

WeOLAP: 微信基于 ClickHouse的向量检索探索

冯吕 微信 技术架构部

目录

- 向量检索简介
- 业务背景
- *ClickHouse*原生向量检索&ANN检索增强
- 基于SSD的向量检索
- 总结与展望

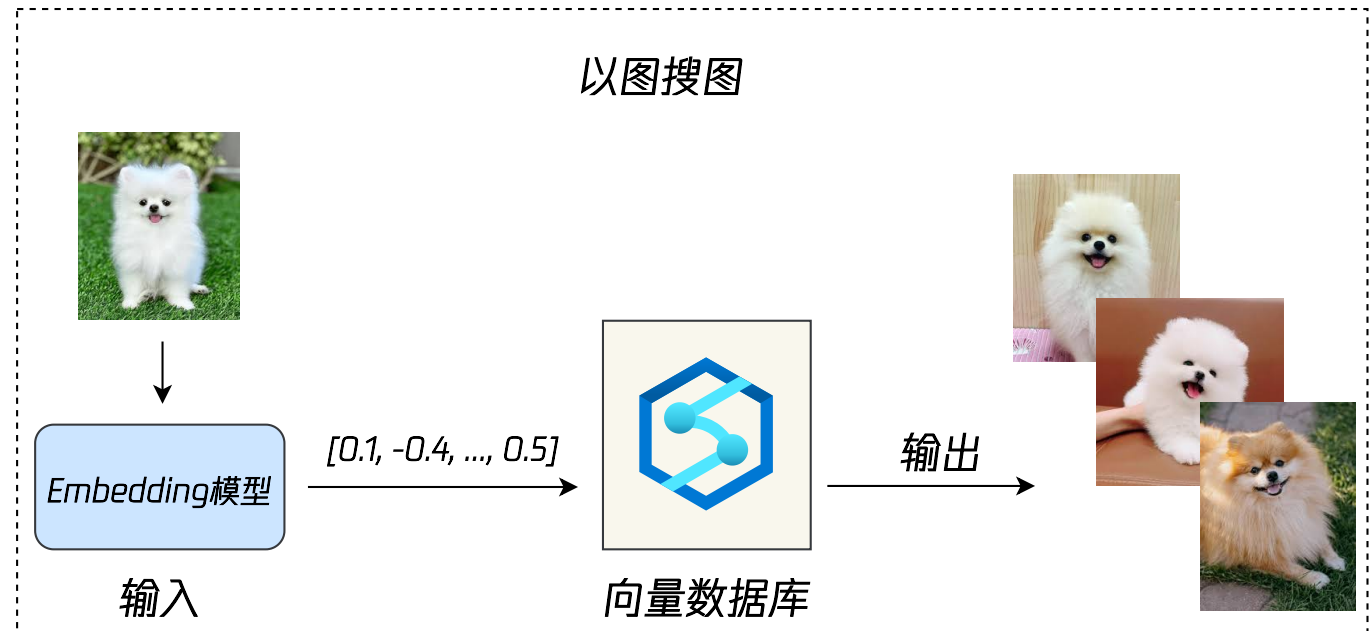
向量检索简介

什么是 *embedding*

- *Embedding* 是真实世界中“离散”物体，如单词，文本，图片视频等，映射到“连续”向量空间的一种表示
- AI 理解真实世界的基础，“万物皆可*embedding*”
- 向量检索并不“新鲜”，过去已广泛应用于计算机领域的各行各业中

