匀，减少空白。并且可以通过改变边缘之间的最大夹角 α 可以控制图的稀疏程度，由此可以根据不同的数据分布调节 α ，并找到搜索路径长度和出度的平衡点。

○ Not-In-DataBase Search

根据定理，证明当 $\alpha \leq 30^\circ$ 时，SSG的搜索路径一定是单调的。而NSG没有角度约束，故在非数据库点情况下，无法保证单调性

● NSSG构建步骤：NSSG（导航SSG），主要思想是

1. 从局部邻域中选择邻居，而非从全部节点中选择
2. 在节点的候选邻居集中应用SSG边选择策略
3. 保证图的连通性（确保每个节点都可到达）

○ 截剪候选集

- 实验表明：一个小的局部邻域可以为每个节点覆盖足够的有效邻域，所以不需要对每个节点都搜索全部节点
- 故为了降低搜索构建复杂度，腰围每个节点收集一小组的有效邻居。因为，从每个节点的局部邻域中选择邻居。为了达到这个目的
 - 首先建立一个具有小k $k \ll n$ 的knn图
 - 收集当前节点的邻居，及其邻居的邻居来扩展候选集，这比NSG的搜索收集方法快得多

○ 有效边选择

- 因为有效边聚集在一个较小的邻域中，因此NSSG的关键是继承每个节点本地邻域的结构。故需要高精度的KNN图，然后对上述方式收集到的候选邻居集应用SSG边选择策略，以近似复制SSG的局

部结构

- 注意，需要设置最大度数限制，避免少数点的度数爆炸

○ 保证图连通

- NSG中通过固定开始搜索节点来保证图的连通性
- DFS扩展算法可为图添加少量边，保证图的单向连通性。主要思想是从选定的节点生成DFS树，并尝试将所有其他节点连接到树，选定的节点被称为导航点，并从导航点开始搜索
- SSG随机选择m个导航点 ($m << n$)，并通过DFS扩展算法增强了所有节点的连通性。由于随机选择，导航点可均匀分布在数据集中。
- NSSG比NSG能更好的处理具有不同性质或分布的数据

● 检索

- 同样基于贪婪搜索，相比NSG，NSSG是计算m个导航点和查询点q的距离，从小打到排序，选择最近的导航点开始查询

实现

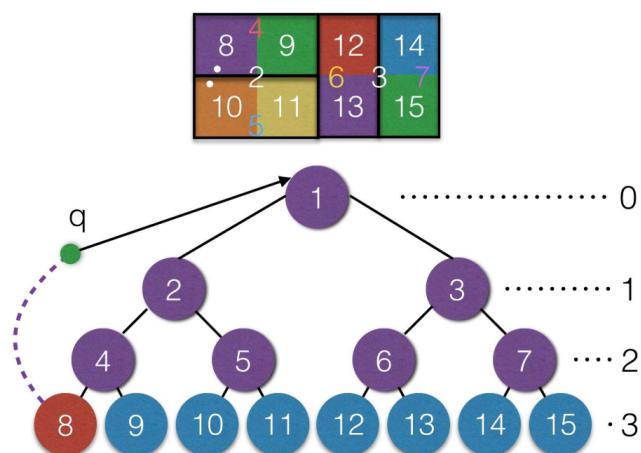
1. 构建KNN图

1. 初始化knn图

1. 老版使用随机初始化knn图：随机初始化，针对每个图节点，随机选取固定数目个邻居

2. 新版使用kd-tree构建knn图：

1. 采用分治的策略来快速获得一个初始化的kNN图。得到一个初始化的kNN图无非就是给基数据集中的每个点初始化邻居，这个过程其实就是在已经建好的几个随机截断KD树上为数据点找邻居，在这里作者采用的方法不过是“分层分治”的策略。



2. 对数据集中的每一个数据点，我们都要给它们初始化邻居，对某个数据点初始化邻居时，我们要在已经建立的所有树上以该数据点为查询点进行分层分治查找，也就是说在每棵树上都要查找一遍。下面只详细分析在一棵树上的分层分治过程，多棵树只是重复的过程。

1. 为了便于分析，下面以上图示例说明。现在要给数据点 q 找邻居（在树中找一些离它较近的数据点），从树根开始执行深度优先搜索进行查找，一直找到标号为 8 的叶子结点，显然 q 也处于这个结点中（整棵树涵盖了所有的数据点），这个叶子结点中的数据点加入到数据点 q 的邻居候选集中，此时我们处理的是最底层（深度为3）。标号为 1、2 和 4 的结点（非叶结点）是搜索的过程中遍历到的结点。接着该处理倒数第2层（深度为2）了，我们只处理该层中遍历到的结点，也就是结点 4 了，其它的标号为 5、6 和 7 的结点都没有遍历到。结点 4 的孩子结点分别为结点 8 和 9，因为结点 8 已经遍历过了，此时我们只处理没有遍历过的孩子结点，因此，我们只处理结点 9，对

