

rec introduction

rec线上框架

在线检索系统

工作流

chain链

分支带并行

DAG

索引设计

正排设计

紧凑型内存

定长与变长设计

查找字段值

倒排设计

获取倒排链过程

一级索引：找输入term id的位置

二级索引：找doc list

增量相关

增量消息三种类型

更新方式

增量倒排加载

增量倒排rebuild

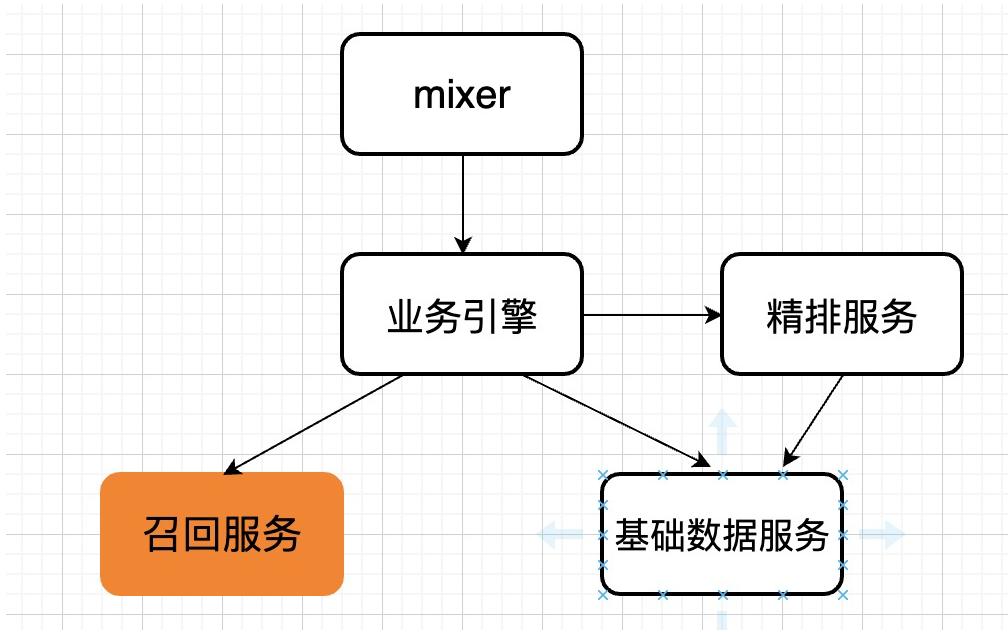
ANN

处理流程

调研

适配

rec线上框架



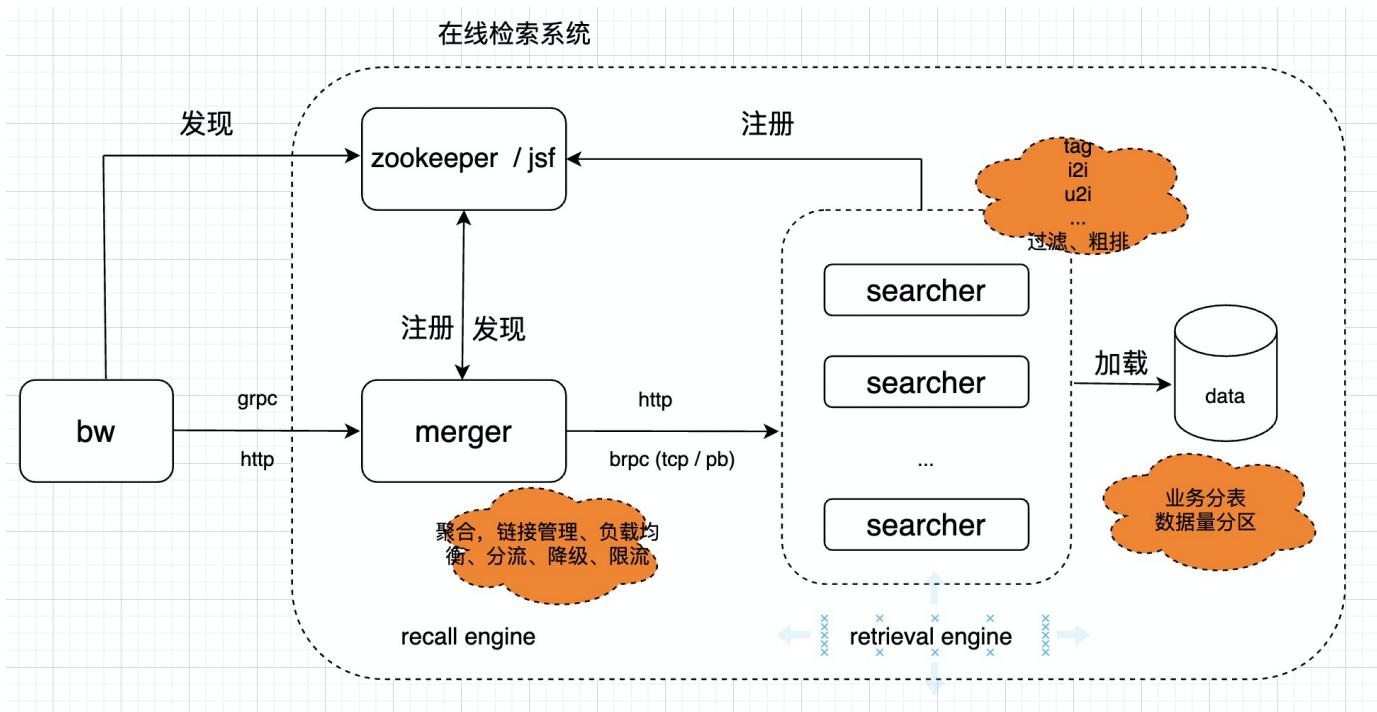
mixer: 推荐+广告结果聚合

基础数据: u侧数据

召回服务: 召回+粗排

精排服务: 精排+重排

在线检索系统



业务引擎: java服务、merger: go服务、searcher: c++服务、zookeeper: 服务发现（迁jsf）

zookeeper做服务发现的缺点

(zookeeper是一个分布式的、开放源码的分布式协调服务，是Google的Chubby一个开源的实现，是Hadoop和Hbase的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。由于Hadoop生态系统中很多项目都依赖于zookeeper，如Pig, Hive等，似乎很像一个动物园管理员，于是取名为Zookeeper。)