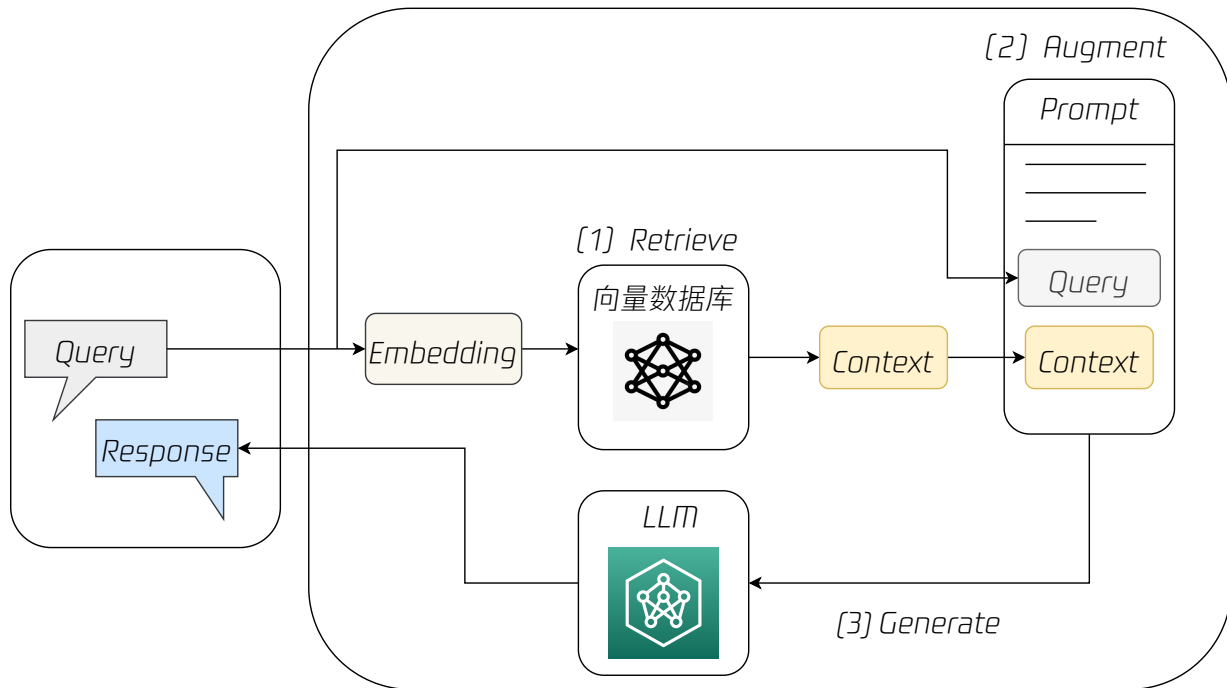
- 传统向量检索用途：

- 推荐系统
- 相似性检索
- 自然语言处理：语义相似性搜索和相关性分析



向量检索重回视野

- 随着大模型的兴起，向量检索再一次重回了大众的视野
- RAG框架：针对用户query进行embedding提取，检索向量数据库获取相关上下文，通过prompt模版构造基于用户query和检索上下文的LLM指令输入，基于LLM生成对应的回复
- 从“无记忆交互” [如ChatGPT等大模型] 到“有记忆交互” [如LangChain]

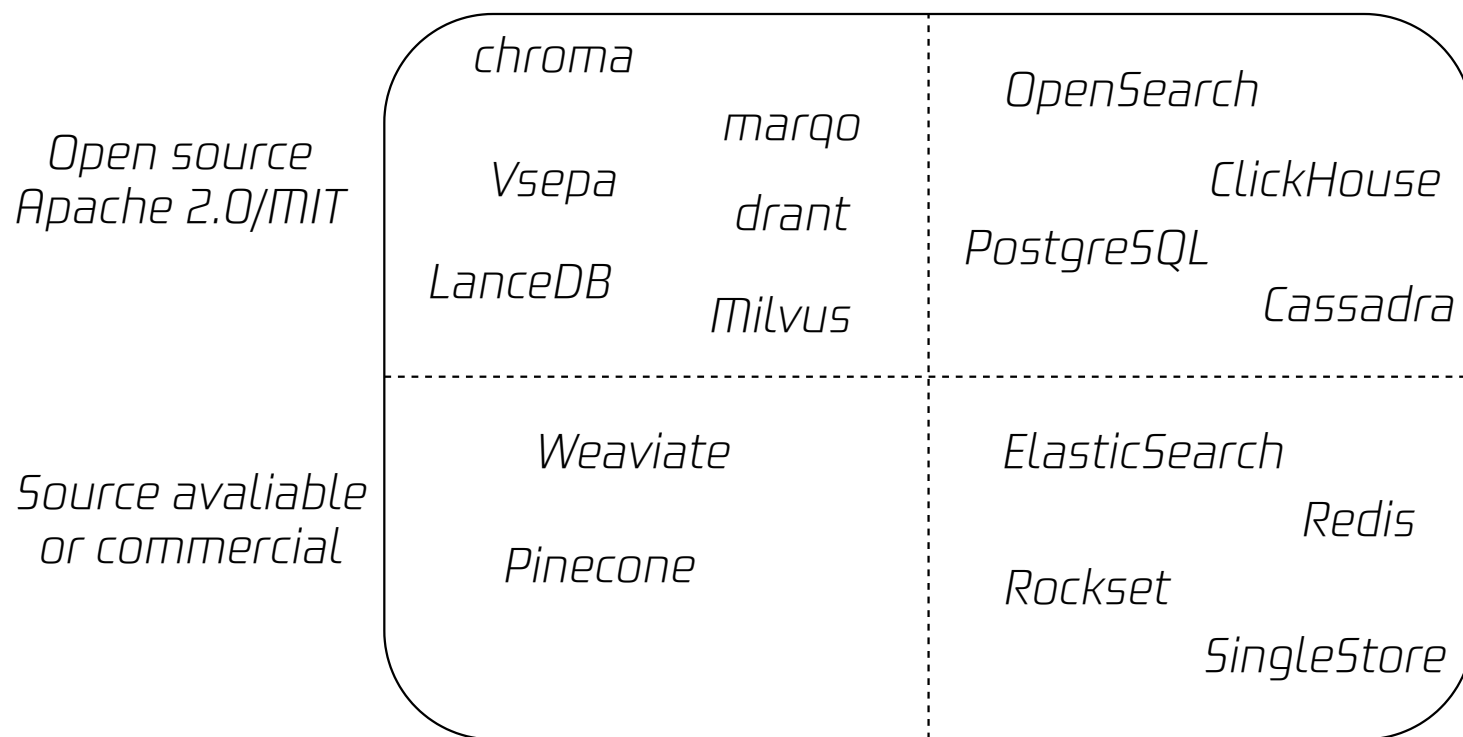


在RAG中，向量数据库发挥了重要基础组件作用，而Embedding模型的搭建、相关数据存储和管理等RAG操作的门槛较高：希望实现向量数据的无感知计算和检索

如何选择Vector Database

- 目前，市面上的向量数据库主要分为两类： [1] 基于向量检索引擎实现关系数据库； [2] 基于原生数据库引入向量检索引擎

从向量检索引擎到数据库 数据库添加向量检索引擎



向量检索算法分类:

- Hash-based
- 基于量化
- 基于聚类
- Tree-based
- Graph-based (如HNSW)
- ...

