# zh$_{d}ocs$

**发布 *0.1***

**2023 年 06 月 30 日**

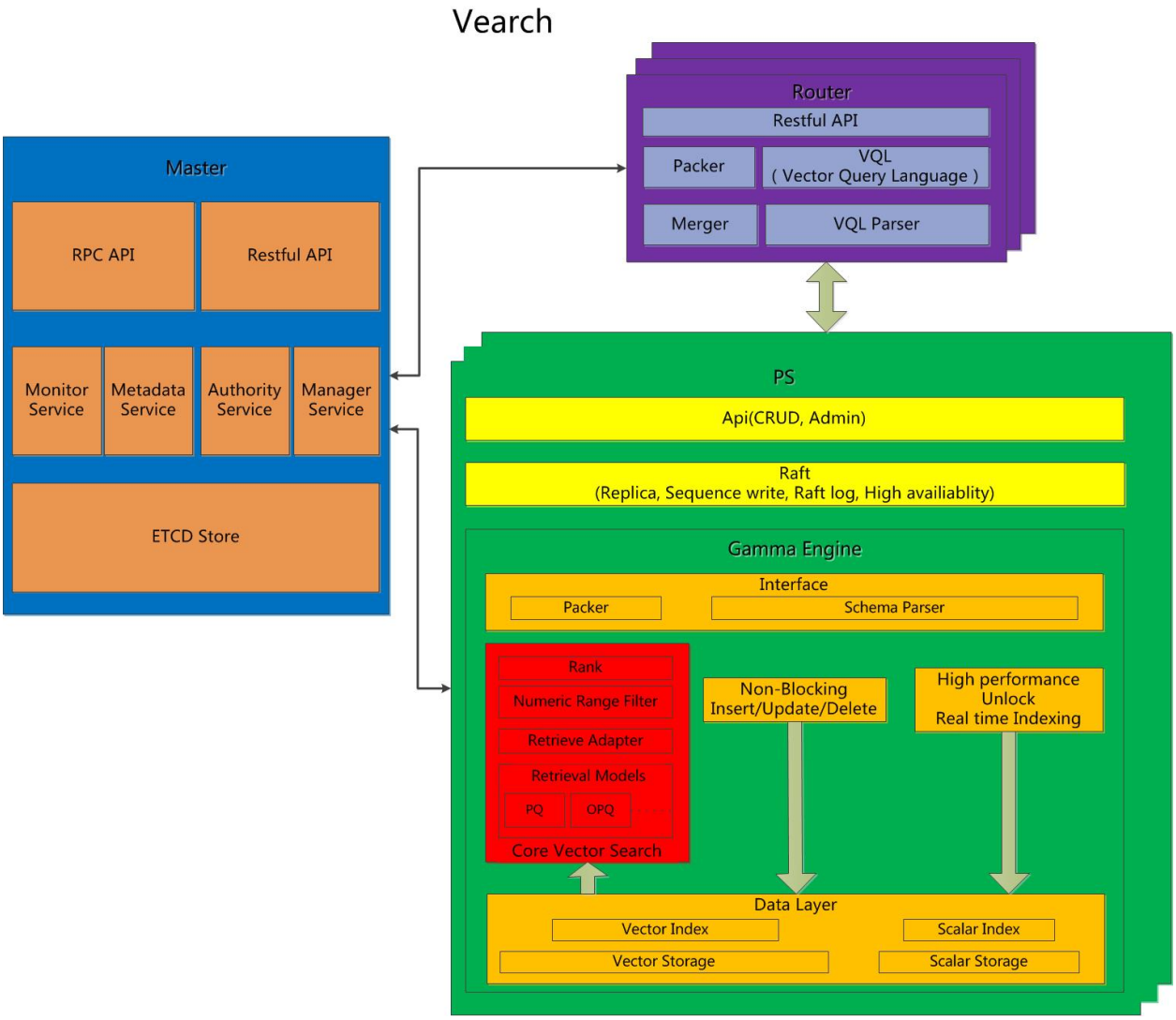
# 快速开始

CHAPTER 1

# 概述

Vearch 是对大规模深度学习向量进行高性能相似搜索的弹性分布式系统。

## 1.1 整体架构




数据模型: 空间，文档，向量，标量。

组件: Master，Router，PartitionServer。

Master: 负责 schema 管理，集群级别的源数据和资源协调。

Router: 提供 RESTful API: create、delete、search、update; 请求路由转发及结果合并。

PartitionServer(PS): 基于 raft 复制的文档分片; Gamma 向量搜索引擎, 它提供了存储、索引和检索向量、标量的能力。

## 1.2 功能简介

1、支持 CPU 与 GPU 两种版本。

2、支持实时添加数据到索引。

3、支持单个文档定义多个向量字段, 添加、搜索批量操作。

4、支持数值字段范围过滤与 string 字段标签过滤。

5、支持 IVFPQ、HNSW、二进制等索引方式 (HNSW、二进制方式 4 月下旬发布)。

6、支持 Python SDK 本地快速开发验证。

7、支持机器学习算法插件方便系统部署使用。

## 1.3 系统特性

1、自研 gamma 引擎，提供高性能的向量检索。

2、IVFPQ 倒排索引支持 compaction，检索性能不受文档更新次数的影响。

3、支持内存、磁盘两种数据存储方式，支持超大数据规模。

4、基于 raft 协议实现数据多副本存储。

5、支持内积 (InnerProduct) 与欧式距离 (L2) 方法计算向量距离。

CHAPTER 2

# 安装使用

## 2.1 编译

环境依赖

1. CentOS、ubuntu 和 Mac OS 都支持（推荐 CentOS >= 7.2）
2. go >= 1.11.2
3. gcc >= 5（使用 scann 模型时，gcc >= 9）
4. cmake >= 3.17
5. OpenBLAS
6. tbb，CentOS 可使用 yum 安装，如：yum install tbb-devel.x86_64
7. [RocksDB](https://github.com/facebook/rocksdb) == 6.2.2 （可选），你不需要手动安装，脚本自动安装。但是你需要手动安装 rocksdb 的依赖。请参考如下安装方法：https://github.com/facebook/rocksdb/blob/master/INSTALL.md
8. [zfp](https://github.com/LLNL/zfp) == v0.5.5 (可选)，你不需要手动安装，脚本自动安装。
9. CUDA >= 9.0，如果你不使用 GPU 模型，可忽略。
10. clang >= 8.0，bazel, python >= 3.7, 如果你不使用 scann 模型，可忽略。

编译

- 进入 *GOPATH* 目录, *cd $GOPATH/src mkdir -p github.com/vearch cd github.com/vearch*

- 下载源代码: `git clone https://github.com/vearch/vearch.git` (后续使用 $vearch 代表 vearch 目录绝对路径)
- 下载 gamma 代码: `cd vearch git submodule update --init --recursive`
- 如果使用 GPU 版本, 修改 $vearch/engine/CMakeLists.txt 文件中 BUILD_WITH_GPU 变为 on
- 如果使用 SCANN 模型, 修改 $vearch/engine/CMakeLists.txt 文件中 BUILD_WITH_SCANN 变为 on
- 编译 vearch 和 gamma

  1. `cd build`
  2. `sh build.sh`

  生成 `vearch` 文件表示编译成功

## 2.2 部署

单机模式:

- 生成配置文件 config.toml(master_server 端口使用 8817, router_server 端口使用 9001)

```
[global]
    # the name will validate join cluster by same name
    name = "vearch"
    # you data save to disk path ,If you are in a production environment, You'd better␣
↪set absolute paths
    data = ["datas/"]
    # log path , If you are in a production environment, You'd better set absolute paths
    log = "logs/"
    # default log type for any model
    level = "info"
    # master <-> ps <-> router will use this key to send or receive data
    signkey = "vearch"
    skip_auth = true

# if you are master, you'd better set all config for router、ps and router, ps use␣
↪default config it so cool
[[masters]]
    # name machine name for cluster
    name = "m1"
    # ip or domain
    address = "127.0.0.1"
```