- 不是为高可用性设计，cp非ap；选举流程通常耗时30到120秒，期间zookeeper由于没有master，都是不可用的；
- 典型的zookeeper的tps(transactions per second)大概是一万多，即便有client缓存，还是有可能性能、可用性问题；场景：大量机器同时重启时带来的访问；

searcher召回方式：

- tag (标签AND、OR、NOT, 场景 + u2tag) : xapian or `rec_index`
- i2i (相似、相关) : redis or leveldb
- u2i (embedding表示、ANN) : tf + faiss

功能：

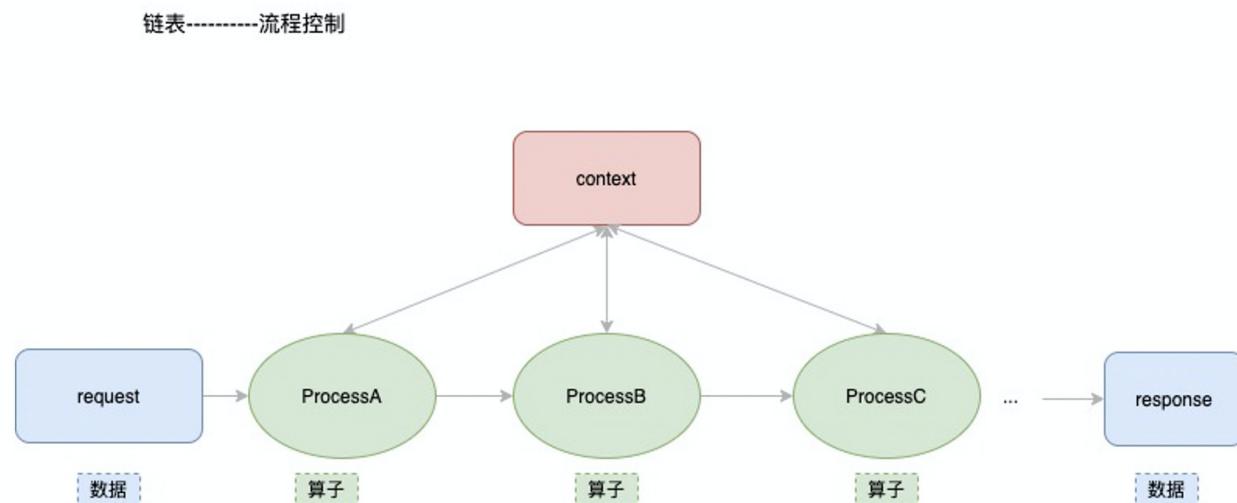
上述方式的召回、过滤、粗排（业务自组织）

工作流

执行器模式：业务用执行流描述，只需关注执行流的创建与节点上的业务处理；模块化、算子复用、函数式编程