以结点 9 为根的子树进行深度优先搜索直到某个叶子结点，因为结点 9 本身就是叶子结点，因此返回的叶子结点也就是它了，把该叶子结点中的数据点加入到数据点 qq 的邻居候选集中。接着处理第1层，同理选中结点 2，然后同理选中它的孩子结点 5，对以结点 5 为根的子树执行深度优先搜索，将返回叶子结点 10（上面的局限图可以看出结点 10 离 qq 更近），同样把该叶子结点中的数据点加入到数据点 qq 的邻居候选集中……

2. 上面的那个过程可以一种处理到第0层，即根结点。但是，实际应用时这个过程是很耗时的，因为我们要对多个树重复上述过程，因此，都要处理到根结点是没必要的。在此，可以把能处理到的最小层设定为一个参数 Dep ， Dep 具体值的设定可根据精度和速度的权衡来安排。
3. 明白了上述过程，我们也就很容易明白作者给该算法命名为分层分治的原因了。

2. nnDescent精致化knn图方法

1. NNDescent精致化knn图：精致化算法是在初始kNN图的基础上进行的，精致化之后将得到结果图 G 。

1. 一开始先将结果图初始化为预先建好的初始kNN图，对数据集中的每一个点，它在 G 中会有一定量的邻居（此时就是初始kNN图中它的邻居），将它的这些邻居之间互相添加为各自的邻居（添加到 G 中），添加后将新添加的邻居标记为new，对数据集中的所有点都执行上述过程后，对每个点保留其最近的一定量个邻居（预先设定的上界），这便是第一次迭代的过程，也是相对简单的一次。
2. 第二次迭代。对数据集中的每一个点，它在 G 中会有一定量的邻居（此时的邻居有新添加标记为new的，也有初始kNN图中没标记为new的记其标记为old），将它的这些标记为new的邻居（遍历过之后取消new标记记其标记为old）之间互相添加为各自的邻居（同样添加到 G 中），添加后将新添加的邻居标记为new，不仅如此，对每个标记为new的邻居，还要将所有标记为old的邻居添加到它的邻居中，同时也添加反向边（标记为new的点也要添加到标记为old的点的邻居中），对数据集中的所有点都执行上述过程后，对每个点保留其最近的一定量个邻居（预先设定的上界）。
3. 接下来的各次迭代就和第二次类似了，可以预先设置一个合适的迭代次数。

2. 构建SSG图：

1. 获取kNN邻居集合：获取某节点的邻居节点和邻居的邻居节点，共获取 L 个候选邻居

```
void IndexSSGGraph::get_neighbors(const unsigned q, const Parameters &parameter,
                                    std::vector<Neighbor> &pool) {
    boost::dynamic_bitset<> flags{nd_, 0}; // 是否已加入标识
    unsigned L = parameter.Get<unsigned>("L"); // 候选邻居数目阈值
    flags[q] = true;
    for (unsigned i = 0; i < final_graph_[q].size(); i++) { // 第一种循环 q点的邻居
        unsigned nid = final_graph_[q][i];
        for (unsigned nn = 0; nn < final_graph_[nid].size(); nn++) { // 第二重循环，q点的
            neighbors的邻居
            unsigned nnid = final_graph_[nid][nn];
            if (flags[nnid]) continue;
            flags[nnid] = true;
            float dist = distance_->compare(master_item_vec_[q]-
>emb_node().embedding_vec().data(),
                                         master_item_vec_[nnid]-
>emb_node().embedding_vec().data(),
                                         dimension_);
```

```

        pool.push_back(Neighbor(nnid, dist, true)); // 加入到pool中
        if (pool.size() >= L) break; // 候选邻居数目超过L, 则跳出循环
    }
    if (pool.size() >= L) break;
}
}