(下页继续)

(续上页)

```
    # api port for http server
    api_port = 8817
    # port for etcd server
    etcd_port = 2378
    # listen_peer_urls List of comma separated URLs to listen on for peer traffic.
    # advertise_peer_urls List of this member's peer URLs to advertise to the rest of␣
↪the cluster. The URLs needed to be a comma-separated list.
    etcd_peer_port = 2390
    # List of this member's client URLs to advertise to the public.
    # The URLs needed to be a comma-separated list.
    # advertise_client_urls AND listen_client_urls
    etcd_client_port = 2370

[router]
    # port for server
    port = 9001

[ps]
    # port for server
    rpc_port = 8081
    # raft config begin
    raft_heartbeat_port = 8898
    raft_replicate_port = 8899
    heartbeat-interval = 200 #ms
    raft_retain_logs = 10000
    raft_replica_concurrency = 1
    raft_snap_concurrency = 1
```

- 启动

启动 vearch 前，需要设置 LD_LIBRARY_PATH 环境变量的值，添加 gamma, rocksdb, zfp 等依赖的 lib 包; 执行 ldd vearch 和 ldd $vearch/ps/engine/gammacb/lib/lib/libgamma.so.0.1 命令可以查看 vearch 和 gamma 依赖的包)。

```
./vearch -conf conf.toml all
```

集群模式:

- vearch 有三个模块: `ps`, `master`, `router`, run `./vearch -conf config.toml ps/router/master` 启动相应模块

假如有 5 台机器，2 台作为 master 管理，2 台作为 ps 计算节点，1 台 router 请求转发

- master
  - 192.168.1.1
  - 192.168.1.2
- ps
  - 192.168.1.3
  - 192.168.1.4
- router
  - 192.168.1.5
- 生成 toml 格式配置文件 config.toml，作为 master 的机器 ip 配置在 [[masters]] 中，支持多个，router 和 ps 所在机器 ip 无需配置。

```
[global]
    name = "vearch"
    data = ["datas/"]
    log = "logs/"
    level = "info"
    signkey = "vearch"
    skip_auth = true

# if you are master, you'd better set all config for router、ps and router, ps use␣
↪default config it so cool
[[masters]]
    name = "m1"
    address = "192.168.1.1"
    api_port = 8817
    etcd_port = 2378
    etcd_peer_port = 2390
    etcd_client_port = 2370
[[masters]]
    name = "m2"
    address = "192.168.1.2"
    api_port = 8817
    etcd_port = 2378
    etcd_peer_port = 2390
    etcd_client_port = 2370
[router]
    port = 9001
    skip_auth = true
```

(下页继续)

(续上页)

```
[ps]
    rpc_port = 8081
    raft_heartbeat_port = 8898
    raft_replicate_port = 8899
    heartbeat-interval = 200 #ms
    raft_retain_logs = 10000
    raft_replica_concurrency = 1
    raft_snap_concurrency = 1
```

- 启动 vearch 前，设置 LD_LIBRARY_PATH 环境变量加载依赖包
- on 192.168.1.1 , 192.168.1.2 run master

```
./vearch -conf config.toml master
```

- on 192.168.1.3 , 192.168.1.4 run ps

```
./vearch -conf config.toml ps
```

- on 192.168.1.5 run router

```
./vearch -conf config.toml router
```

CHAPTER 3

# 集群监控

http://master_server 代表 master 服务

## 3.1 集群状态

```
curl -XGET http://master_server/_cluster/stats
```

## 3.2 健康状态

```
curl -XGET http://master_server/_cluster/health
```

查看集群状态及库、表记录数据量 (doc_num)。

## 3.3 端口状态

```
curl -XGET http://master_server/list/server
```

## 3.4 清除锁

```
curl -XGET http://master_server/clean_lock
```

在创建表时会对集群加锁，若在此过程中，服务异常，会导致锁不能释放，需要手动清除才能新建表。

## 3.5 副本扩容缩容

```
curl -XPOST -H "content-type: application/json"  -d'
{
    "partition_id":1,
    "node_id": 1,
    "method": 0
}
' http://master_server/partition/change_member
```

method=0: node id 1 上添加分片 id 1 的副本; method=1: 删除 node id 1 上分片 id 1 的副本。

## 3.6 集群数据迁移

可以通过下述方式将某个集群的数据拷贝到新集群，实现集群的数据迁移。

1. 建立新的目标集群

新集群与待迁移集群节点数保持一致，部署完整的 vearch 系统，将新集群所有的 ps 节点的进程 kill。

2. 元数据同步

通过 etcd 的镜像功能，将待迁移集群的元数据，拷贝到目标集群。etcdctl 是 etcd 的客户端工具。

操作命令如下：

```
export ETCDCTL_API=3
# etcd 镜像的原理是读取一行一行的 key，写入到另外的集群中。其中：sourceMasterIP 是原始集群
master 的一个节点，targetMasterIP 是目标集群 master 的一个节点。
# ETCDCTL_API=3 ./etcdctl make-mirror target --endpoints=source
ETCDCTL_API=3 ./etcdctl make-mirror ${targetMasterIP}:2370 --endpoints=${sourceMasterIP}
↪:2370
```

3. 删除目标集群的/server 元信息

```
export ETCDCTL_API=3
./etcdctl --endpoints=http://${targetMasterIP}:2370 del /server --prefix
```

4. 拷贝向量数据

```
scp -r root@sourcePsIP:/export/vdb/baud root@targetPsIP:/export/vdb
... 省略其他 ip 的数据拷贝
```

sourcePsIP 是待迁移集群 PS 节点的 ip，targetPsIP 是目标集群 ps 节点的 ip。此处只需要保证待迁移集群与目标集群的 ps 节点 ip 一对一进行迁移即可，不需要特殊顺序。

CHAPTER 4

# 库操作

http://master_server 代表 master 服务，$db_name 是创建的库名

## 4.1 查看集群中所有库

```
curl -XGET http://master_server/list/db
```

## 4.2 创建库

```
curl -XPUT -H "content-type:application/json" -d '{
    "name": "db_name"
}
' http://master_server/db/_create
```

## 4.3 查看库

```
curl -XGET http://master_server/db/$db_name
```

## 4.4 删除库

```
curl -XDELETE http://master_server/db/$db_name
```

若库下存在表空间则无法删除

## 4.5 查看指定库下所有表空间

```
curl -XGET http://master_server/list/space?db=$db_name
```

CHAPTER 5

# 表空间操作

http://master_server 代表 master 服务，$db_name 是创建的库名, $space_name 是创建的表空间名

## 5.1 创建表空间

```
curl -XPUT -H "content-type: application/json" -d'
{
    "name": "space1",
    "partition_num": 1,
    "replica_num": 1,
    "engine": {
        "index_size": 70000,
        "id_type": "String",
        "retrieval_type": "IVFPQ",
        "retrieval_param": {
            "metric_type": "InnerProduct",
            "ncentroids": 2048,
            "nsubvector": 32
        }
    },
    "properties": {
        "field1": {
```

(下页继续)

(续上页)

```
            "type": "keyword"
        },
        "field2": {
            "type": "integer"
        },
        "field3": {
            "type": "float",
            "index": true
        },
        "field4": {
            "type": "string",
            "array": true,
            "index": true
        },
        "field5": {
            "type": "integer",
            "index": true
        },
        "field6": {
            "type": "vector",
            "dimension": 128
        },
        "field7": {
            "type": "vector",
            "dimension": 256,
            "format": "normalization",
            "store_type": "RocksDB",
            "store_param": {
                "cache_size": 2048,
                "compress": {"rate":16}
            }
        }
    }
}
' http://master_server/space/$db_name/_create
```