chain链

结构简单、表达能力低、数据白板问题

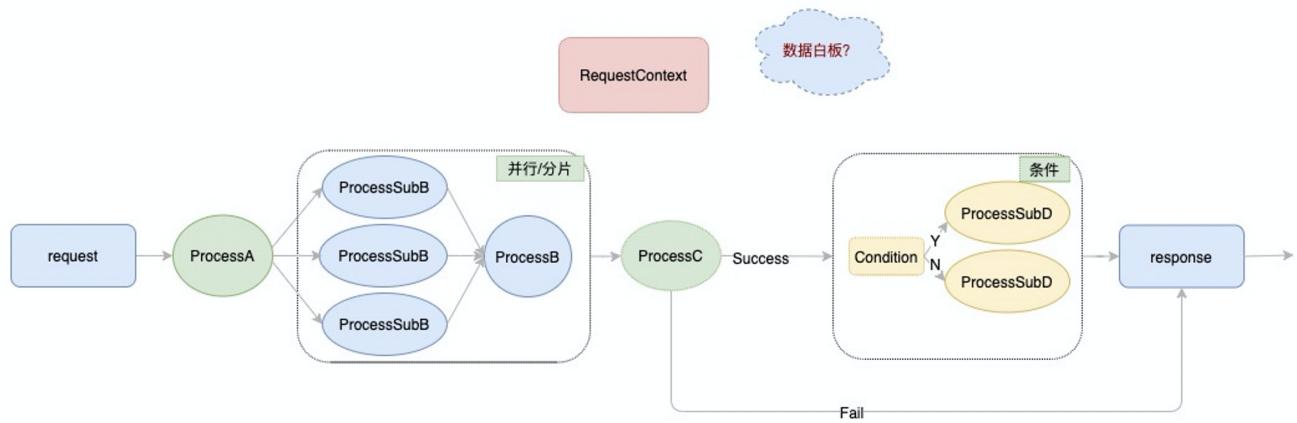


分支带并行

带并行和条件，还是有数据白板、资源编排不够高效

树

流程控制

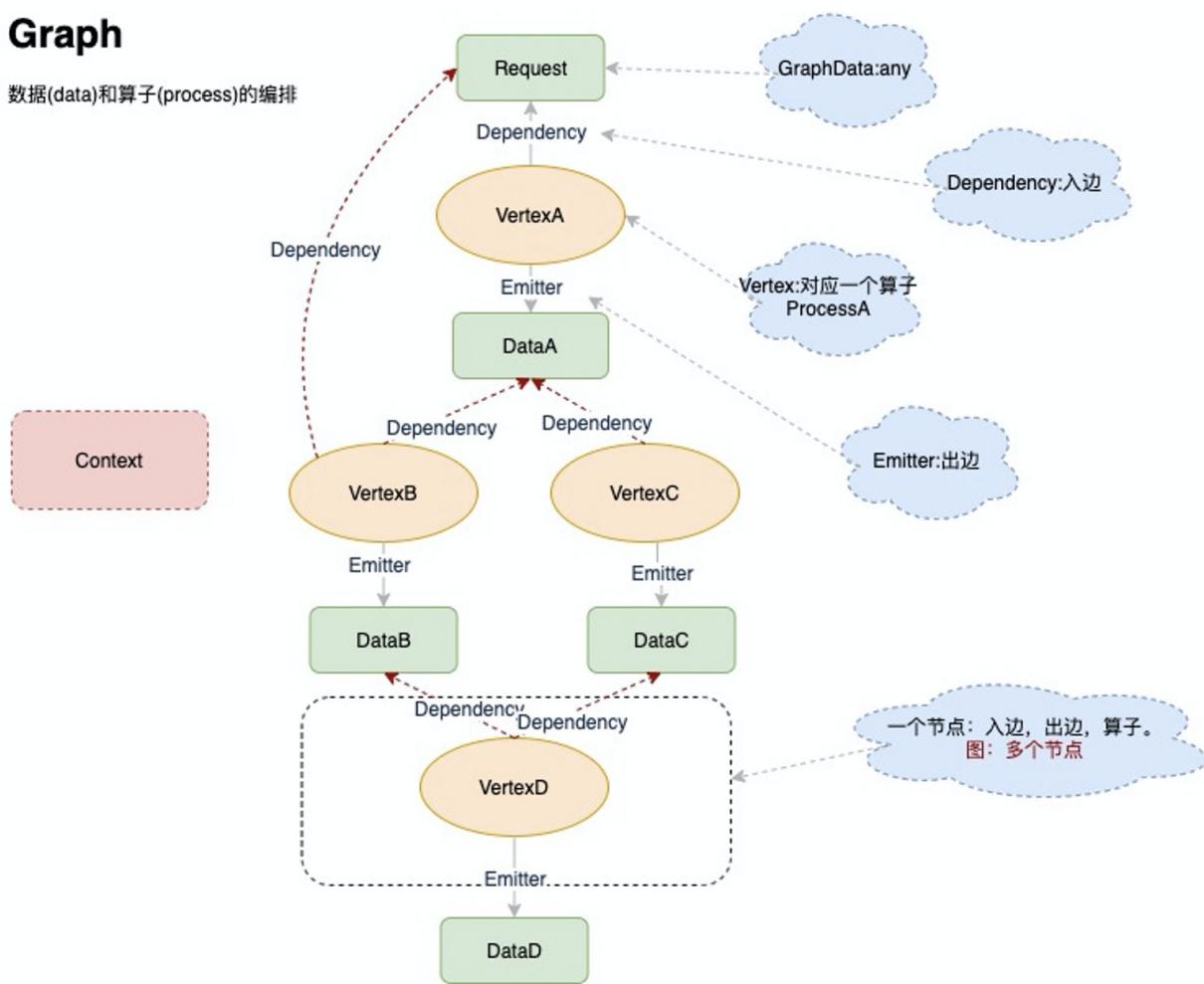


DAG

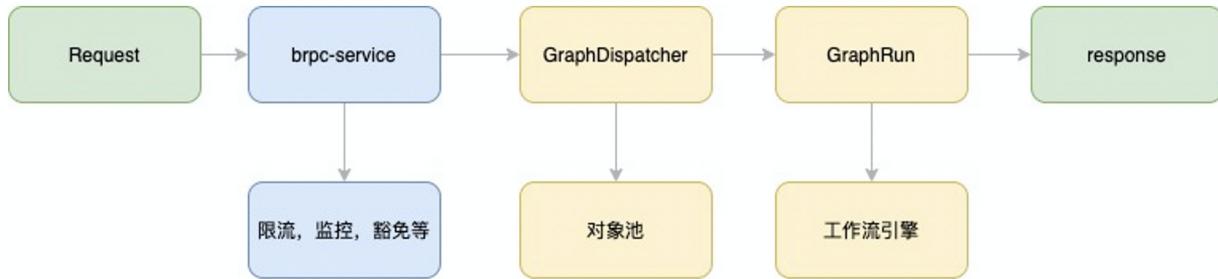
表达能力强，解决数据白板、提高资源编排

Graph

数据(data)和算子(process)的编排



执行情况：



流程：

逆序触发、依赖具备后分配处理单元去执行processor；

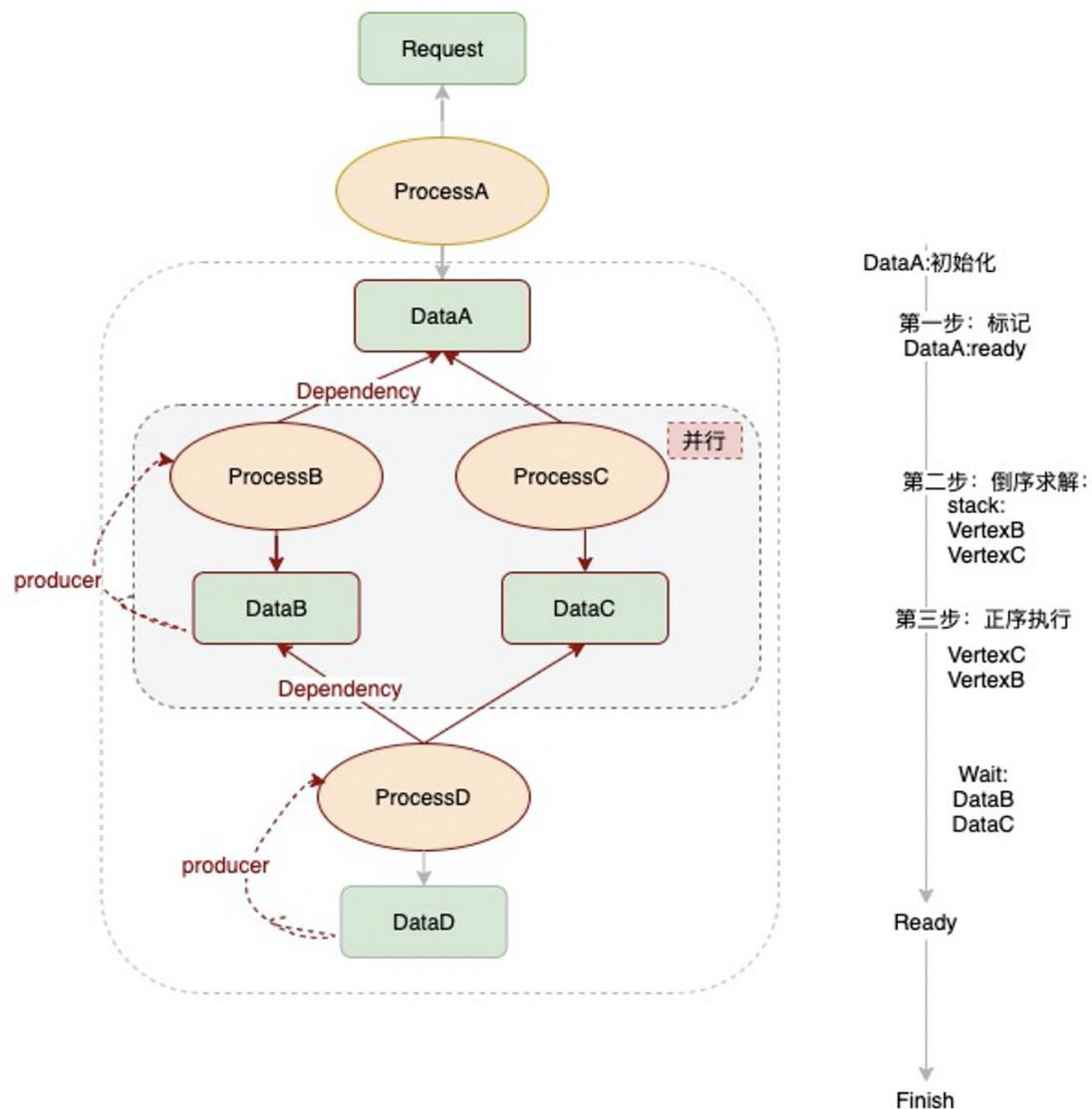
当处理流程本身特别简单时，会有调度的额外开销。

Graph

局部执行：