业务背景

业务场景

- 按照时效性、性能要求等，可以将emb查询分为离线、近线和在线三类场景：

离线：

- 批处理，小时/天调度
- 计算量大，对时效要求低
- 场景：投放种子包扩散，文章近似度分析等

系统？

近线：

- 介于离线和在线之间
- 平衡时效与准确性
- 场景：以图搜图，商品检索，近线召回，大模型RAG等

系统？

在线：

- 高QPS，百万QPS低延迟
- 时效性要求高
- 场景：在线推荐服务召回

系统？

- 对于上述不同的业务场景，通常需要不同的系统和服务来满足需求

批处理emb查询

- 号码种子包扩散：在投放场景中，客户提供十万量级种子包，画像系统使用 *embedding* 表征用户的特征和行为；为实现精准的人群投放，需要在数亿 *embedding* 中选择相似近邻来扩充人群包到百万甚至千万量级

视频号·加热平台

加热视频

订单管理

数据分析

加热直播

订单管理

数据分析

账户管理

下单金额 1000 2000 3000 4000 5000 自定义

加热时长 0.5小时

观众性别 不限 男 女

根据粉丝层推荐 不限 选择相似作者的粉丝
将定向推给你选择作者的粉丝，以及与粉丝相似的人。

1.emb 扩散场景

根据名单推荐 不限 选择指定名单
支持手机号SHA256、手机号MD5、微信OpenID名单

观众年龄 24-30岁

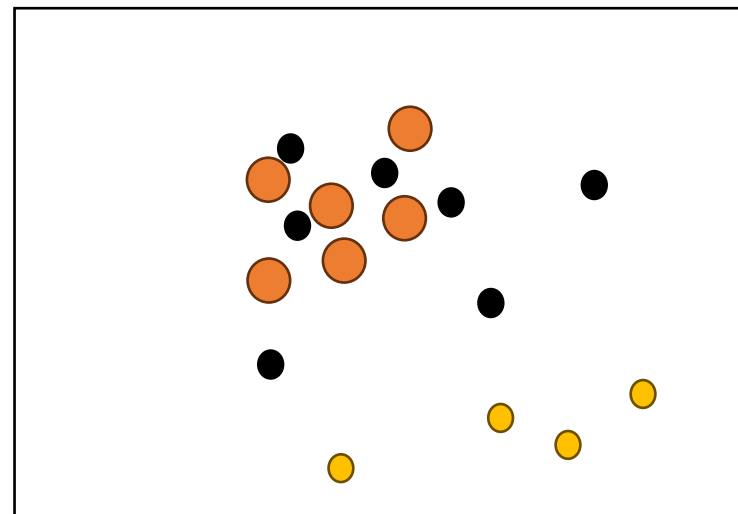
观众设备 不限 iOS 安卓

观众城市 已添加北京市

2.商业画像

观众兴趣 已添加养蜂器具

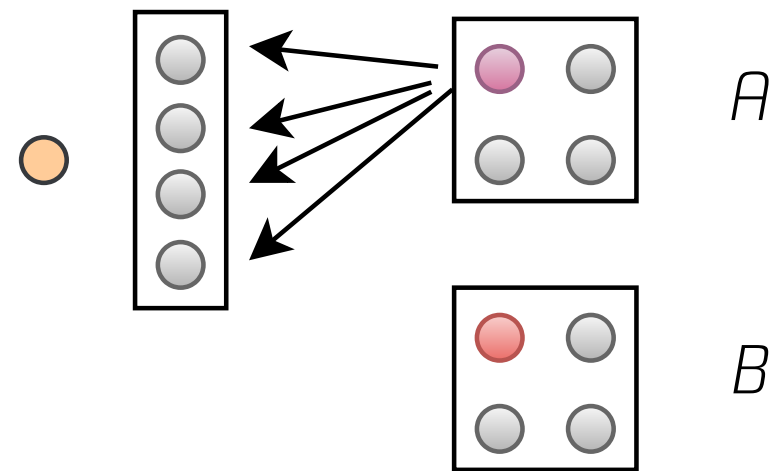
支付方式



批处理emb查询

- 文章emb近似度分析: User-Items-Items查询举例, 给用户A推荐与A“阅读习惯类似群体”阅读过的文章, 研究员一般会采不同embedding模型 + 多类距离计算方式 + 不同的算法组合策略等方式, 尝试调试出主观上表现好召回策略, 后续再上AB实验分析。此过程需要大量的手工调试分析, 需要一种高效敏捷调试交互看板, 进行策略探索
- 算法描述: 先求某个id对应的集合 `Items_emb_set`, 再遍历每个item求解它离 `Items_emb_set` 集合每个元素的平均值, 并按照每个分类取距离最近一条

```
with
  [select groupUniqArray(itemid) from table where xxx IN ('1234') and
  day_ = '2023-06-18'] as itemid_filter_arr ,
  [select groupArray(embedding) as arr1 FROM table WHERE emb_type
= 'xxx' and has(itemid_filter_arr,itemid)] as emb_set ,
  embedding as emb_l ,
  arrayReduce('avg',arrayMap(x -> cosineDistance(x,emb_l),emb_set))
as cosineDistance
select itemid, title, xxx, 1-cosineDistance as similarity
FROM table
WHERE emb_type = 'xxx' order by cosineDistance asc limit 1 by xxx
```