参数说明:

| 字段标识 | 字段含义 | 类型 | 是否必填 | 备注 |
|---|---|---|---|---|
| name | 空间名称 | string | 是 | |
| partition_num | 分片数量 | int | 是 | |
| replica_num | 副本数量 | int | 是 | |
| engine | 引擎配置 | json | 是 | 引擎配置 |
| properties | 空间配置 | json | 是 | 定义表字段及类型 |

1、name 不能为空，不能以数字或下划线开头，尽量不使用特殊字符等。

2、partition_num 指定表空间数据分片数量，不同的分片可分布在不同的机器，来避免单台机器的资源限制。

3、replica_num 副本数量，建议设置成 3，表示每个分片数据有两个备份，保证数据高可用。

engine 配置:

| 字段标识 | 字段含义 | 类型 | 是否必填 | 备注 |
|---|---|---|---|---|
| index_size | 分片索引阀值 | int | 否 | 默认 100000 |
| id_type | 唯一主键类型 | string | 否 | |
| retrieval_type | 检索模型 | string | 是 | |
| retrieval_param | 检索模型参数配置 | json | 否 | |

1、index_size 指定每个分片的记录数量达到多少开始创建索引，默认值 100000。

2、id_type 指定唯一记录主键类型，支持 String 和 Long(定义为 Long 可节省内存占用)。

3、retrieval_type 检索模型，目前支持六种类型，IVFPQ，HNSW，GPU，IVFFLAT，BINARYIVF，FLAT，不同的检索模型需要的参数配置及默认值如下:

IVFPQ:

现在 IVFPQ 可以与 HNSW 和 OPQ 组合使用。如果要使用 HNSW，建议将 ncentroids 设置为较大的值。对于 HNSW 的组合，现在解除了对训练数据量的限制，您可以使用不超过 ncentroids * 256 的数据进行训练。而在组合使用 OPQ 时，训练占用的内存为 2 * indexing_size * dimension * sizeof（float），因此对于 HNSW 和 OPQ 的组合使用，训练将占用更多的内存并花费较长时间，故要特别注意 indexing_size 的设置，防止使用的太多内存。

index_size：对于 IVFPQ，在建立索引之前需要训练，因此您应该将 index_size 设置为合适的值，例如 100000，如果与 HNSW 结合使用，index_size 可以是 ncentroids * 39 到 ncentroids * 256 之间的值。

如何组合使用 HNSW 和 OPQ 由 retrieve_param 控制。如果同时设置 HNSW 和 OPQ，则将使用 OPQ + IVF + HNSW + PQ，建议将 OPQ 的 nsubvector 设置为与 PQ 的 nsubvector 相同。如果只想使用 IVF + HNSW + PQ，则只需要设置 HNSW。如果您只想使用 IVFPQ，则无需在 retrieval_param 中设置 HNSW 或 OPQ。

| 字段标识 | 字段含义 | 类型 | 是否必填 | 备注 |
|---|---|---|---|---|
| metric_type | 计算方式 | string | 是 | L2 或者 InnerProduct |
| ncentroids | 聚类中心数量 | int | 是 | 默认 2048 |
| nsubvector | PQ 拆分子向量大小 | int | 是 | 默认 64 |
| bucket_init_size | 倒排链表（IVF）初始化的大小 | int | 否 | 默认 1000 |
| bucket_max_size | 倒排链表（IVF）最大容量 | int | 否 | 默认 1280000 |

```
"retrieval_type": "IVFPQ",
"retrieval_param": {
    "metric_type": "InnerProduct",
    "ncentroids": 2048,
    "nsubvector": 64
}
```

您可以这样设置 hnsw 或 opq：

```
"index_size": 2600000,
"id_type": "string",
"retrieval_type": "IVFPQ",
"retrieval_param": {
    "metric_type": "InnerProduct",
    "ncentroids": 65536,
    "nsubvector": 64,
    "hnsw" : {
        "nlinks": 32,
        "efConstruction": 200,
        "efSearch": 64
    },
    "opq": {
        "nsubvector": 64
    }
}
```

HNSW:

| 字段标识 | 字段含义 | 类型 | 是否必填 | 备注 |
|---|---|---|---|---|
| metric_type | 计算方式 | string | 是 | L2 或者 InnerProduct |
| nlinks | 节点邻居数量 | int | 是 | 默认 32 |
| efConstruction | 构图时寻找节点邻居过程中在图中遍历的深度 | int | 是 | 默认 40 |

```
"retrieval_type": "HNSW",
"retrieval_param": {
    "metric_type": "InnerProduct",
    "nlinks": 32,
    "efConstruction": 40
}

注意: 1、向量存储只支持 MemoryOnly
      2、创建索引不需要训练, index_size 值大于 0 均可
```

GPU（针对 GPU 编译版本）：

| 字段标识 | 字段含义 | 类型 | 是否必填 | 备注 |
|---|---|---|---|---|
| metric_type | 计算方式 | string | 是 | L2 或者 InnerProduct |
| ncentroids | 聚类中心数量 | int | 是 | 默认 2048 |
| nsubvector | PQ 拆分子向量大小 | int | 是 | 默认 64 |

```
"retrieval_type": "GPU",
"retrieval_param": {
    "metric_type": "InnerProduct",
    "ncentroids": 2048,
    "nsubvector": 64
}
```

SCANN（针对 SCANN 编译版本）：

| 字段标识 | 字段含义 | 类型 | 是否必填 | 备注 |
|---|---|---|---|---|
| metric_type | 计算方式 | string | 是 | L2 或者 InnerProduct |
| ncentroids | 聚类中心数量 | int | 是 | 默认 2048 |
| nsubvector | PQ 拆分子向量大小 | int | 是 | 默认 128, 量化为 4bit , 建议使用 ivfpq 模型 nsubvector 的 2 倍 |
| thread_num | 线程池线程数 | int | 否 | 可以不使用，如果使用建议为 cpu 核数 |

```
"retrieval_type": "VEARCH",
"retrieval_param": {
    "metric_type": "InnerProduct",
    "ncentroids": 2048,
    "nsubvector": 64,
    "thread_num": 8
}

注意: 1、目前 scann 模型，索引不支持 dump/load; 不支持 update。
```

IVFFLAT:

| 字段标识 | 字段含义 | 类型 | 是否必填 | 备注 |
|---|---|---|---|---|
| metric_type | 计算方式 | string | 是 | L2 或者 InnerProduct |
| ncentroids | 聚类中心数量 | int | 是 | 默认 2048 |

```
 "retrieval_type": "IVFFLAT",
 "retrieval_param": {
     "metric_type": "InnerProduct",
     "ncentroids": 2048
 }

注意: 1、向量存储方式只支持 RocksDB
```

BINARYIVF:

| 字段标识 | 字段含义 | 类型 | 是否必填 | 备注 |
|---|---|---|---|---|
| ncentroids | 聚类中心数量 | int | 是 | 默认 2048 |

```
"retrieval_type": "BINARYIVF",
"retrieval_param": {
    "ncentroids": 2048
}

注意: 1、向量长度是 8 的倍数
```

properties 配置:

1、表空间结构定义字段支持的类型 (即 type 的值) 有 6 种: string(keyword)，integer，long，float，double，vector。

2、string 类型的字段支持 index、array 属性，index 定义是否创建索引，array 指定是否允许多个值，创建索引后支持 term 过滤。

3、integer，long，float，double 类型的字段支持 index 属性，index 设为 true 创建索引后支持数值范围过滤查询 (range)。