DataA ready

求解DataD



索引设计

正排设计

紧凑型内存

内存设计成紧凑型，且分为定长、变长两部分，分开存储；用偏移量来进行查找；这样可以随机访问而且节约了内存；

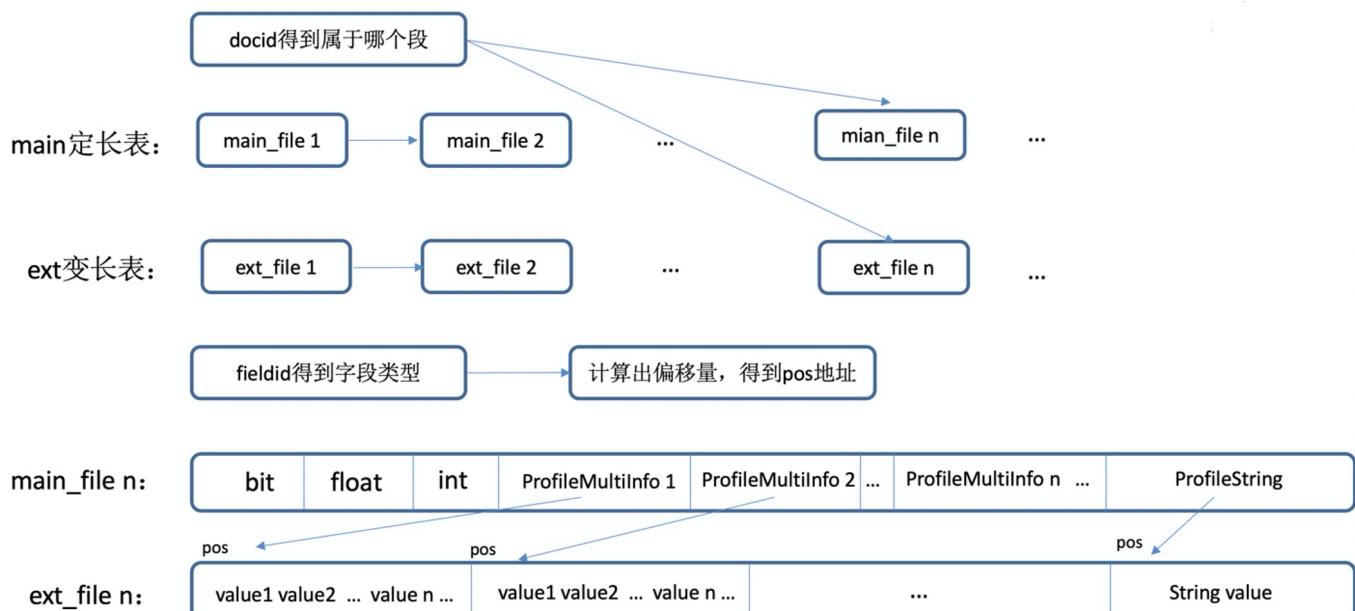
定长与变长设计

定义主表和扩展表存储结构分别对应定长、变长存储；通过配置可以得到每个字段的类型。

所有字段类型：bit类型（定长）、单值数值类型（定长）、多值数值类型（变长）、字符串类型（变长）

查找字段值

- 通过docid先找到具体的段；
- 通过field得到此字段配置；
- 计算偏移量，得到具体地址；



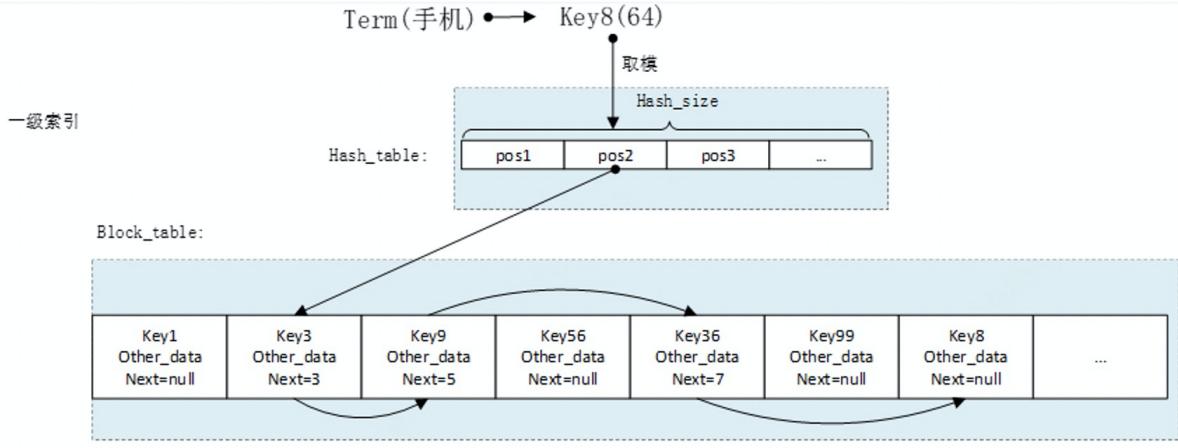
倒排设计

获取倒排链过程

- 找到当前term所在倒排域的倒排；
- 根据term_id查找一级索引，得到一级索引信息；