业务场景

- 按照时效性、性能要求等，可以将emb查询分为离线、近线和在线三类场景：
 - 批处理场景：ClickHouse已经足够完美支撑
 - RAG等近线场景：ClickHouse存在不足，过去点查能力差，QPS低
 - 在线场景：需要靠专用Sim服务支撑

离线：

- 批处理，小时/天调度
- 计算量大，对时效要求低
- 场景：投放种子包扩散，文章近似度分析等

ClickHouse：优秀批处理能力

近线：

- 介于离线和在线之间
- 平衡时效与准确性
- 场景：以图搜图，商品检索，近线召回，大模型RAG等

系统？

在线：

- 高QPS，百万QPS低延迟
- 时效性要求高
- 场景：在线推荐服务召回

专用Sim服务，如微信内部SimSvr

- 我们希望进一步扩展系统能力边界，从离线往近/在线场景靠近

*ClickHouse*原生向量
检索&ANN检索增强

Why ClickHouse

- 向量化引擎，原生批处理能力强，支持数据的高效过滤、聚合、统计分析等
 - 但点查能力一般，近线向量检索精确计算计算量大，与数据量成正比
- 新版本引入了基于HNSW算法的ANN索引，理论上能够较好支撑近线场景的检索需求：

Initial implementation of vector similarity index #63675

Merged rschu1ze merged 31 commits into ClickHouse:master from rschu1ze:vector-search on Aug 13

Conversation 31 Commits 31 Checks 121 Files changed 50



rschu1ze commented on May 13 · edited

Member

- 此外，业内还有基于ClickHouse的向量数据库：MyScaleDB
 - 支持多种不同向量检索算法以及BM 25全文检索



MYSCALE

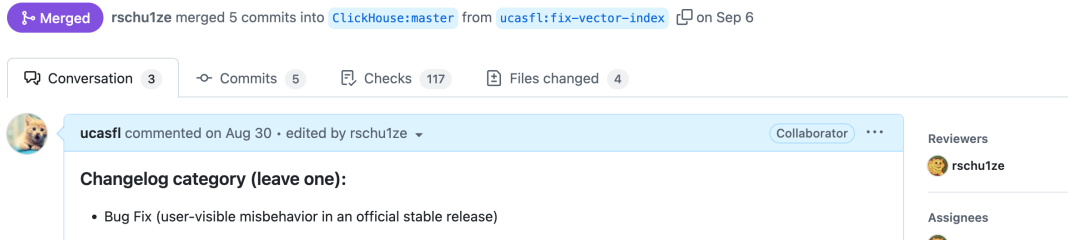
The SQL Vector Database for Scalable AI

Why ClickHouse

- 评估发现, ClickHouse 现有的ANN索引有诸多不符合预期以及性能问题

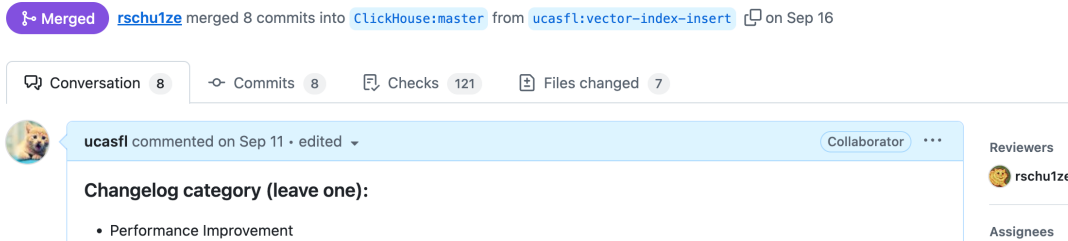
- 1. 不支持cosineDistance:

Fix bug that vector similarity index does not work for cosine distance #69090



- 2. 索引构建慢:

Speedup insert performance of vector similarity index by parallelization #69493



- 3. 没有缓存: 索引加载/反序列化耗时巨大

ClickHouse VS MyScaleDB

```
WITH [...] as query_vector
SELECT
    day_,
    product_id,
    cosineDistance(me5_embedding, query_vector) AS distance
FROM dwd_ec_me5_embedding
ORDER BY distance ASC
LIMIT 5
SETTINGS allow_experimental_analyzer = 0
```

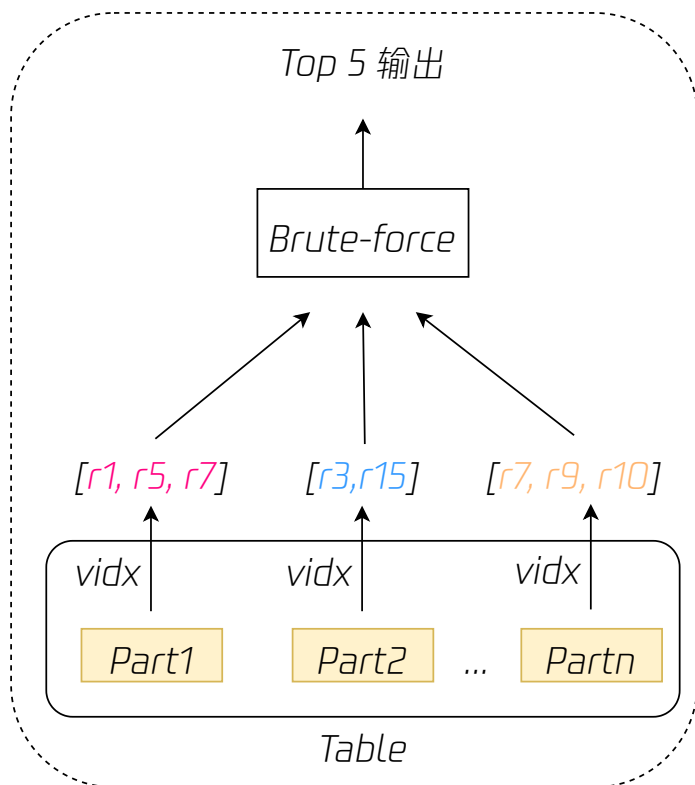
```
-- ch
5 rows in set. Elapsed: 0.938 sec. Processed 243.81 thousand rows, 757.02 MB
Peak memory usage: 362.32 MiB.

-- myscaledb
5 rows in set. Elapsed: 0.019 sec. Processed 18.54 thousand rows, 2.39 MB (962
```