```

2. 选边

1. 继承原有邻居，将q点的所有不存在与候选邻居集pool中的一度邻居，全部保存到pool中。在刚才的函数 `get_neighbor` 中，因为要保存一度邻居和二度邻居，还有候选邻居集数目 `L` 限制，可能遍历完全部一度邻居，所以这里做个补充。
2. 排序当前候选邻居集
3. 选边，遍历候选邻居pool，对每个待加入的新边，都计算与所有的已加入的边的cos_ji值，若超过阈值则跳过，否则加入，继续遍历。比较公式是余弦定理 $\cos(C) = \frac{a^2+b^2-c^2}{a \times \sqrt{a \times b}}$ ，计算两条边的夹角cos值，若大于阈值 $\text{acos}(-1) = 3.1415$ ，则排除。否则保留该边
4. 保存到结果数组 `cut_graph_` 中

```

void IndexSSGGraph::sync_prune(unsigned q, std::vector<Neighbor> &pool,
                                const Parameters &parameters, float threshold,
                                SimpleNeighbor *cut_graph_) {
    unsigned range = parameters.Get<unsigned>("R"); // 获取出度阈值
    width_ = range;
    unsigned start = 0;

    boost::dynamic_bitset<> flags{nd_, 0};
    for (unsigned i = 0; i < pool.size(); ++i) {
        flags[pool[i].id] = 1;
    }
    // 继承全部一度邻居，防止get_neighbors函数中未继承完全部一度邻居
    for (unsigned nn = 0; nn < final_graph_[q].size(); nn++) {
        unsigned id = final_graph_[q][nn];
        if (flags[id]) continue;
        float dist = distance_->compare(master_item_vec_[q]-
            >emb_node().embedding_vec().data(),
            master_item_vec_[id]-
            >emb_node().embedding_vec().data(),
            dimension_);
        pool.push_back(Neighbor(id, dist, true));
    }

    // 排序候选邻居集合
    std::sort(pool.begin(), pool.end());
    std::vector<Neighbor> result;
    if (pool[start].id == q) start++;
    result.push_back(pool[start]);

    // 选边
    while (result.size() < range && (++start) < pool.size()) {

```

```

auto &p = pool[start];
bool occlude = false;
for (unsigned t = 0; t < result.size(); t++) {
    if (p.id == result[t].id) {
        occlude = true;
        break;
    }
    float djk = distance_->compare(master_item_vec_[result[t].id]-
>emb_node().embedding_vec().data(),
                                         master_item_vec_[p.id]-
>emb_node().embedding_vec().data(),
                                         dimension_);
    float cos_ij = (p.distance + result[t].distance - djk) / 2 /
        sqrt(p.distance * result[t].distance); // 余弦定理: cos(C)
    = (a^2 + b^2 - c^2) / 2 * sqrt(a * b)
    if (cos_ij > threshold) {
        occlude = true;
        break;
    }
}
if (!occlude) result.push_back(p);
}

// 保存到cut_graph_中
SimpleNeighbor *des_pool = cut_graph_ + (size_t)q * (size_t)range;
for (size_t t = 0; t < result.size(); t++) {
    des_pool[t].id = result[t].id;
    des_pool[t].distance = result[t].distance;
}
if (result.size() < range) {
    des_pool[result.size()].distance = -1;
}
}
}

```

3. 交叉选边? 遍历所有图节点, 以该节点为入口, 遍历该节点的邻居集, 验证邻居中的邻居集是否包含该入口节点。不包含且邻居集数目超过度数阈值, 则对该邻居的邻居集进行选边。否则将该入口节点加入到其邻居的邻居集中。话说这样做的目的是啥呢? 搞不懂

```

void IndexSSGGraph::InterInsert(unsigned n, unsigned range, float threshold,
                                std::vector<std::mutex> &locks,
                                SimpleNeighbor *cut_graph_) {
    SimpleNeighbor *src_pool = cut_graph_ + (size_t)n * (size_t)range; // 节点n的
    邻居集数组
    for (size_t i = 0; i < range; i++) {
        if (src_pool[i].distance == -1) break; // 距离为-1之后的所有邻居都不处理

        SimpleNeighbor sn(n, src_pool[i].distance);
        size_t des = src_pool[i].id;
    }
}

```

```

SimpleNeighbor *des_pool = cut_graph_ + des * (size_t)range; // des表示节点
n的邻居, des_pool表示des节点的邻居集

std::vector<SimpleNeighbor> temp_pool;
int dup = 0;
{
    LockGuard guard(locks[des]);
    for (size_t j = 0; j < range; j++) {
        if (des_pool[j].distance == -1) break;
        if (n == des_pool[j].id) { // 若des节点的邻居集中存在节点n, 则设置dup=1
            dup = 1;
            break;
        }
        temp_pool.push_back(des_pool[j]);
    }
}
if (dup) continue; // 跳过des节点, 不需要处理

temp_pool.push_back(sn);
if (temp_pool.size() > range) { // 若邻居集超过度数阈值 range
    std::vector<SimpleNeighbor> result;
    unsigned start = 0;
    std::sort(temp_pool.begin(), temp_pool.end());
    result.push_back(temp_pool[start]);
    while (result.size() < range && (++start) < temp_pool.size()) { // 选边
        auto &p = temp_pool[start];
        bool occlude = false;
        for (unsigned t = 0; t < result.size(); t++) {
            if (p.id == result[t].id) {
                occlude = true;
                break;
            }
            float djk = distance_->compare(master_item_vec_[result[t].id]-
>emb_node().embedding_vec().data(),
                                         master_item_vec_[p.id]-
>emb_node().embedding_vec().data(),
                                         dimension_);
            float cos_ij = (p.distance + result[t].distance - djk) / 2 /
                sqrt(p.distance * result[t].distance);
            if (cos_ij > threshold) {
                occlude = true;
                break;
            }
        }
        if (!occlude) result.push_back(p);
    }
    // 保存选好的边
    LockGuard guard(locks[des]);
    for (unsigned t = 0; t < result.size(); t++) {