4、vector 类型字段为特征字段，一个表空间中支持多个特征字段，vector 类型的字段支持的属性如下:

| 字段标识 | 字段含义 | 类型 | 是否必填 | 备注 |
|---|---|---|---|---|
| dimension | 特征维数 | int | 是 | |
| format | 归一化处理 | string | 否 | 设置为 normalization 对添加的特征向量归一化处理 |
| store_type | 特征存储类型 | string | 否 | 支持 MemoryOnly、Mmap 和 RocksDB, 默认 MemoryOnly |
| store_param | 存储参数设置 | json | 否 | 针对不同 store_type 的存储参数 |
| model_id | 特征插件模型 | string | 否 | 使用特征插件服务时指定 |

5、dimension 定义 type 是 vector 的字段，指定特征维数大小。

6、store_type 特征向量存储类型，有以下三个选项:

“MemoryOnly”: 原始向量都存储在内存中，存储数量的多少受内存限制，适用于数据量不大（千万级），对性能要求高的场景

“RocksDB”: 原始向量存储在 RockDB（磁盘）中，存储数量受磁盘大小限制，适用单机数据量巨大（亿级以上），对性能要求不高的场景

“Mmap”: 原始向量存储在磁盘文件中，使用 cache 提高性能，存储数量受磁盘大小限制，适用单机数据量巨大（亿级以上），对性能要求不高的场景

7、store_param 针对不同 store_type 的存储参数，其包含以下两个子参数。

cache_size: 数值类型，单位是 M bytes，默认 1024。store_type=" RocksDB" 时，表示 RocksDB 的读缓冲大小，值越大读向量的性能越好，一般设置 1024、2048、4096 和 6144 即可；store_type=" Mmap" 时，表示读缓冲的大小，一般 512、1024、2048 和 4096 即可，可根据实际应用场景设置大小；store_type=" MemoryOnly"，cache_size 不生效。

compress: 设置为 { "rate" :16} 压缩 50%；默认不压缩。

标量索引

Gamma 引擎支持标量索引，提供对标量数据的过滤功能，开启方式参考 "properties 配置" 中的第 2 条和第 3 条，检索方式参考 "查询" 中的 "filter json 结构说明"

## 5.2 查看表空间

```
curl -XGET http://master_server/space/$db_name/$space_name
```

## 5.3 删除表空间

```
curl -XDELETE http://master_server/space/$db_name/$space_name
```

## 5.4 修改 cache 大小

```
curl -H "content-type: application/json" -XPOST -d'
{
    "cache_models": [
        {
            "name": "table",
            "cache_size": 1024,
        },
        {
            "name": "string",
            "cache_size": 1024,
        },
        {
            "name": "field7",
            "cache_size": 1024,
        }
    ]
```

(下页继续)

(续上页)

```
}
' http://master_server/config/$db_name/$space_name
```

1、table cache size：表示所有定长的标量字段（integer，long，float，double）使用 cache 的大小，默认为 512M，单位为 M bytes。

2、string cache size：表示所有变长的标量字段（string）使用 cache 的大小，默认为 512M，单位为 M bytes。

3、对于向量字段只支持 store_type 为 Mmap 的进行修改 cache size。

## 5.5 查看 cache 大小

```
curl -XGET http://master_server/config/$db_name/$space_name
```

1、对于向量字段只支持 store_type 为 Mmap 的查看 cache size。

CHAPTER 6

# 数据操作

http://router_server 代表 router 服务，$db_name 是创建的库名, $space_name 是创建的空间名, $id 是数据记录的唯一 id.

## 6.1 单条插入

插入时不指定唯一标识 id

```
curl -XPOST -H "content-type: application/json"  -d'
{
    "field1": "value1",
    "field2": "value2",
    "field3": {
        "feature": [0.1, 0.2]
    }
}
' http://router_server/$db_name/$space_name
```

field1 和 field2 是标量字段，field3 是特征字段。所有字段名、值类型和定义表结构时保持一致。

返回值格式如下:

```
{
    "_index": "db1",
```

(下页继续)

(续上页)

```
    "_type": "space1",
    "_id": "AW5J1lNmJG6WbbCkHrFW",
    "status": 200
}
```

其中 _index 库名称，_type 表空间名称，_id 是服务端生成的记录唯一标识，可以由用户指定，对数据的修改和删除需要使用该唯一标识。

插入时指定唯一标识

```
curl -XPOST -H "content-type: application/json"  -d'
{
    "field1": "value1",
    "field2": "value2",
    "field3": {
        "feature": [0.1, 0.2]
    }
}
' http://router_server/$db_name/$space_name/$id
```

$id 是插入数据时使用指定的值替换服务端生成的唯一标识，$id 值不能使用 url 路径等特殊字符。若库中已存在该唯一标识的记录则覆盖。

## 6.2 批量插入

```
curl -H "content-type: application/json" -XPOST -d'
{"index": {"_id": "v1"}}\n
{"field1": "value", "field2": {"feature": []}}\n
{"index": {"_id": "v2"}}\n
{"field1": "value", "field2": {"feature": []}}\n
' http://router_server/$db_name/$space_name/_bulk
```

json 格式的变体，{ “index” : { “_id” : “v1” }} 指定记录的 id, _id 值为空后台自动生成唯一 id, { “field1” : “value” , “field2” : { “feature” : []}} 指定插入的数据，每行 json 字符串均以\n 结尾。

## 6.3 更新

更新时必须指定唯一标识 id

```
curl -H "content-type: application/json" -XPOST -d'
{
    "field1": "value1",
    "field2": "value2",
    "field3": {
        "feature": [0.1, 0.2]
    }
}
' http://router_server/$db_name/$space_name/$id/_update
```

请求路径中指定唯一标识 $id，使用指定 id 插入的方式进行数据覆盖更新 (后续可支持单个字段修改)。

## 6.4 删除

根据唯一 id 标识删除数据

```
curl -XDELETE http://router_server/$db_name/$space_name/$id
```

根据查询过滤结果删除数据

```
curl -H "content-type: application/json" -XPOST -d'
{
    "query": {
        "sum": [{}]
    }
}
' http://router_server/$db_name/$space_name/_delete_by_query
```

查询详细语法见下文

## 6.5 查询

查询示例

```
curl -H "content-type: application/json" -XPOST -d'
{
    "query": {
        "sum": [{
            "field": "field_name",
            "feature": [0.1, 0.2, 0.3, 0.4, 0.5],
```

(下页继续)

(续上页)

```
            "min_score": 0.9,
            "boost": 0.5
        }],
        "filter": [{
            "range": {
                "field_name": {
                    "gte": 160,
                    "lte": 180
                }
            }
        },
        {
             "term": {
                 "field_name": ["100", "200", "300"],
                 "operator": "or"
             }
        }]
    },
    "retrieval_param": {
        "nprobe": 20
    },
    "fields": ["field1", "field2"],
    "is_brute_search": 0,
    "online_log_level": "debug",
    "quick": false,
    "vector_value": false,
    "client_type": "leader",
    "l2_sqrt": false,
    "sort": [{"field1":{"order": "asc"}}],
    "size": 10
}
' http://router_server/$db_name/$space_name/_search
```

查询参数整体 json 结构如下:

```
{
    "query": {
        "sum": [],
        "filter": []
    },
```

(下页继续)

(续上页)

```
    "retrieval_param": {"nprobe": 20},
    "fields": ["field1", "field2"],
    "is_brute_search": 0,
    "online_log_level": "debug",
    "quick": false,
    "vector_value": false,
    "client_type": "leader",
    "l2_sqrt": false,
    "sort": [{"field1":{"order": "asc"}}],
    "size": 10
}
```

