Milvus v1.1.1

Contents

关于 Milvus 2
- Milvus 是什么 . . . 2
- v1.0 发布公告 . . . 4
- Milvus 数据迁移指南 . . . 5
- 向量索引 . . . 7
- 存储相关 . . . 12
- 存储操作 . . . 13
- 相似度计算方式 . . . 17
- 预写式日志 . . . 21
- Milvus 术语 . . . 22
- 安装、启动 Milvus 服务 . . . 22
- Hello Milvus . . . 25
- 连接服务端 . . . 25
- 创建、删除集合 . . . 26
- 创建、删除分区 . . . 27
- 插入、删除向量 . . . 27
- 创建、删除索引 . . . 28
- 查询向量 . . . 29
- 数据落盘 . . . 30
- 数据段整理 . . . 30
- 关闭客户端 . . . 30

参考手册 30
- 服务端配置 . . . 30
- 使用 MySQL 管理元数据 . . . 34
- 监控与报警 . . . 35
- 性能调优 . . . 38
- Mishards: 集群分片中间件 . . . 41

常见问题 51
- 性能优化问题 . . . 51
- 产品问题 . . . 52
- 部署运维问题 . . . 56
- 故障诊断 . . . 58

API 参考 59

开发工具 59
- 概述 . . . 59

发版说明 64

关于 Milvus

Milvus 是什么

Milvus 是一款开源的向量数据库，支持针对 TB 级向量的增删改操作和近实时查询，具有高度灵活、稳定可靠以及高速查询等特点。Milvus 集成了 Faiss、NMSLIB、Annoy 等广泛应用的向量索引库，提供了一整套简单直观的 API，让你可以针对不同场景选择不同的索引类型。此外，Milvus 还可以对标量数据进行过滤，进一步提高了召回率，增强了搜索的灵活性。

Milvus 服务器采用主从式架构 (Client-server model)。

- 在服务端，Milvus 由 Milvus Core 和 Meta Store 两部分组成:
  - Milvus Core 存储与管理向量和标量数据。
  - Meta Store 存储与管理 SQLite 和 MySQL 中的元数据，分别用于测试和生产。
- 在客户端，Milvus 还提供了基于 Python、Java、Go、C++ 的 SDK 和 RESTful API。

Milvus 在 Apache 2 License 协议下发布，于 2019 年 10 月正式开源，是 LF AI & DATA 基金会的毕业项目。Milvus 的源代码被托管于 Github。

> 目前，Milvus 的服务器在单节点上运行。对于有更大数据规模或者高并发需求的用户，可以使用目前尚在开发阶段的集群分片中间件 Mishards 进行部署。

整体架构

应用场景

Milvus 在全球范围内已被数百家组织和机构所采用，广泛应用于以下场景:

- 图像、视频、音频等音视频搜索领域
- 文本搜索、推荐和交互式问答系统等文本搜索领域
- 新药搜索、基因筛选等生物医药领域

详见 应用场景。

主要特性

异构计算

- 优化了基于 GPU 搜索向量和建立索引的性能。
- 可以在单台通用服务器上完成对 TB 级数据的毫秒级搜索。
- 动态数据管理。

支持主流索引库、距离计算方式和监控工具

- 集成了 Faiss、NMSLIB、Annoy 等向量索引库。
- 支持基于量化的索引、基于图的索引和基于树的索引。
- 相似度计算方式包括欧氏距离 (L2)、内积 (IP)、汉明距离、杰卡德距离等。
- Prometheus 作为监控和性能指标存储方案，Grafana 作为可视化组件进行数据展示。

近实时搜索

- 插入 Milvus 的数据默认在 1 秒后即可被搜索到。

标量字段过滤 (即将上线)