```

```
        des_pool[t] = result[t];
    }
    if (result.size() < range) {
        des_pool[result.size()].distance = -1;
    }
}
} else { // 若邻居集数目为超过度数阈值 range, 则将n节点直接加入到des节点的邻居集中
    LockGuard guard(locks[des]);
    for (unsigned t = 0; t < range; t++) {
        if (des_pool[t].distance == -1) {
            des_pool[t] = sn;
            if (t + 1 < range) des_pool[t + 1].distance = -1;
            break;
        }
    }
}
}
```

3. 保证图的连通性：代码写的是DFS_Expand(), 为啥感觉算法是BFS呢？

1. 从所有图节点中，随机选择 `n_try` 个导航点
 2. 遍历所有导航点，对每个导航点都做如下操作
 1. 将当前导航点加入队列，以当前导航点开始搜索整个SSG图，并设置当前导航点的visited标志
 2. while (true)循环，直到当前导航点下，所有点都连通
 1. while (!myqueue.empty())循环
 1. 将队首元素出队，将其并未遍历过的孩子节点都入队，并设置visited数组，表示已遍历过，不走回头路。
 2. 遍历完整个图后。再统计是否存在图节点未遍历过，若存在，则将该节点加入到图中。加入操作是遍历所有图节点，找到一个连通的，并且其出度小于阈值（即邻居数目低于阈值）的图节点，将该未连通节点加入到这个节点的邻居中
 3. 重复上述操作，直到当前导航点下，所有点都连通
 3. 然后再循环下一个导航点

```
void IndexSSGGGraph::DFS_expand(const Parameters &parameter) {
    unsigned n_try = parameter.Get<unsigned>("n_try"); // 导航点个数
    unsigned range = parameter.Get<unsigned>("R"); // 邻居数目，即出度

    // 随机选择导航点
    std::vector<unsigned> ids(nd_);
    for(unsigned i = 0; i < nd_; i++){
        ids[i] = i;
    }
    std::random_shuffle(ids.begin(), ids.end());
    for(unsigned i = 0; i < n_try; i++){
        eps.push_back(ids[i]);
    }
}
```

```

}

// dfs 图扩展

#pragma omp parallel for
for(unsigned i = 0; i < n_try; i++){
    unsigned rootid = eps_[i]; // 选择导航点
    boost::dynamic_bitset<> flags{nd_, 0}; // 设置是否已遍历的标志数组
    std::queue<unsigned> myqueue; // 队列
    myqueue.push(rootid);
    flags[rootid]=true;

    while(true){
        // 遍历整个图
        while(!myqueue.empty()){
            unsigned q_front = myqueue.front();
            myqueue.pop();
            for(unsigned j = 0; j < final_graph_[q_front].size(); j++){
                unsigned child = final_graph_[q_front][j];
                if(flags[child])continue;
                flags[child] = true;
                myqueue.push(child);
            }
        }
    }

    // 查找未连通的节点
    unsigned unconnect_point;
    bool connect_flag = true;
    for(unsigned j=0; j<nd_; j++){
        if(flags[j])continue;
        unconnect_point = j;
        connect_flag = false;
        break;
    }
    if (connect_flag) break;
    // 连通节点
    for (unsigned j = 0; j < nd_; ++j) {
        if(flags[j] && final_graph_[j].size()<range){
            final_graph_[j].push_back(unconnect_point);
            break;
        }
    }
    myqueue.push(unconnect_point);
    flags[unconnect_point] = true;
} //while(true)
}
}

```

效果

参数

- knn_graph_k: 当前节点的K个最近邻居, 当前选择200
- knn_graph_l: 选取当前节点最近邻的候选个数, 200
- knn_graph_iter: 迭代次数, 12
- knn_graph_s: 节点初始化时, 随机选取的邻居个数, 10
- knn_graph_r: 在一次迭代中添加到候选集的个数, 100
- knn_number_control, 200
- ssg_l: 当前节点候选集大小, 100
- ssg_r: 控制当前节点出度大小, 50
- ssg_angle: 控制选边角度, 60
- ssg_n_try: 导航点个数, 10

论文说 $\alpha = 60^\circ$ 效果最好。我们采用了参数

数据集测试

实验机器: 8C 32G

数据集	构建索引耗时	检索平均耗时	检索tp99	top100召回率	距离评估次数	索引文件大小
SIFT1M (100w * 128)	425s	3.7ms	4.2ms	99.97%	-	154M
GIST1M (100w * 960)	3700s	2ms	2.9ms	99.7%	-	137M
搜索 (2kw * 128, 不分片)	22.5h	3.47ms	5.6ms	99.9%	-	7.6G
搜索 (2kw * 128, 分29w分片)	3h	2.3ms	4.45ms	98.5%	14k	11G
多叉树 (2kw * 128, 分片)	30m	-	5.7ms	99.9%	47k	-