参数说明:

| 字段标识 | 类型 | 是否必填 | 备注 |
|---|---|---|---|
| sum | json 数组 | 是 | 查询特征 |
| filter | json 数组 | 否 | 查询条件过滤: 数值过滤 + 标签过滤 |
| fields | json 数组 | 否 | 指定返回那些字段, 默认只返回唯一 id 和分值 |
| is_brute_search | int | 否 | 默认 0 |
| online_log_level | string | 否 | 值为 debug，开启打印调试日志 |
| quick | bool | 否 | 默认 false |
| vector_value | bool | 否 | 默认 false |
| client_type | string | 否 | 默认 leader |
| ivf_flat | bool | 否 | 默认 false, 仅适用于 IVFPQ 模型, 结果开根号 |
| sort | json 数组 | 否 | 指定字段排序 (只针对匹配结果，非整体) |
| size | int | 否 | 指定返回结果数量, 默认 50 |

retrieval_param 参数指定模型计算时的参数，不同模型支持的参数不同，如下示例:

- metric_type: 计算类型，支持 InnerProduct 和 L2, 默认 L2。
- nprobe: 搜索桶数量。
- recall_num: 召回数量，默认等于查询参数中 size 的值，设置从索引中搜索数量，然后计算 size 个最相近的值。
- parallel_on_queries: 默认 1，搜索间并行；0 代表桶间并行。
- efSearch: 图遍历的距离。

IVFPQ:

```
"retrieval_param": {
    "parallel_on_queries": 1,
    "recall_num" : 100,
    "nprobe": 80,
    "metric_type": "L2"
}
```

GPU:

```
"retrieval_param": {
    "recall_num" : 100,
    "nprobe": 80,
    "metric_type": "L2"
}
```

HNSW:

```
"retrieval_param": {
    "efSearch": 64,
    "metric_type": "L2"
}
```

IVFFLAT:

```
"retrieval_param": {
    "parallel_on_queries": 1,
    "nprobe": 80,
    "metric_type": "L2"
}
```

FLAT:

```
"retrieval_param": {
    "metric_type": "L2"
}
```

- sum json 结构说明:

```
"sum": [{
          "field": "field_name",
          "feature": [0.1, 0.2, 0.3, 0.4, 0.5],
          "min_score": 0.9,
```

(下页继续)

(续上页)

```
        "boost": 0.5
    }]
```

(1) sum 支持多个 (对应定义表结构时包含多个特征字段)。

(2) field 指定创建表时特征字段的名称。

(3) feature 传递特征，维数和定义表结构时维数必须相同。

(4) min_score 指定返回结果中分值必须大于等于 0.9，两个向量计算结果相似度在 0-1 之间，min_score 可以指定返回结果分值最小值，max_score 可以指定最大值。如设置: “min_score” : 0.8, “max_score” : 0.95 代表过滤 0.8<= 分值 <= 0.95 的结果。同时另外一种分值过滤的方式是使用: “symbol” :” >=” , ” value” :0.9 这种组合方式，symbol 支持的值类型包含: > 、>= 、<、<= 4 种，value 及 min_score、max_score 值在 0 到 1 之间。

(5) boost 指定相似度的权重，比如两个向量相似度分值是 0.7，boost 设置成 0.5 之后，返回的结果中会将分值 0.7 乘以 0.5 即 0.35。

- filter json 结构说明:

```
"filter": [
            {
                "range": {
                    "field_name": {
                         "gte": 160,
                         "lte": 180
                    }
                }
            },
            {
                "term": {
                    "field1": ["100", "200", "300"],
                    "operator": "or"
                }
            },
            {
                "term": {
                    "field2": ["a", "b", "c"],
                    "operator": "and"
                }
            },
            {
                "term": {
```

(下页继续)

(续上页)

```
                "field3": ["A1", "B2"],
                "operator": "not"
            }
        }
    ]
```

(1) filter 条件支持多个，多个条件之间是交的关系。

(2) range 指定使用数值字段 integer、long、float、double 过滤，filed_name 是数值字段名称，gte、lte 指定范围，lte 小于等于，gte 大于等于，若使用等值过滤，lte 和 gte 设置相同的值。上述示例表示查询 field_name 字段大于等于 160 小于等于 180 区间的值。

(3) term 使用标签过滤（string 字段），field1 是定义的标签字段名，允许使用多个值过滤，可以求并“operator”：“or”，求交：“operator”：“and”，不包含：“operator”：“not”。

- is_brute_search 0 使用索引搜索（建完索引前查询结果为空），1 使用暴力搜索，默认值 0。
- online_log_level 设置成”debug”可以指定在服务端打印更加详细的日志，开发测试阶段方便排查问题。
- quick 搜索结果默认将 PQ 召回向量进行计算和精排，为了加快服务端处理速度设置成 true 可以指定只召回，不做计算和精排。
- vector_value 为了减小网络开销，搜索结果中默认不包含特征数据只包含标量信息字段，设置成 true 指定返回结果中包含原始特征数据。
- client_type leader，random，no_leader，默认 leader 仅从主数据节点查询，random: 从 ps 主从节点随机选择，no_leader: 只查询从节点。
- size 指定最多返回的结果数量。若请求 url 中设置了 size 值 http://router_server/$db_name/$space_name/_search?size=20 优先使用 url 中指定的 size 值。

## 6.6 id 查询

```
curl -XGET http://router_server/$db_name/$space_name/$id
```

## 6.7 批量 id 查询

```
curl -H "content-type: application/json" -XPOST -d'
{
    "query": {
              "ids": ["id1", "id2"],
```

(下页继续)

(续上页)

```
            "fields": ["field1"]
    }
}
' http://router_server/$db_name/$space_name/_query_byids
```

ids 指定多个 id，fields 指定返回每条记录中那些字段

## 6.8 批量特征查询 1

```
curl -H "content-type: application/json" -XPOST -d'
[{
   "query": {
       "sum": [{
           "field": "vector_field_name",
           "feature": [0.1, 0.2]
       }]
   }
},
{
   "query": {
       "sum": [{
           "field": "vector_field_name",
           "feature": [0.1, 0.2]
       }]
    }
}]
' http://router_server/$db_name/$space_name/_bulk_search
```

把多个单条查询的参数拼接成数组作为请求参数，返回结果和请求顺序保持一致。

## 6.9 批量特征查询 2

```
curl -H "content-type: application/json" -XPOST -d'
{
    "query": {
        "sum": [{
            "field": "vector_field_name",
            "feature": [0.1, 0.2]
```

(下页继续)

(续上页)

```
        }]
    }
}
' http://router_server/$db_name/$space_name/_msearch
```

适用于多个查询使用相同过滤条件的情况，将多个查询的特征拼接成一个特征数组（比如定义 128 维的特征，批量查询 10 条，则将 10 个 128 维特征按顺序拼接成 1280 维特征数组赋值给 feature 字段），后台接收到请求后按表结构定义的特征字段维度进行拆分，按顺序返回匹配结果。

## 6.10 根据 id 特征查询

```
curl -H "content-type: application/json" -XPOST -d'
{
    "query": {
        "sum":[{"field": "field_name", "feature": []}],
        "ids": ["id1", "id2"]
     },
     "size": 10
}
' http://router_server/$db_name/$space_name/_query_byids_feature
```

sum 条件中 field_name 指定特征字段名称，feature 设为空，ids 传入唯一记录 id，后台处理首先根据唯一 id 查询出该记录的特征，然后再用特征进行相似查询，返回匹配结果。

## 6.11 多向量查询

表空间定义时支持多个特征字段，因此查询时可以支持相应数据的特征进行查询。以每条记录两个向量为例：
定义表结构字段

```
{
    "field1": {
        "type": "vector",
        "dimension": 128
    },
    "field2": {
        "type": "vector",
        "dimension": 256
```

(下页继续)