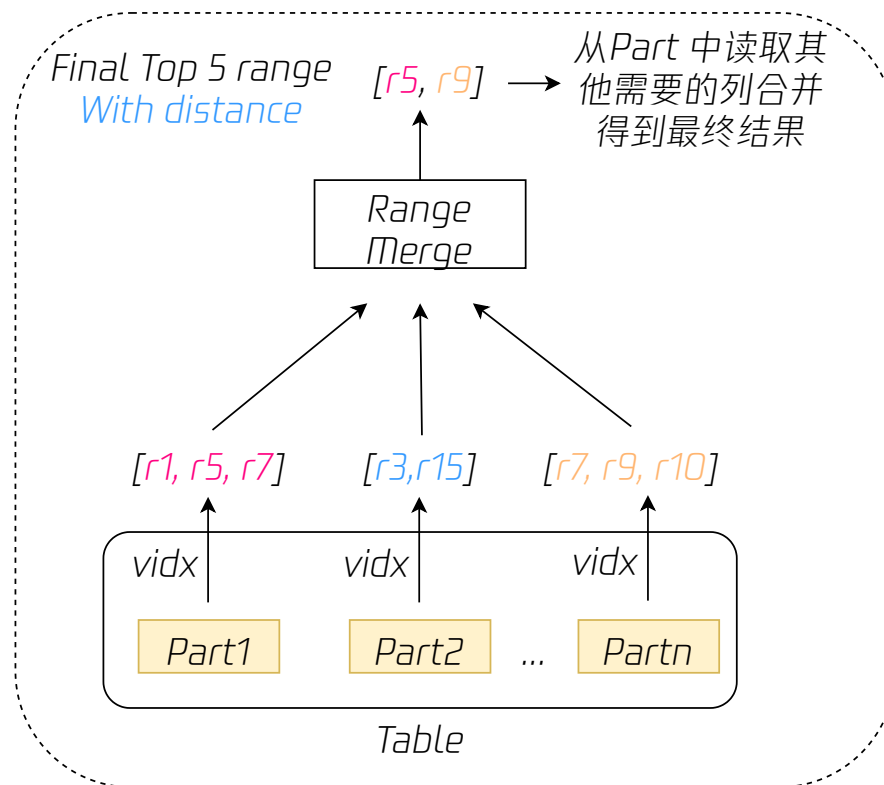
相同索引查询, ClickHouse 的耗时和IO远大于 MyScaleDB([ClickHouse#Issue69320](#))

ClickHouse VS MyScaleDB

- 索引工作原理对比：找到与检索向量距离最近的Top 5向量



ClickHouse



MyScaleDB

ClickHouse:

- ANN索引实现框架类似普通跳数索引
- IO, 计算更多: 没有合并range和复用索引返回的distance

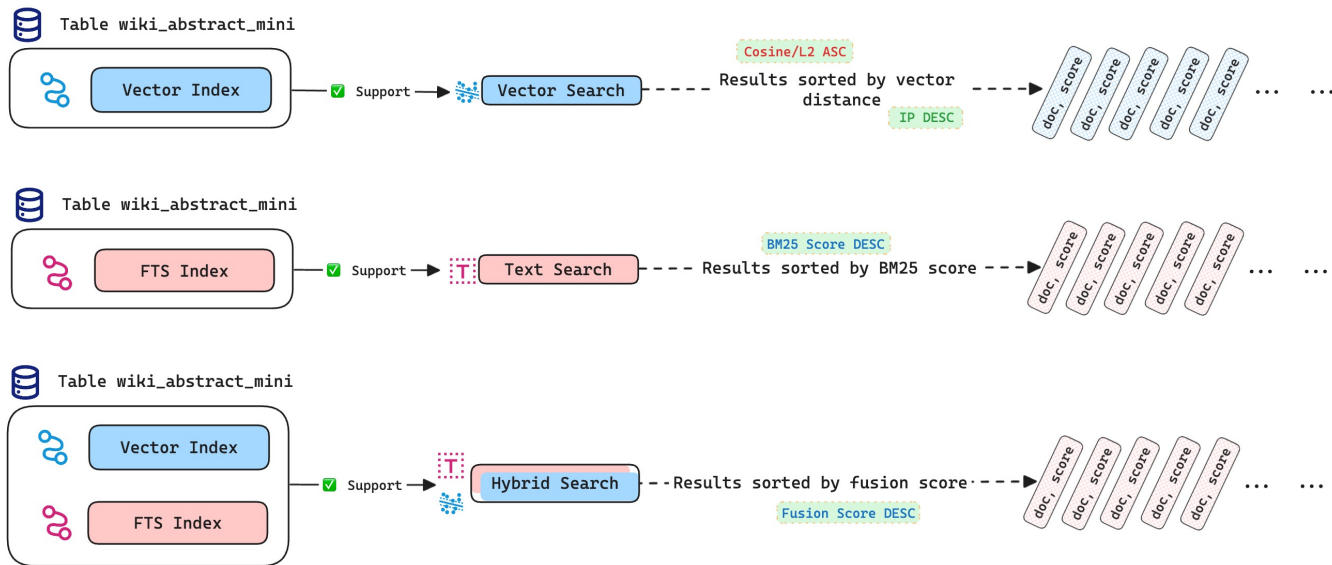
MyScaleDB:

- 索引框架实现方式较trick
- 有range合并和距离复用, 性能更佳

因此, 我们引入了基于ClickHouse的向量数据库MyScaleDB, 用于微信实际业务中

全文检索&混合搜索

- 向量搜索在跨文档语义匹配和深度语义理解方面表现出色
- 全文检索适用于基于关键词的检索和文本匹配，MySQL支持BM25算法
- 混合搜索能够结合向量检索和全文检索的优点，将二者的搜索结果融合 [RSF或RRF]，以提高不同场景下的适应性

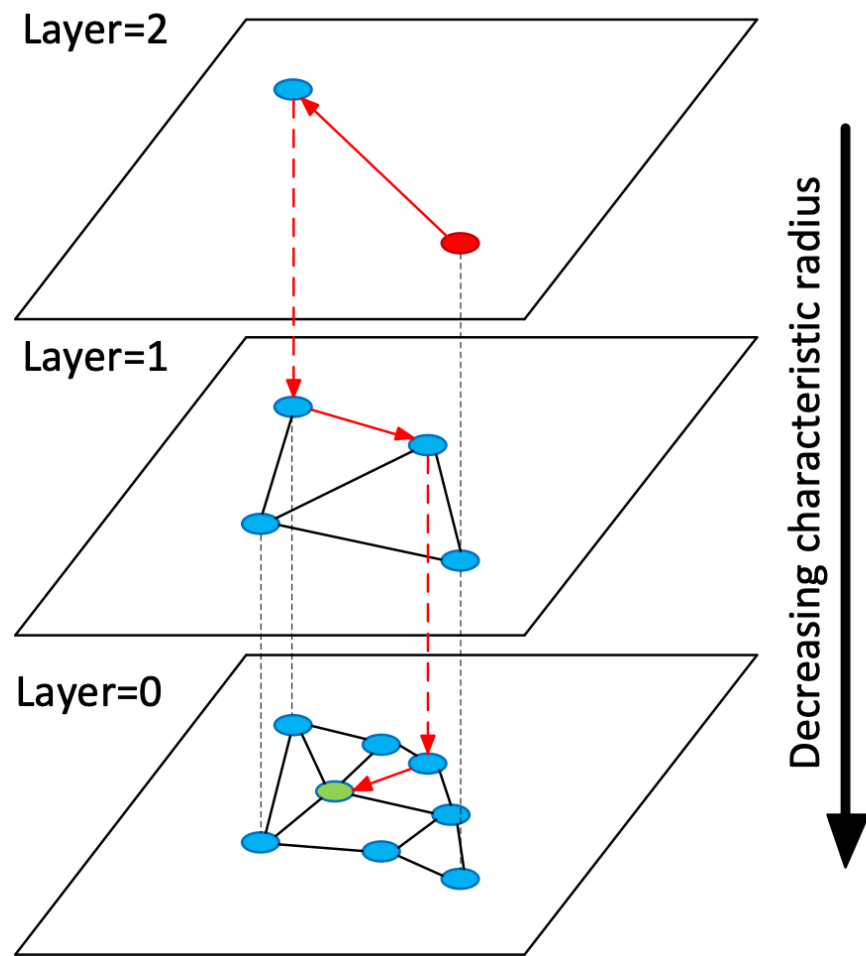


```
SELECT
  id,
  title,
  body,
  HybridSearch('fusion_type=RSF', 'fusion_weight=0.6')
    (body_vector, body, TextEmbedding('Charted by the BGLE'),
     ' BGLE') AS score
FROM default.wiki_abstract_mini
ORDER BY score DESC
LIMIT 5;
```


基于SSD的向量检索

内存向量索引算法弊端

- 内存资源占用大，以HNSW为例：
 - 索引结构主要由近邻图和原始向量两部分组成，检索时需要全放在内存
 - 1亿 X 768 维度的32位浮点数向量，构建索引后内存占用 ~ 270 GB
- 难以支撑十亿到百亿级别的向量检索需求
- 例如，微信某广告视频判重业务：
 - 数据量 ~ 200亿 X 512维
 - 按HNSW的内存占用评估约需要 90 TB 内存
 - 内存资源消耗巨大



DiskANN-基于SSD的磁盘索引

- *DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node* [微软 NeurIPS 2019]
 - <https://github.com/microsoft/DiskANN>
- 单台机器能够支撑10亿级别近邻搜索
- 基于图的索引，索引数据能够放在磁盘上提供服务，同时保证较好的性能
 - 字节ES、Milvus[一个专用向量数据库]中都引入了DiskANN算法
 - 字节ES描述：“引入了内存和磁盘更好平衡的 DiskANN 算法后，云搜索团队在 200 亿单一向量库中已成功验证了其效果：DiskANN 论文提到可以节约 95% 的资源，从多个实际用户案例来看，这一收益值非常接近。客户仅需几十台机器即可稳定高效地满足百亿级业务需求。”