- 根据一级索引信息中的2级，doc的数量确定倒排链；

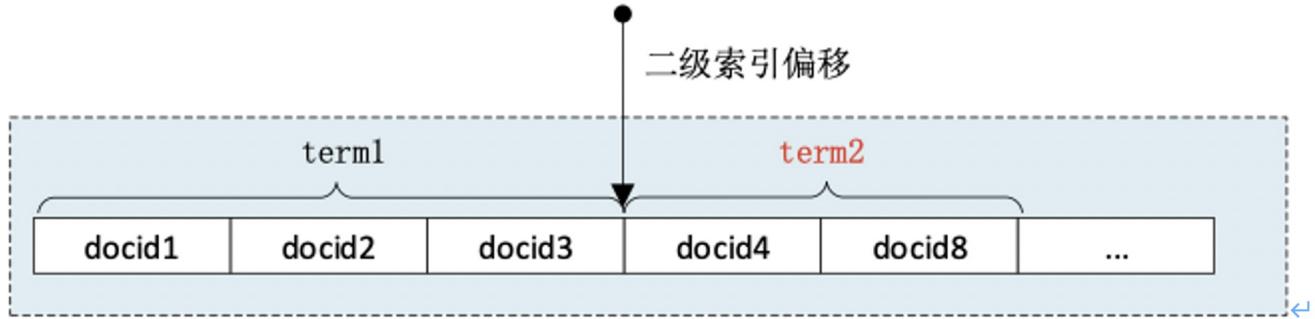
一级索引：找输入term id的位置

一个term视图：



其实是拉链法的hash, 可控桶数量 (term数量相关) ; 增量新增term时减少数据竞争, 因为可以直接插到链表后面

二级索引：找doc list



term id下的doc id有序

增量相关

增量消息三种类型

- add 新增doc
- modify 修改 (倒排字段修改)
- profile 修改正排 (只有正排字段修改)

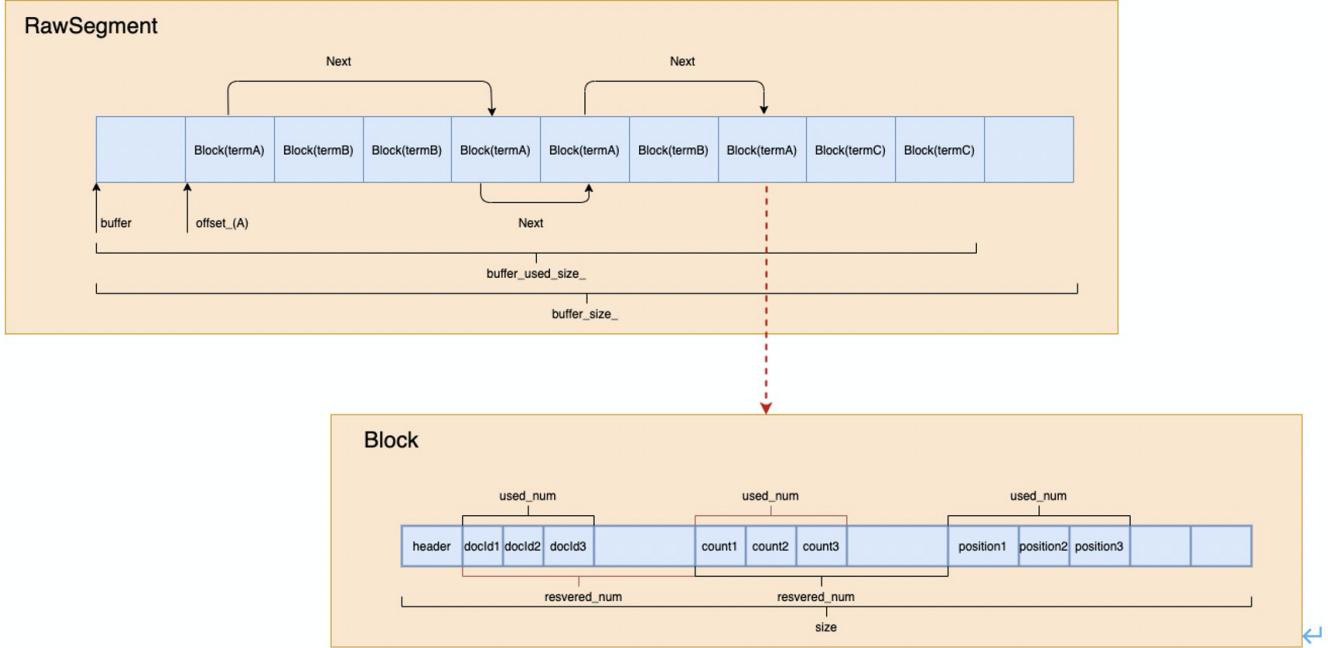
更新方式

- 更新流程从数据源拉取增量数据, copy到本地磁盘上
- 更新流程向server发送加载增量命令
- server加载增量, 与启动加载一致: 扫描磁盘文件并将新数据加载