(续上页)

```
    }
}
```

field1、field2 均为向量字段，查询时搜索条件可以指定两个向量：

```
{
    "query": {
        "sum": [{
            "field": "filed1",
            "feature": [0.1, 0.2, 0.3, 0.4, 0.5],
            "min_score": 0.9
        },
        {
            "field": "filed2",
            "feature": [0.8, 0.9],
            "min_score": 0.8
        }]
    }
}
```

field1 和 field2 过滤的结果求交集，其他参数及请求地址和普通查询一致。

CHAPTER 7

# GPU 使用

## 7.1 注意事项

1. 编译环境需要带 GPU 安装 CUDA>=9.0.
2. 编译前 Vearch 引擎 CMakeLists.txt 配置中 BUILD_WITH_GPU 设置为 on.
3. 创建表时 gamm 设置参数 nprobe 不大于 1024(CUDA9.0) 或者不大于 2048(CUDA>=9.2), index_size 设置为 0.
4. 建表时特征字段 retrieval_type 参数设置为 GPU.
5. 数据插入和建索引时不支持搜索。
6. 数据不自动创建索引，调用 curl -XPOST http://router_server/\protect___xunadd_text_character:nN{\textdollar}{$}db_name/\protect\___xunadd_text_character:nN{\textdollar}{$}space_name/_forcemerge 创建索引。
7. 不支持实时添加数据到 GPU 索引，新增数据只有更新索引后才会生效。(更新索引:curl -XPOST http://router_server/\protect___xunadd_text_character:nN{\textdollar}{$}db_name/\protect\___xunadd_text_character:nN{\textdollar}{$}space_name/_forcemerge)。

CHAPTER 8

# Python SDK

参照文档:https://github.com/vearch/vearch/blob/master/docs/APIPythonSDK.md

CHAPTER 9

# 算法插件

参照文档:https://github.com/vearch/vearch/blob/master/docs/Quickstart.md

CHAPTER 10

# 效果评估

**基准**

本文档显示了我们做的实验和得到的结果。我们做了两个系列的实验。首先，我们在单个节点上进行了基于 faiss 的改进的 Vearch ivfpq 模型的召回率实验。其次，基于 Vearch 集群进行了实验。

我们使用检索返回的 k 个候选值（k {1，10，100}）是否包含最近邻的结果来计算召回，其中标签通过暴力搜索获得。

实验数据会因为实现、不同机器等的变化而略有变化。

**数据集**

实验分别在 128 维 SIFT 特征和 512 维 VGG 特征上进行。

**数据集 SIFT1M**

可以从如下网址下载 ANN_SIFT1M

http://corpus-texmex.irisa.fr/

**数据集 VGG1M 和 VGG10M**

分别收集 100 万和 1000 万的图片数据提取 VGG 特征得到 VGG1M （100 万）和 VGG10M（1000 万），其中 VGG1M 和 VGG10M 并不相关。

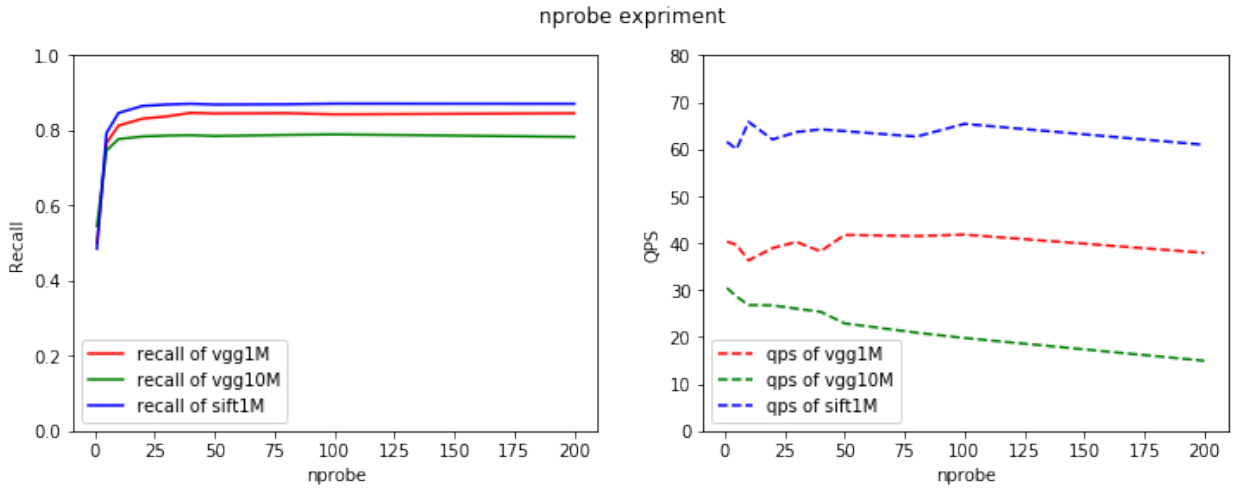
**数据集 VGG100M , VGG500M 和 VGG1B**

另外收集了 10 亿图片数据来构建 VGG100M(1 亿) , VGG500M（5 亿）和 VGG1B（10 亿）。

**Nprobe 实验**

实验分别在 SIFT1M, VGG1M 和 VGG10M 上进行。其中 ncentroids =256, nbytes = 32, nprobe {1,5,10,20,30,40,50,80,100,200}。图中数据为 recall@1 结果。

**结果**

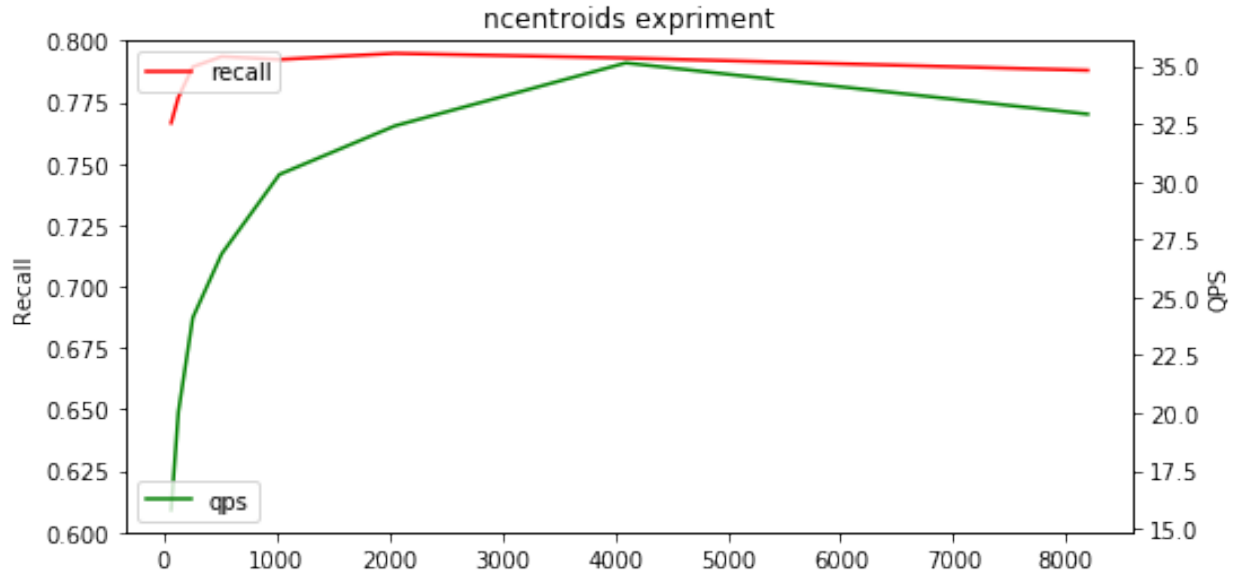



可以看到当 nprobe 超过 25 后，召回基本上没有明显的变化了。

**Ncentroids 实验**

实验在 VGG10M 进行。其中 nprobe = 50，nbytes = 32，ncentroids {64,128,256,512,1024,2048,4096,8192} 。图中数据为 recall@1 结果。