与其他算法比较

索引类型	检索tp99	Recall@100
ssg	7.86ms	0.999
lvfpq	10.96ms	0.9645
Hnsw	9.74ms	0.98

上线

展点消, 正常波动, 线上服务耗时降低 1ms。

优点

1. SSG确保节点之间的连通性，通过调节节点的出度，适应不同高维数据在空间的分布，优化检索路径，减少绕路计算
2. SSG使用多导航点的方式能够更加快速的定位到待查询query的子区域，减少不必要的计算
3. 从实验结果可以看出，与KMeans聚类树相比较，SSG的距离评估次数减少了3倍，有效的节省了计算算力，在检索耗时上，SSG tp99下降1.2ms左右，提升了检索效率

不足

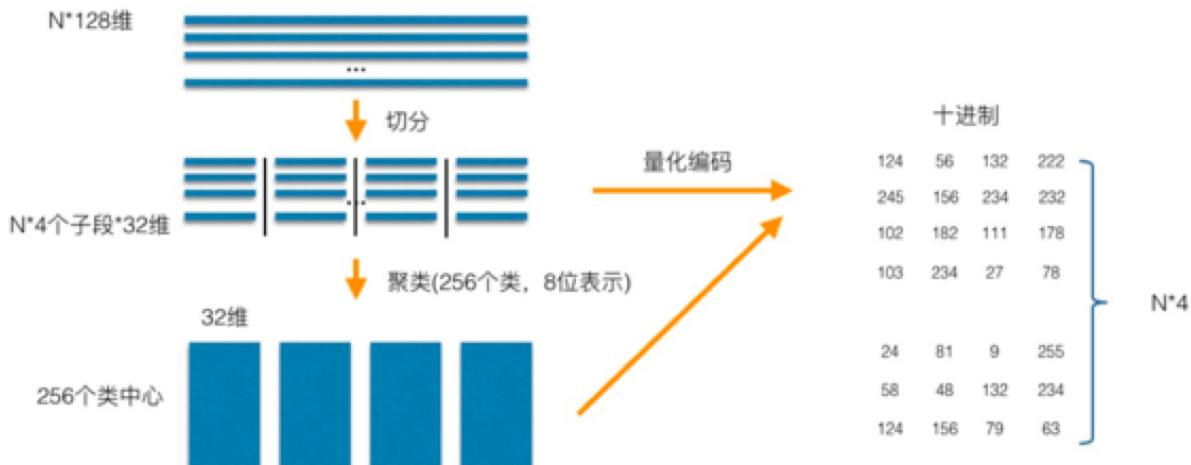
搜索构建时间还是太久。故后面上了kd-tree优化

	索引构建耗时	内存消耗	召回准确率
kd-tree	80m	20%	0.9866
随机knn图	120m	40%	0.9851

其他向量化检索技术

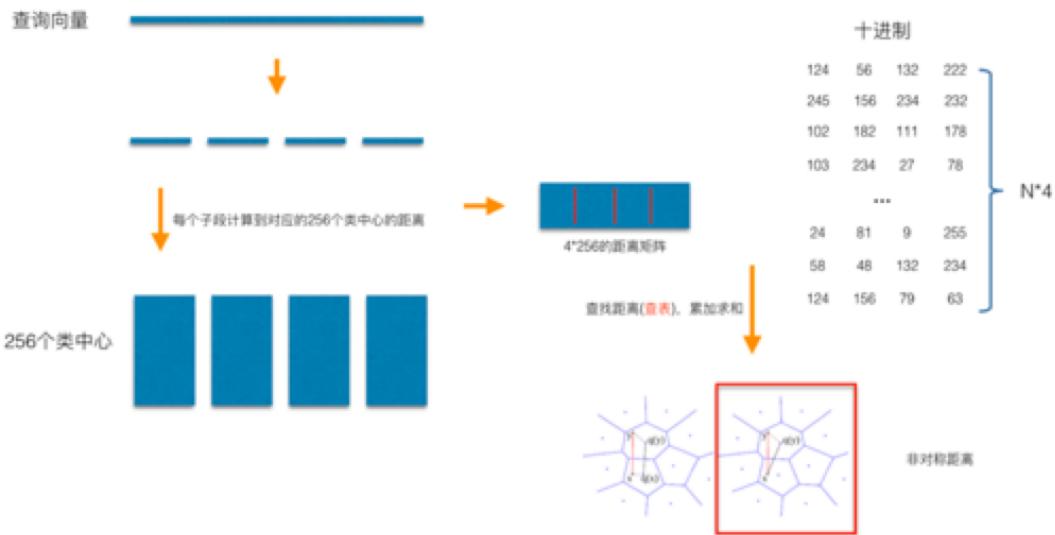
乘积量化

索引构建



1. 训练阶段：针对数据集 $N * 128$ ，切分向量空间为4个子空间，则子空间维度为32维，然后分别对每个子空间进行聚类(图例聚类为256类) 得到子空间码本。这样向量子段都可使用聚类的类别id近似，向量可量化为4个聚类id近似表示。
2. 码本：子空间中的256个聚类中心构成该子空间的码本。子空间码本的笛卡尔积就是原始D维向量对应的码本。用 C_j 表示子空间 j 的码本，那么原始D维向量对应的码本就是 $C = C_1 \times C_2 \times C_3 \times C_4$ ，码本大小为 k 的 m 次方。