增量倒排加载

增量与全量分开存储, 以RawSegment组织分配, 以block进行管理, 倒排的doc信息放在block里。

block固定内存方式, 每次插入操作不会影响之前的内容, 因为二级索引中新增的docid总是递增的, 可以保证这一点。



行调度 + write模块，两个动作单线程；解压、解析等其余动作多线程

行调度：保证顺序性

write模块：单线程，一写多读（配合cache line减少数据竞争）

增量倒排rebuild

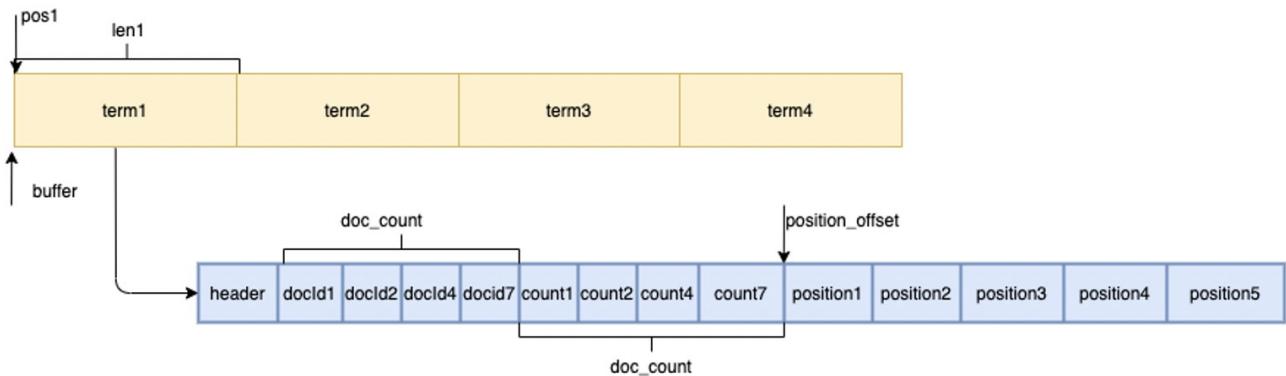
不断增加doc时，减少内存的方法。

进行增量倒排更新的时候，当前 RawSegment 的内存已写满，新申请一个新的RawSegment进行更新，当前RawSegment 加入 rebuild 任务。

- 确定需重建的 RawSegment
- 计算所需空间
- 分配内存
- 重建倒排=>去掉失效的doc
- 逐条更新增量一级索引
- 替换 segment 指针
- 延时释放被重建的 RawSegment （双buffer）

重建后的Segment不再有block的结构

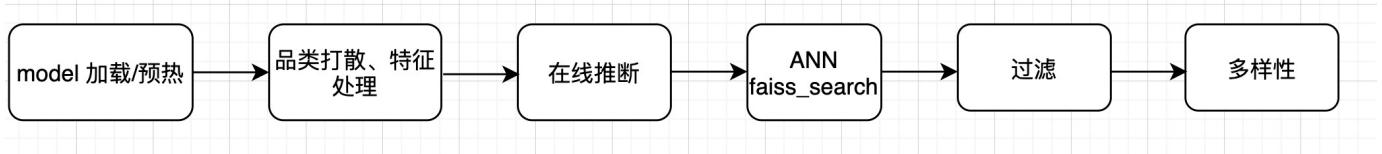
RebuildSegment



ANN

处理流程

在线推断+ANN，常用的在线两步处理；



调研

候选三方库

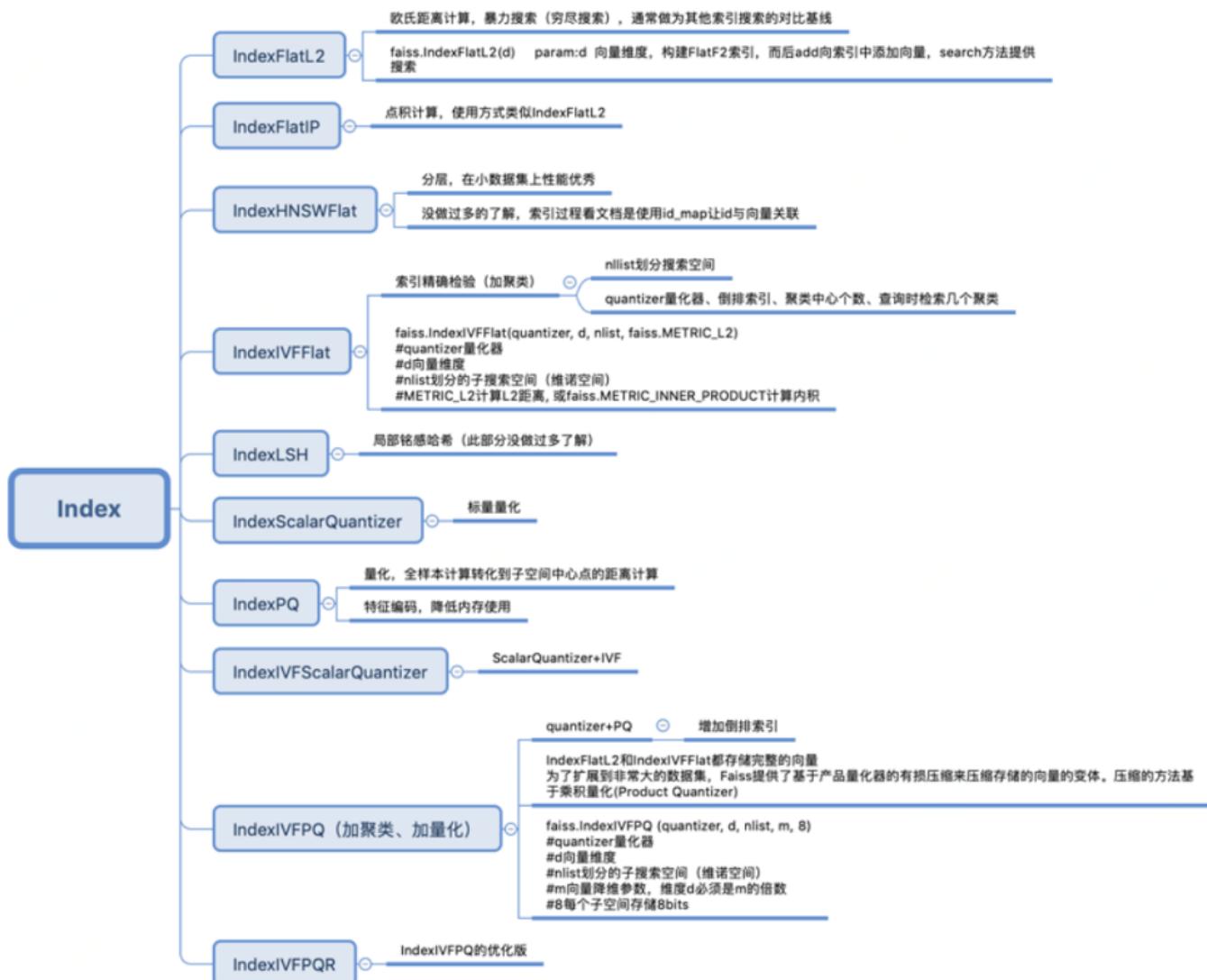
Annoy: 出自Spotify，C++库（提供Python绑定）。支持使用静态索引文件，不同进程可以共享索引。