**结果**



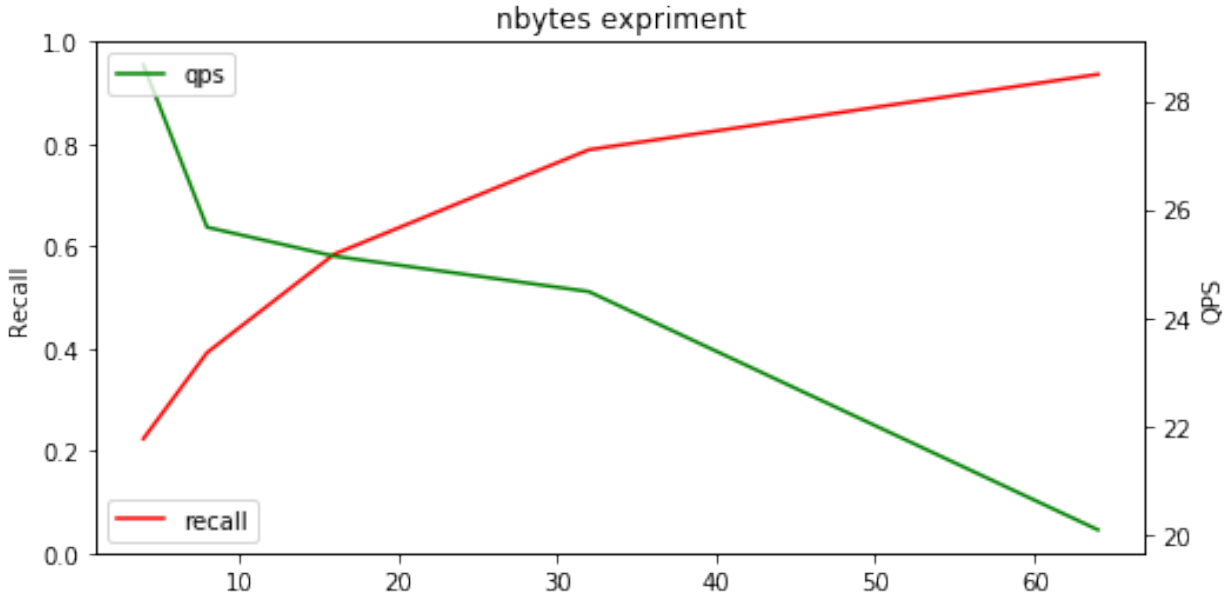

可以看到召回基本不随 ncentroids 变化而变化，但是 ncentroids 越大，QPS 越高。

**Nbytes 实验**

实验在 VGG10M 进行。其中 nprobe = 50，ncentroids = 256，nbytes {4,8,16,32,64}。图中数据为 recall@1

结果。

**结果**




当 nbytes 越大，召回越高，当然 QPS 随之降低。

**对比实验**

实验在 SIFT1M, VGG1M 和 VGG10M 上进行，并与 faiss 中的一些模型进行对比。

**模型参数**

表格中参数为空，则对应模型不包含该参数。其中 links, efSearch 和 efConstruction 为 faiss 中的 hnsw 定义的参数。

| model | ncentroids | nprobe | bytes of SIFT | bytes of VGG | links | efSearch | efConstruction |
|---|---|---|---|---|---|---|---|
| pq \| | | | 32 | 64 | | | |
| ivfpq \|256 | | 20 | 32 | 64 | | | |
| imipq | 2^(2*10) | 2048 | 32 | 64 | | | |
| opq+pq | | | 32 | 64 | | | |
| hnsw | | | | | 32 | 64 | 40 |
| ivfhnsw | 256 | 20 | | | 32 | 64 | 40 |
| Vearch | 256 | 20 | 32 | 64 | | | |

**结果**

SIFT1M 的召回:

| model | recall@1 | recall@10 | recall@100 |
|---|---|---|---|
| pq | 0.6274 | 0.9829 | 0.9999 |
| ivfpq | 0.6167 | 0.9797 | 0.9960 |
| imipq | 0.6595 | 0.9775 | 0.9841 |
| opq+pq | 0.6250 | 0.9821 | 1.0000 |
| hnsw | 0.9792 | 0.9867 | 0.9867 |
| ivfhnsw | 0.9888 | 0.9961 | 0.9961 |
| Vearch | 0.9581 | 0.9645 | 0.9645 |

VGG1M 的召回:

| model | recall@1 | recall@10 | recall@100 |
|---|---|---|---|
| pq | 0.5079 | 0.8922 | 0.9930 |
| ivfpq | 0.4985 | 0.8792 | 0.9704 |
| imipq | 0.5077 | 0.8618 | 0.9248 |
| opq+pq | 0.5213 | 0.9105 | 0.9975 |
| hnsw | 0.9496 | 0.9550 | 0.9551 |
| ivfhnsw | 0.9690 | 0.9744 | 0.9745 |
| Vearch | 0.9536 | 0.9582 | 0.9585 |

VGG10M 的召回:

| model | recall@1 | recall@10 | recall@100 |
|---|---|---|---|
| pq | 0.5842 | 0.8980 | 0.9888 |
| ivfpq | 0.5913 | 0.8896 | 0.9748 |
| imipq | 0.5925 | 0.8878 | 0.9570 |
| opq+pq | 0.6126 | 0.9160 | 0.9944 |
| hnsw | 0.8877 | 0.9069 | 0.9074 |
| ivfhnsw | 0.9638 | 0.9839 | 0.9843 |
| Vearch | 0.9272 | 0.9464 | 0.9468 |

CHAPTER 11

# 集群实验

集群实验分别对 VGG100M , VGG500M 和 VGG1B 进行实验，并添加是否过滤来进行实验，其中过滤是指在搜索的时候指定过滤条件来缩小搜索范围。VGG100M 搭建了 3 个 masters, 3 个 routers 和 5 个 partition services 的集群。VGG500M 搭建了 3 个 masters, 3 个 routers 和 24 个 partition services 的集群。VGG1B 搭建了 3 个 masters, 6 个 routers 和 48 个 partition services 的集群。

**结果**


可以看到当 average latency 超过一定程度，QPS 就不再发生明显变化了。

CHAPTER 12

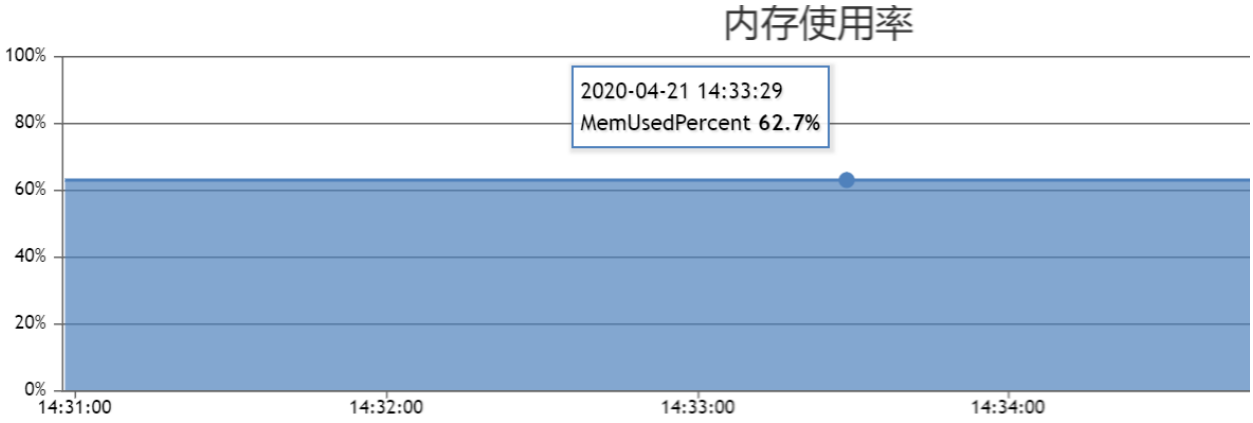
# HNSW 性能评测

data_size = 10M, 参数为 InnerProduct, nlinks = 32, efConstruction = 40, efSearch = 64