- 支持向量和标量数据。
- 可以对标量数据进行过滤，增强搜索的灵活性。

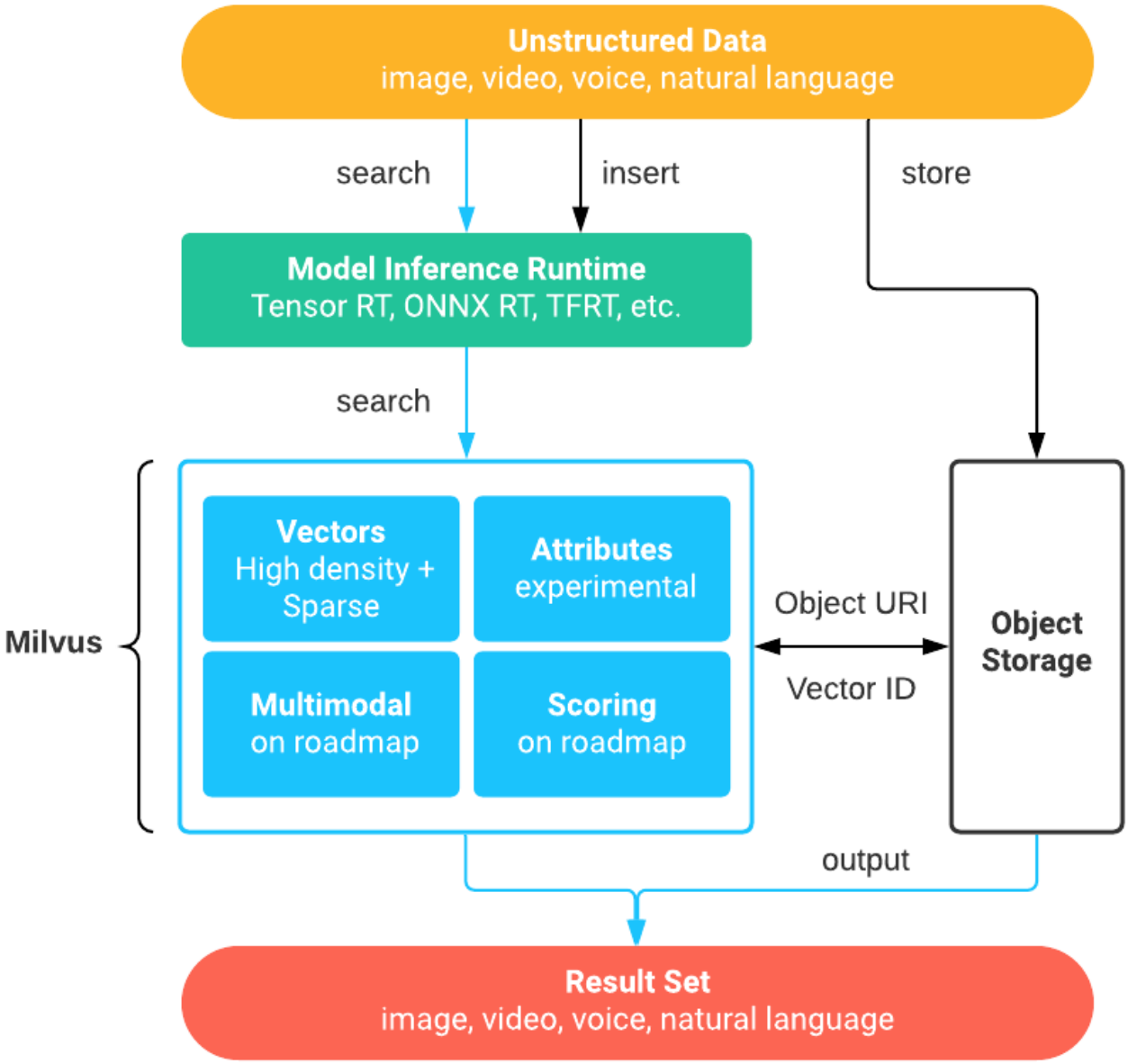

Figure 1: Milvus 架构

发行版本

Milvus 提供两个发行版本：CPU 版本和 GPU 版本。

CPU 版 Milvus 仅支持使用 CPU 建索引和搜索。

GPU 版 Milvus 在 CPU 版的基础上进行了 GPU 加速：支持同时进行索引创建和搜索计算以提高查询效率。你可以在同一时间内使用 GPU 建索引，使用 CPU 搜索向量。

如果你的计算机上安装了支持 CUDA 功能的 GPU 设备，你可以安装 Milvus 的 GPU 版本以获取针对海量数据的更优的查询性能。

详见：Milvus 发行版本。

加入开发者社区

如果你想为 Milvus 项目贡献代码，请参考我们的代码贡献指南：Contribute to Milvus。

如果你对 Milvus 有任何与功能、SDK 等相关的问题，欢迎加入 Slack 参与讨论。## 发行版本

Milvus 版本比较

概述

Milvus 提供两个发行版本：CPU 版本和 GPU 版本。

CPU 版 Milvus 仅支持使用 CPU 建索引和搜索。

GPU 版 Milvus 在 CPU 版的基础上进行了 GPU 加速：支持同时进行索引创建和搜索计算以提高查询效率。你可以在同一时间内使用 GPU 建索引，使用 CPU 搜索向量。

如果你的计算机上安装了支持 CUDA 功能的 GPU 设备，你可以安装 Milvus 的 GPU 版本以获取针对海量数据的更优的查询性能。

CPU 版 MilvusGPU 版 Milvus

CPU 版 Milvus 支持的索引类型

Milvus 会根据不同的向量类型提供不同的索引支持。请根据你的向量类型查看适用于你的索引：

浮点型向量 二值型向量

索引类型	CPU 建索引	GPU 建索引	CPU 搜索	GPU 搜索
FLAT	N/A	N/A	✔	×
IVF_FLAT	✔	×	✔	×
IVF_SQ8	✔	×	✔	×
IVF_PQ	✔	×	✔	×
RNSG	✔	×	✔	×
HNSW	✔	×	✔	×
Annoy	✔	×	✔	×