NMSLIB: (Non-Metric Space Library) C++库，提供Python绑定，并且支持通过Java或其他任何支持Apache Thrift协议的语言查询。提供了SWGraph、HNSW、BallTree、MPLSH实现。

hnswlib(NMSLIB项目的一部分) 相比当前NMSLIB版本，hnswlib内存占用更少。

Faiss: Facebook出品的C++库，提供可选的GPU支持（基于CUDA）和Python绑定。包含支持搜寻任意大小向量的算法。

faiss索引类型

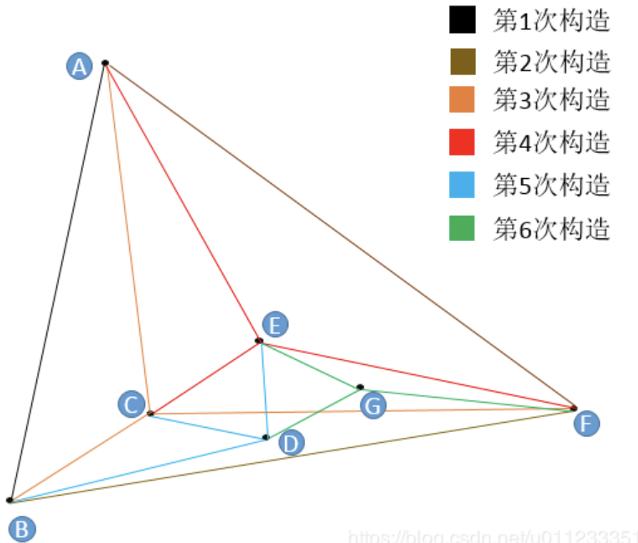


HNSW

Hierarchical Navigable Small World:分层的可导航小世界

要理解 Hierarchical NSW，要先理解 NSW，即没有分层的可导航小世界的结构：

构建：



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

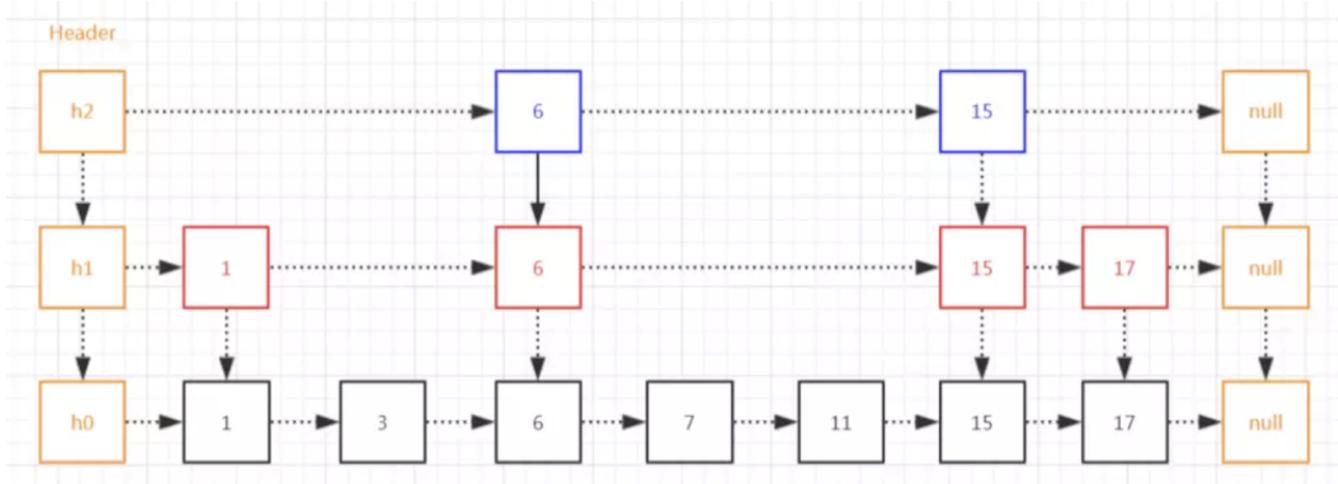
(我们对7个二维点进行构图，用户设置 $m=3$ （每个点在插入时找3个紧邻友点）。首先初始点是A点（随机出来的），A点插入图中只有它自己，所以无法挑选“友点”。然后是B点，B点只有A点可选，所以连接BA，此为第1次构造。然后插入F点，F只有A和B可以选，所以连接FA，FB，此为第2次构造。然后插入了C点，同样地，C点只有A，B，F可选，连接CA，CB，CF，此为第3次构造。重点来了，然后插入了E点，E点在A，B，F，C中只能选择3个点（ $m=3$ ）作为“友点”，根据我们前面讲规则，要选最近的三个，怎么确定最近呢？朴素查找！从A，B，C，F任意一点出发，计算出发点与E的距离和出发点的所有“友点”和E的距离，选出最近的一点作为新的出发点，如果选出的点就是出发点本身，那么看我们的 m 等于几，如果不满足，就继续找第二近的点或者第三近的点，本着不找重复点的原则，直到找到3个近点为止。由此，我们找到了E的三个近点，连接EA，EC，EF，此为第四次构造。第5次构造和第6次与E点的插入一模一样，都是在“现成”的图中查找到3个最近的节点作为“友点”，并做连接。请关注E点和A点的连线，如果我再这个图的基础上再插入6个点，这6个点有3个和E很近，有3个和A很近，那么距离E最近的3个点中没有A，距离A最近的3个点中也没有E，但因为A和E是构图早期添加的点，A和E有了连线，我们管这种连线叫“高速公路”，在查找时可以提高查找效率）