DiskANN-索引构建

- 类似NSG, 引入乘法因子 $\alpha > 1$ 解决贪婪搜索路径过长问题, 生成的图相对HNSW, NSG具有更小的半径, 能够最小化磁盘IO的次数

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

Data: Graph G with start node s , query x_q , result size k , search list size $L \geq k$

Result: Result set \mathcal{L} containing k -approx NNs, and a set \mathcal{V} containing all the visited nodes

begin

```
initialize sets  $\mathcal{L} \leftarrow \{s\}$  and  $\mathcal{V} \leftarrow \emptyset$ 
while  $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$  do
    let  $p^* \leftarrow \arg \min_{p \in \mathcal{L} \setminus \mathcal{V}} \|x_p - x_q\|$ 
    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  $x_q$ 
return [closest  $k$  points from  $\mathcal{L}; \mathcal{V}$ ]
```

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  $\mathcal{V}$ 
```

DiskANN-索引布局

- 通过PQ量化对vector进行压缩，压缩后的vector放在内存中，graph和全精度向量放在SSD上
 - 对于SSD上的每一个向量，我们将其R个邻居ID跟随向量本身存储在一起，如果邻居个数不足R，通过补0对齐：

R=5 [最多邻居个数]

| | | | | | |
|---------|--------------------|---------|------------------|---------|------------------|
| Vector1 | [3, 6, 11, 19, 55] | Vector2 | [2, 11, 8, 0, 0] | Vector3 | [1, 4, 5, 14, 0] |
|---------|--------------------|---------|------------------|---------|------------------|

- 内存占用预估：

$$\frac{(n \times index_pq_bytes)}{2^{30}} + \frac{250000 \times (4 \times max_degree + sizeof(T) \times ndim)}{2^{30}}$$

- 其中，n为数据量，T为数据类型，ndim为数据维度，index_pq_bytes为量化字节数，max_degree为近邻图的最大邻居数
- 例如，假设原属向量维度为512，数据量为1亿，index_pq_bytes为64，max_degree为64，数据类型为32位浮点数，则需要占用的内存约为7.87GB

DiskANN-索引搜索

- *Beam Search*: SSD获取少量随机扇区需要的时间与获取一个扇区需要的时间几乎一样, 为了减少SSD的访问次数, 在进行*GreedySearch*时一次性访问少量 W 个未访问的邻居 ($W=1$ 退化为*GreedySearch*)
- 使用内存中的压缩向量计算距离来指导搜索方向, 同时通过磁盘上的全精度向量计算距离进行重排序保证精度
 - 根据上文提到的磁盘布局, 全精度向量能够和其邻居ID存储在SSD的同一个扇区, 因此在加载节点邻居ID的时候, 同时缓存了全精度向量, 避免了额外的磁盘访问
- 同时, 在内存中缓存频繁访问的节点, 减少磁盘访问次数

引入DiskANN向量索引

- 因此，我们在MySQL中引入了DiskANN索引算法

- 创建表:

```
CREATE TABLE test_vector
(
  `day_` Date,
  `product_id` String,
  `product_name` String,
  `me5_embedding` Array(Float32),
  CONSTRAINT check_length CHECK length(me5_embedding) = 768
)
ENGINE=MergeTree
PARTITION BY day_
ORDER BY tuple()
```

- 创建索引:

```
ALTER TABLE test_vector
  ADD VECTOR INDEX idx_me5_embedding TYPE DiskANN('metric_type = Cosine');
```

- 查询:

```
WITH (
  SELECT me5_embedding
  FROM test_vector
  LIMIT 1
) AS query_vector
SELECT distance(me5_embedding, query_vector) AS distance
FROM test_vector
ORDER BY distance ASC
LIMIT 5
```

一些索引参数:

- *metric*: 距离类型
- *max_degree*: 最大邻居数
- *index_pq_bytes*: 量化字节数, 影响索引内存占用
- *search_list_size*: 搜索优先队列大小
- ...

性能评估

- 与HNSW对比，使用真实业务数据进行测试 [768维的32位浮点数]
 - 内存占用:

| 数据量 | DiskANN | HNSW |
|------|---------|--------|
| 400万 | 几百MB以内 | 14.8GB |
| 1亿 | 7.5GB | 270GB |
| 10亿 | 68GB | |

注：测试时，DiskANN索引的index_pq_bytes参数设置为64

- 结论:
 - ✓ 相同数据量下，DiskANN索引的内存占用远远小于HNSW，资源节省95%以上
 - ✓ 10亿量级，DiskANN内存占用仅68GB，单台机器即可满足需求

性能评估

- 与HNSW对比，使用真实业务数据进行测试 [768维的32位浮点数]：
 - 点查QPS:

| 数据量 | DiskANN | HNSW |
|------|---------|------|
| 400万 | 377 | 506 |
| 1亿 | 371 | 518 |
| 10亿 | 60 | |

注：400万量级为在开发机上测试，与1亿、10亿测试机器不对等

- 结论：
 - ✓ 在400万，1亿相同数据量级下，DiskANN索引的点查QPS能够达到基于内存的HNSW索引的2/3
 - ✓ 在10亿量级下，DiskANN的QPS较低，只有60：数据量大，IO计算更多，并且存在请求放大问题 [请求放大与part数成正比]

如何扩展QPS

- 向量检索存在请求放大问题 [shard, part] :
 - 向量索引的检索是一个开销较大的操作
 - 当存在多个shard时，需要先在一个shard检索Top K，再进行合并得到最终的Top K，扩展shard难以提升QPS
 - 同理，相同数据量下，part数越多，需要检索的向量索引越多，QPS越低
- 近似检索，无法通过Hash方式对数据进行划分和请求路由
- 如何扩展QPS:
 - ✓ 增加副本，QPS横向扩展
 - ✓ 基于聚类的方式对数据进行分片和路由，原始数据通过聚类分成不同的shards，在路由请求时，仅需要路由到少数几个shards

总结与展望

向量数据库 VS 专业Sim检索服务

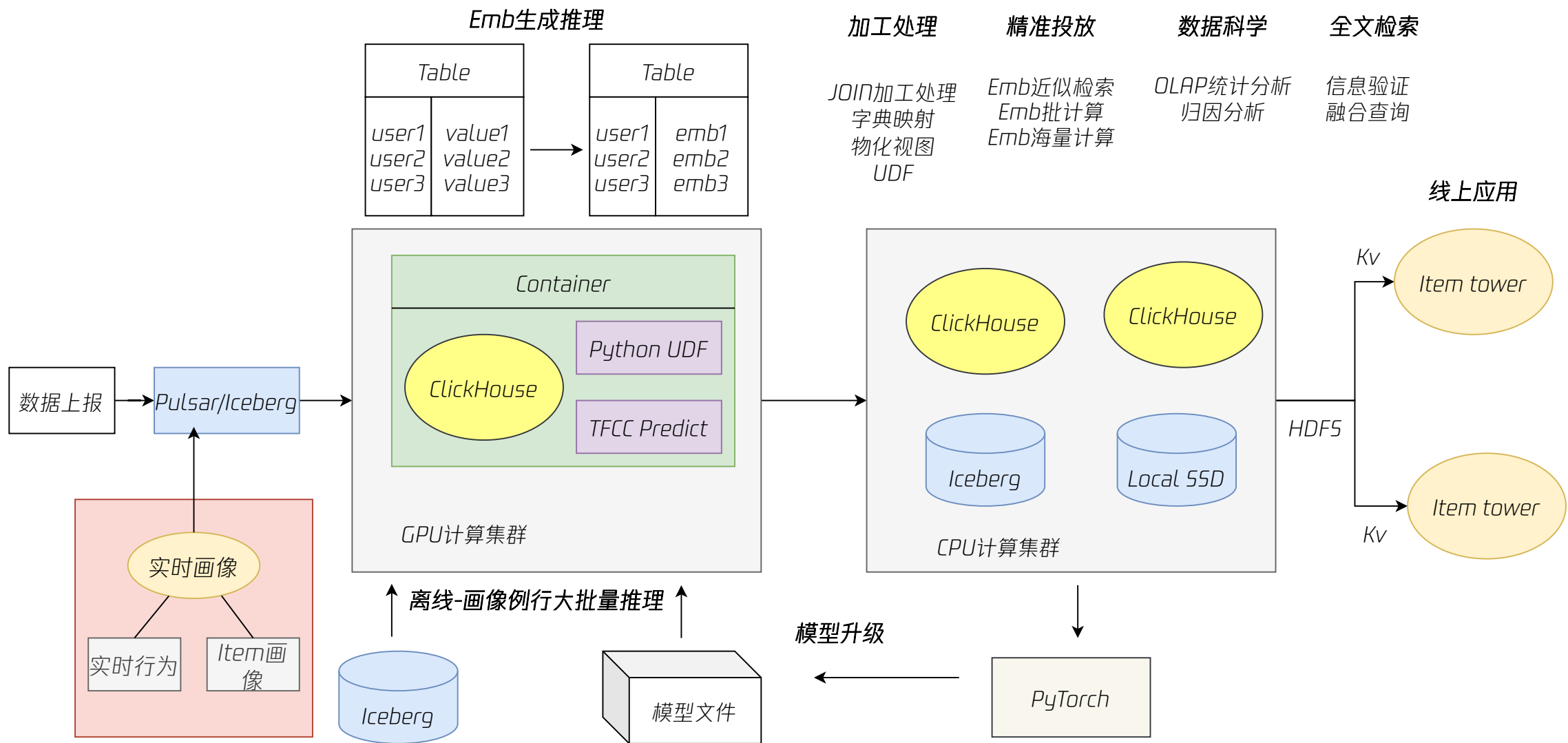
- 何时选择向量数据库:

- ✓ 数据正好需要在数仓内存储一份, 并且有“向量检索”以及“科学探索”需求, 偶尔进行批处理
- ✓ **ALL in ONE**: 科学探索, 全文检索, 数据加工Pipeline, 向量检索, 训练存储一体化, 省去多个存储系统之间的交互流程问题与多份存储
- ✓ **SQL表达**: 交互友好, 借助ClickHouse极致工程实现, 向量化引擎, 实现高效过滤、聚合统计
- ✓ **融合检索**: 全文检索+向量检索+统计分析+即时推理一站式SQL体验

- 何时选择专业Sim检索服务:

- ✓ **高QPS、低延迟场景**: 在线推荐服务召回通常有更高性能要求, 优秀的sim检索服务可提供100万qps的1ms低延迟查询, 且有98%以上的召回率
- ✓ **与业务系统的亲和度**: SIM检索服务对业务亲和度更高, 定制化专项需求更友好, 而标准数据库则更注重其通用性

基于ClickHouse的All in One AI Pipeline



End