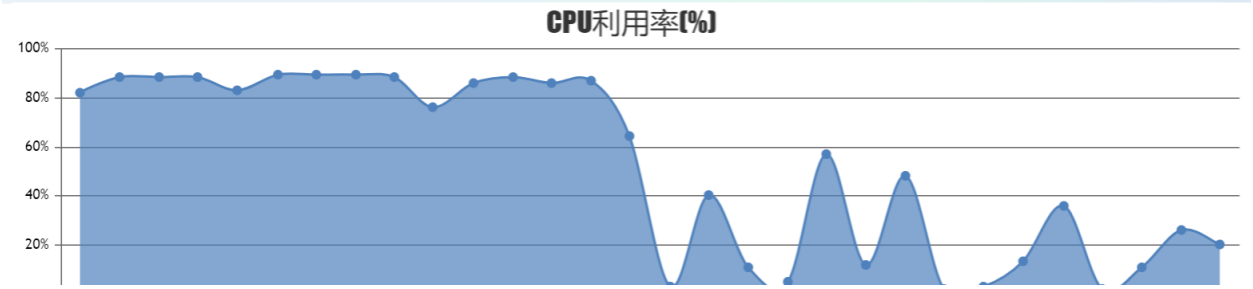
| 集群 | 机器配置 | 数据维度 | 构建索引时间 |
|---|---|---|---|
| masterrouterps 各一个，均在一台机器上 | 56 核 256G 内存 | 128 | 28min |
| masterrouterps 各一个，均在一台机器上 | 56 核 256G 内存 | 512 | 33min10s |
| masterrouterps 各一台，且在不同机器上 | 8 核 16G 内存 | 128 | 2h38m |

**内存使用率**

以 8 核 16G 机器为样本进行分析，vearch ps 共使用内存 16G * 0.627 = 10.032G，其中原始数据特征大小为 10M * 128 * 4 ~ 5G，其它为索引以及正排字段占用内存。使用时仅创建一个正排字段和向量字段。

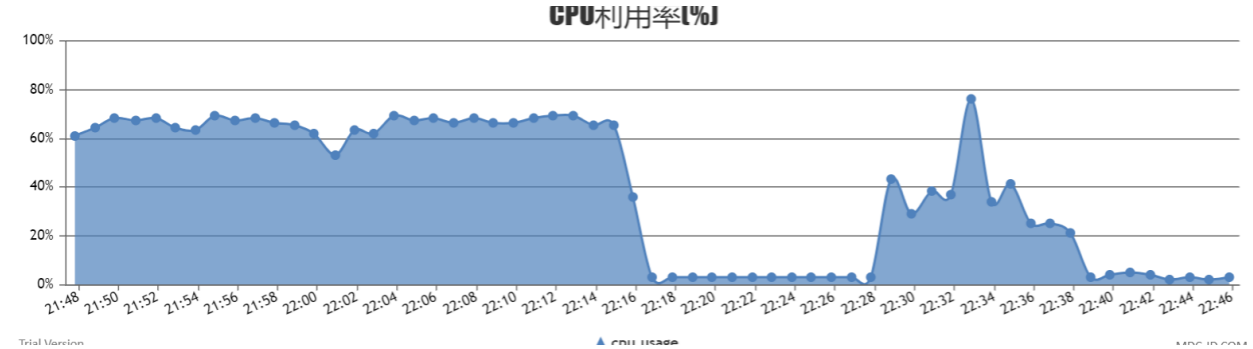


10M 数据, 机器配置 56 核 256G，add time = 28min01s, 左侧是构建索引 CPU 利用率，右侧是搜索时 CPU 利用率




| model | 返回结果数 | 并发数 | QPS | tp99 | tp100 |
|---|---|---|---|---|---|
| HNSW | 100 | 2000 | 5069.48 | 959 | 1944 |
| HNSW | 100 | 1000 | 4680.62 | 521 | 994 |
| HNSW | 100 | 100 | 4508.33 | 49 | 86 |
| HNSW | 100 | 50 | 4146.55 | 22 | 35 |
| HNSW | 100 | 10 | 2048.43 | 8 | 117 |
| HNSW | 100 | 1 | 182.74 | 9 | 160 |

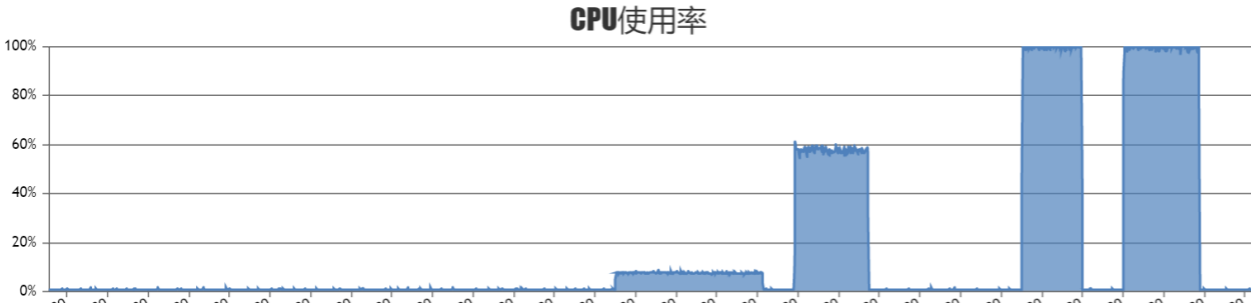
10M 数据，机器配置 56 核 256G，d = 512, add time = 33min10s，左侧是构建索引 CPU 利用率，右侧是搜索时 CPU 利用率




| model | 返回结果数 | 并发数 | QPS | tp99 | tp100 |
|---|---|---|---|---|---|
| HNSW | 100 | 2000 | 4194.72 | 1181 | 2650 |
| HNSW | 100 | 1000 | 4058.59 | 628 | 1059 |
| HNSW | 100 | 100 | 3784.56 | 57 | 172 |
| HNSW | 100 | 50 | 3723.75 | 24 | 43 |
| HNSW | 100 | 10 | 1248.46 | 15 | 1208 |
| HNSW | 100 | 1 | 142.70 | 12 | 71 |

10m 数据，机器配置 8 核 16G，d = 128, add_time = 2h38m

构建索引时 CPU 使用率基本保持在 100%，搜索时 cpu 使用率，当并发数为 50，100 时，可以看到 cpu 使用率已经 100%，故不做并发数 1000/2000 的对比实验




| model | 返回结果数 | 并发数 | QPS | tp99 | tp100 |
|---|---|---|---|---|---|
| HNSW | 100 | 100 | 888.25 | 279 | 402 |
| HNSW | 100 | 50 | 1126.06 | 95 | 199 |
| HNSW | 100 | 10 | 918.50 | 15 | 31 |
| HNSW | 100 | 1 | 91.74 | 14 | 25 |

召回评测使用 sift1M

| model | parameters | re-call@1 | re-call@10 | re-call@100 |
|---|---|---|---|---|
| HNSW | InnerProduct, nlinks = 32, efConstruction = 40, efSearch = 64 | 0.9769 | 0.9852 | 0.9852 |

CHAPTER 13

# GPU 版性能

**服务器配置**

| hardware | config |
|---|---|
| CPU | E5-2683 v4 16cores |
| memory | 16G |
| GPU | Tesla P40 |

数据：128 维 float 数据，总插入数据量 2 亿

性能：并发 20 查询，QPS 达到 3000，tp99 在 30ms 以内

CHAPTER 14

# 常见问题

1. Vearch 的向量搜索引擎 gamma 基于 faiss 实现，faiss 版本有较大改动不兼容历史版本时 Vearch 可能编译不成功。