查找其实是一次模拟构建过程，找topK个最近点

- 算法计算从查询 q 到当前顶点的朋友列表的每个顶点的距离，然后选择具有最小距离的顶点。
- 如果查询与所选顶点之间的距离小于查询与当前元素之间的距离，则算法移动到所选顶点，并且它变为新的当前顶点。
- 算法在达到局部最小值时停止：一个顶点，其朋友列表不包含比顶点本身更接近查询的顶点

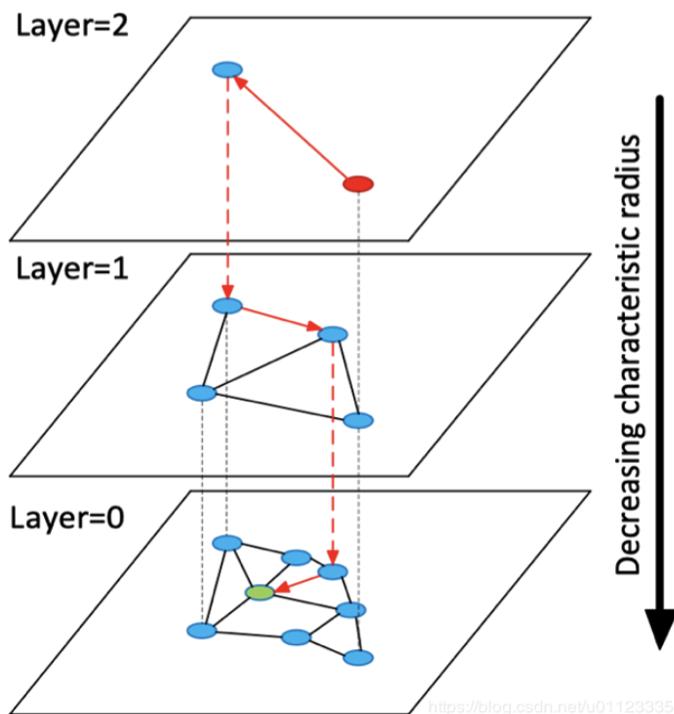
分层思想：表层是“高速通道”，底层是精细查找的思想

skip list:



(我们要查找17这个元素，只需要6、15、17三步就行，是一种空间换时间算法)

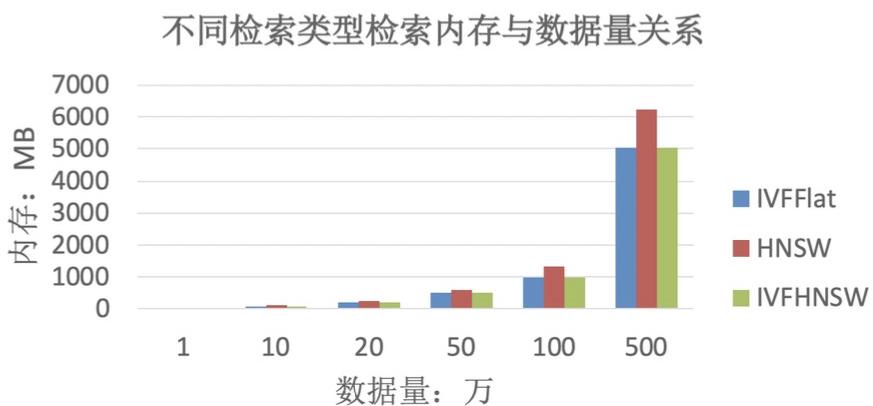
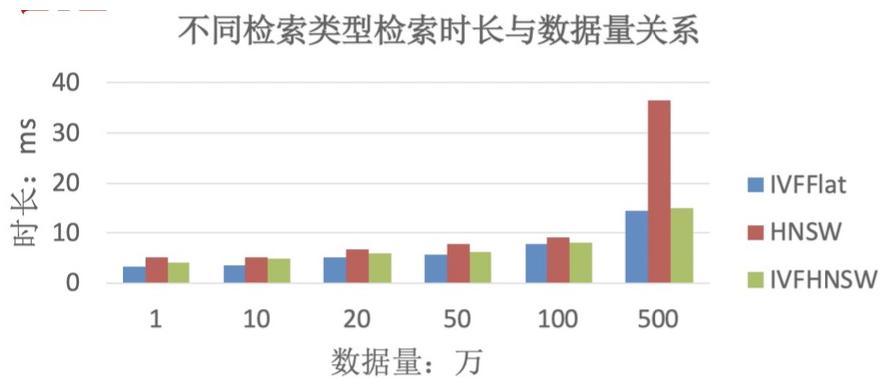
HNSW构建和查询过程



(第0层中，是数据集中的所有点，你需要设置一个常数 m_l ，通过公式 $\text{floor}(-\ln(\text{uniform}(0,1)) \times m_l)$ 来计算这个点可以深入到第几层，关于HNSW算法的描述就基本结束了。我们来大致梳理一下它的查找过程，从表层（上图中编号为Layer=2）任意点开始查找，选择进入点最邻近的一些友点，把它们存储在定长的动态列表中，别忘了把它们也同样在废弃表中存一份，以防后面走冤枉路。一般地，在第 x 次查找时，先计算动态列表中所有点的友点距离待查找点（上图绿色点）的距离，在废弃列表中记录过的友点不要计算，计算完后更新废弃列表，不走冤枉路，再把这些计算完的友点存入动态列表，去重排序，保留前 k 个点，看看这 k 个点和更新前的 k 个点是不是一样的，如果不是一样的，继续查找，如果是一样的，返回前 m 个结果）

适配

256维



选IVFFlat

nprobe经验值 : $4 * \sqrt{N}$ 和 $16 * \sqrt{N}$ 之间

500w 256d: 8092, 在线查询个数50, 要求分布均匀;

离线处理及校验



品类打散的多向量

性能: openMP并行、grpc异步调用