检索



1. 查询阶段：量化查询向量，得到查询向量量化表示，分别计算查询向量子段与数据库各子段的距离。如图中所示，可以得到 4×256 个距离，比如编码为(124, 56, 132, 222)这个样本到查询向量的距离时，将查询向量各子段与数据库向量各子段类别距离相加，得到最终距离。
2. 优化：将全样本的距离计算，转化为到子空间类中心的距离计算。原本暴力计算距离次数随样本数目 N 线性增长，经PQ量化后，距离计算变为 4×256 次。此外编码后，向量使用较短编码表示，内存占用减少。

最近邻搜索

相似搜索由两种方式，区别在于是否对查询向量 X 做量化

1. SDC(symmetric distance computation): 先用PQ量化器对 x 和 y 表示为对应的中心点 $q(x)$ 和 $q(y)$ ，然后用公式1来近似 $d(x,y)$ 。这里 q 表示PQ量化过程。
2. ADC(asymmetric distance computation): 只对 y 表示为对应的中心点 $q(y)$ ，然后用下述公式2来近似 $d(x,y)$

IVFPQ-倒排乘积量化

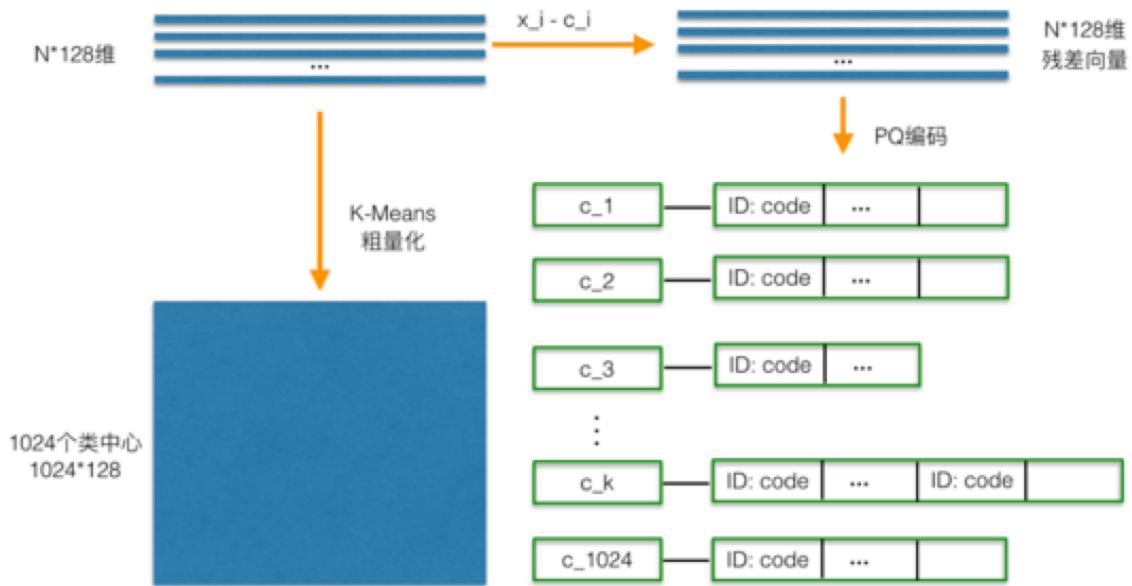
基本思想

通过聚类将查询样本定位到其感兴趣的区域，舍弃不必要的全局计算以及排序。

原理

在PQ乘积量化之前，增加一个粗量化的过程。即采用Kmeans对 N 个训练样本进行聚类，得到聚类中心后，针对每个样本 x_i ，将其与最近的聚类中心 c_i 相减，得到样本 x_i 的残差向量 $(x_i - c_i)$ ，随后进行PQ乘积量化

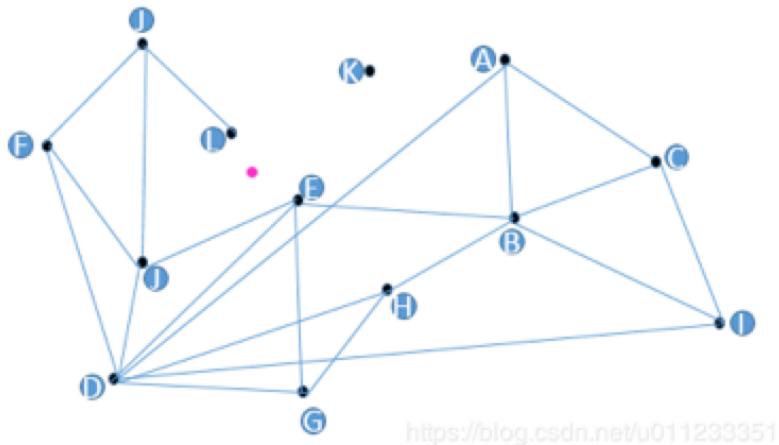
IVF-PQ倒排乘积量化索引



HNSW

朴素想法

查找某个点最邻近的点的朴素想法：将点之间连线，构成查找图。若希望查找粉色点最近邻的点时，可从任意一黑色点出发，计算黑色点的友点与粉色点的距离，并与该黑色点与粉色点的距离对比，若黑色点更近，则终止查找，否则选最近的友点继续。



缺点

1. 图中K点没有友点，无法查询
2. 若距离粉色点最近的两个点，无连线，则效率降低。比如L和E点
3. D点友点过多，怎么确定友点？

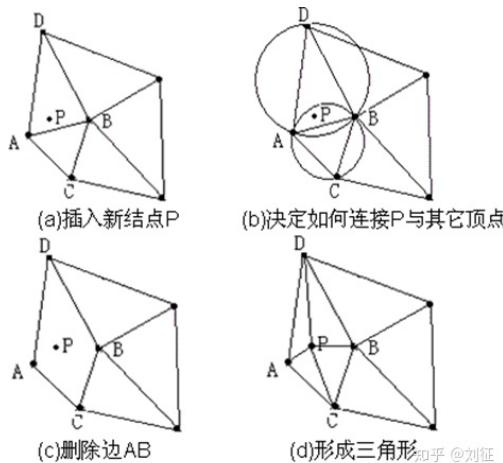
NSW算法

NSW算法采用了得劳内三角形的性质并增加了算法复杂度低以及高速公路机制的构图法，得劳内(Delaunay)三角剖分算法特点是

1. 图中每个点都有友点
2. 相近的点都互为友点
3. 图中所有连接即线的数量最少

得劳内图构建算法：

1. 构造一个超级三角形，包含所有散点，放入三角形链表
2. 将点集中的散点依次插入，在三角形链表中找出其外接圆包含插入点的三角形（称为该点的影响三角形），删除影响三角形的公共边，将插入点同影响三角形的全部顶点连接起来，从而完成一个点在 Delaunay 三角形链表中的插入
3. 根据优化准则对局部新形成的三角形进行优化。将形成的三角形放入 Delaunay 三角形链表
4. 循环执行上述第 2 步，直到所有散点插入完毕

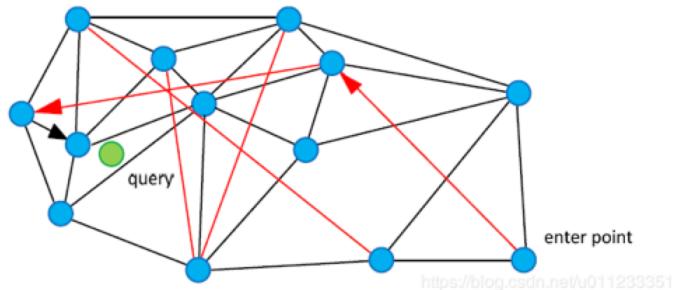


NSW构图算法：得劳内构图和搜索时间复杂度太高，NSW的建图方式非常简单，就是向图中插入新点时，通过随机存在的一个点出发查找到距离新点最近的m个点（m由用户设置），连接新点到这最近的m个点，over。

查找

1. 随机选择一个点作为查询起始点entry_point，把该点加入candidates中，同时加入visitedset
2. 遍历candidates，从candidates中选择距离查询点最近的点c，和results中距离查询点最远的点d进行比较，如果c和查询点q的距离大于d和查询点q的距离，则结束查询，说明当前图中所有距离查询点最近的点都已经都找到了，或者candidates为空
3. 从candidates中删除点c
4. 查询c的所有邻居e，如果e已经在visitedset中存在则跳过，不存在则加入visitedset
5. 把比d和q距离更近的e加入candidates、results中，如果results未满，则把所有的e都加入candidates、results。如果results已满，则弹出和q距离最远的点
6. 循环2-5

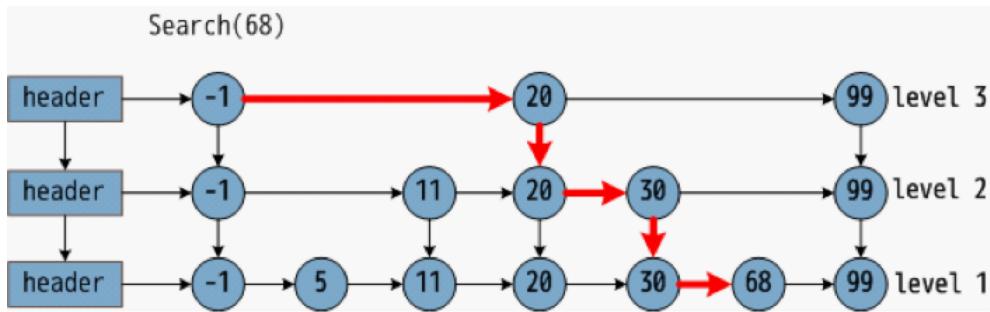
NSW论文中配了这样一张图，黑色是近邻点的连线，红色线就是“高速公路机制”了。我们从enter point点进入查找，查找绿色点临近节点的时候，就可以用过红色连线“高速公路机制”快速查找到结果



<https://blog.csdn.net/u011233351>

跳表

假如有以下跳表结构，分为三层有序链表+分层连接指针。



查找：分层查询，先查第一层，依次查询，构建可采用抛硬币方法选择某节点是否进入上层有序链表，查询速度从原来的有序链表 $O(n^2)$ 变为 $O(\log n)$

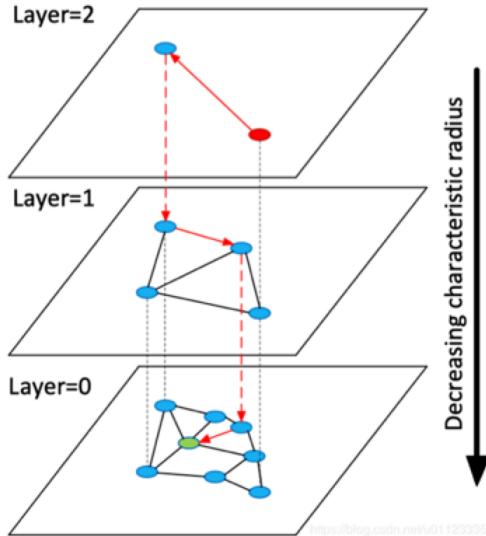
构建：跳表怎么构建呢？三个字，抛硬币。对于sorted_link链表中的每个节点进行抛硬币，如抛正，则该节点进入上一层有序链表，每个sorted_link中的节点有50%的概率进入上一层有序链表。将上一层有序链表中和sorted_link链表中相同的元素做一一对应的指针链接。再从sorted_link上一层链表中再抛硬币，sorted_link上一层链表中的节点有50%的可能进入最表层，相当于sorted_link中的每个节点有25%的概率进入最表层。以此类推。

这样就保证了表层是“高速通道”，底层是精细查找，这个思想被应用到了NSW算法中，变成了其升级版----HNSW。

HNSW

思想：HNSW采用层状结构，将连接线分层，并且其每层中的顶点的平均度数为常数。

查找：从表层即Layer2任意点开始查找，选择进入点最邻近的一些友点，存储在动态列表和废弃表中。计算动态列表中所有点的友点与查找点的距离，废弃表中记录的友点不要计算，再把计算完的友点存入动态链表，去重排序，保留前K个点。若这K个点和之前K个点一样，则返回m个结果，否则继续查找。



构建：

1. 计算待插入点深入到第几层，根据算法 $\text{floor}(-\ln(\text{uniform}(0, 1) \times ml))$ 。 $\text{floor}()$ 的是向下取整， $\text{uniform}(0, 1)$ 的是在均匀分布中随机取出一个值， $\ln()$ 表示取对数， ml 是设置的常数，表示最大层数
2. 插入构图的时候，先计算这个点可以深入到第几层，在每层的NSW图中查找 t 个最紧邻点，分别连接它们，对每层图都进行如此操作，描述完毕
3. 我们需要控制一大堆参数，
 1. 首先，插入时的动态列表c的大小，它的大小直接影响了插入效率，和构图的质量，size越大，图的质量越高，构图和查找效率就越低。
 2. 其次，一个节点至少有几个“友点”，“友点”越多，图的质量越高，查找效率越低。作者在论文中还提到了“max友点连接数”这个参数，设置一个节点至多有多少友点，来提高查找效率，但是设的太小又会影响图的质量，权衡着来。
 3. 上一段中的 ml 也是你来控制的，设置的大了，层数就少，内存消耗少，但严重影响效率，太大了会严重消耗内存和构图时间。
 4. 在论文中，作者将查找状态下的动态列表长度和插入状态下的动态列表长度做了区分，你可以通过调整他们来实现“精构粗找”或者“精找粗构”。