索引类型	CPU 建索引	GPU 建索引	CPU 搜索	GPU 搜索
FLAT	N/A	N/A	✔	×
IVF_FLAT	✔	×	✔	×

v1.0 发布公告

Milvus v1.0 已发布

今天，我们很高兴地宣布 Milvus v1.0 版本的发布。通过数百名 Milvus 社区用户在八个月内不断的测试和试验，Milvus v0.10.x 现在已足够稳定，是时候该发布基于 Milvus v0.10.6 的 Milvus v1.0 了。

Milvus v1.0 具有以下功能:

- 支持主流的相似度计算方式，包括欧氏距离 (L2)、内积 (IP)、汉明距离、杰卡德距离等;
- 可与 SOTA ANN 算法集成和优化，包括 Faiss，Hnswlib，Annoy，NSG 等;
- 可通过 Mishards 集群分片中间件进行横向扩展;
- 支持 AI 场景中常用的处理器，包括 X86，Nvidia GPU，Xilinx FPGA 等。

  请参见发版说明，了解更多有关 v1.0 的功能。

Milvus 是一个正在持续发展的开源软件（OSS）项目。虽然该项目后续会不断迭代，我们仍认为第一个主要发行版本对我们的社区用户至关重要，原因如下:

- 社区将长期将支持 Milvus v1.0;
- 一个稳定版的 Milvus 可以更轻松地集成到 AI 生态系统中;
- Milvus 项目将进一步向新的方向演化。

长期支持（LTS）

Milvus v1.0 是我们的第一个长期支持（LTS）版本。Milvus 社区将在 2024 年 12 月 31 日之前为 Milvus v1.0 提供问题修复支持（由 Zilliz 公司赞助）。v1.x 版本中将不会添加新功能。

更多有关 Milvus 版本发布策略的详细说明，请参见《Milvus 发行指南》。

工具链与 AI 生态系统整合

Milvus 引擎的开发正在快速迭代，但我们尚未在 Milvus 工具链的开发上投入大量资源。从 v1.0 开始，我们计划为 Milvus 用户开发必要的工具和实用程序。请在工具链兴趣小组中找到更多详细信息。

稳定的版本使 Milvus 与 AI 生态系统的集成变得轻而易举。现在，我们正在寻求 Milvus 社区与其他 AI OSS 社区之间的更多合作。我们也鼓励在 Milvus 中支持新的 AI 芯片（ASIC）。如果您有兴趣，请查看 ASIC 兴趣小组。

Milvus 的未来

我们认为，Milvus 光明的发展前景取决于以下因素:

- Milvus 社区开发人员的积极贡献;
- 能够与任何云原生环境集成。

为了不断培育和发展 Milvus 社区，我们制定了社区宪章，据此做出了一些技术决策，以吸引更多的参与者加入社区。

- 我们将使用 Golang 进行 Milvus 引擎的开发，而 ANNS 算法组件仍将使用 C++ 开发;
- 展望未来，分布式/集群/云 Milvus 将尽可能使用现有的云组件。

让我们共同努力，为 AI 构建下一代云数据管理解决方案!

Milvus 数据迁移指南

本指南适用于从 Milvus v0.7.0~0.10.6 至 Milvus v1.0.0 的数据迁移.

Milvus v0.11.0 以及早于 v0.7.0 的版本与 Milvus v1.0.0 不兼容。

[MilvusDM 可支持从 Milvus v0.10.x 至 v1.0.0 的数据迁移。

第 1 步：关闭当前版本 Milvus

停止当前版本的 Milvus:

```
docker stop [Your_milvus_container_id]
```

删除 /milvus 下的 /conf、/logs 和 /wal 文件夹:

```
cd ~/milvus
sudo rm -rf ./conf
sudo rm -rf ./logs
sudo rm -rf ./wal
```

如需保留日志文件，请将 logs 文件夹备份到其他目录。

第 2 步：下载配置文件

新建 conf 目录并下载 v1.0.0 的配置文件：

```
mkdir conf
cd conf
wget https://raw.githubusercontent.com/milvus-io/milvus/v1.1.1/core/conf/demo/server_config.yaml
```

如果下载不成功，可在网页中打开下载网址，在 conf 目录下新建同名文件，把网页中的内容粘贴到文件中保存。

第 3 步：确认、更新 MySQL/SQLite 服务端地址

```
vim ./server_config.yaml
```

确保新版本配置项 `general.meta_uri` 指定的 MySQL/SQLite 服务端地址与老版本配置项 `db_config.backend_url` 指定的服务端地址一致。以 MySQL 为例，更新的配置信息如下所示：

```
general:
timezone:UTC+8
meta_uri: mysql://root:123456@<MySQL_server_host IP>:3306/milvus
```

第 4 步：下载并启动新版 Milvus

使用相同的映射路径下载并启动 Milvus v1.0.0 容器：

```
$ sudo docker run -d --name milvus_cpu_1.1.1 \
-p 19530:19530 \
-p 19121:19121 \
-v /home/$USER/milvus/db:/var/lib/milvus/db \
-v /home/$USER/milvus/conf:/var/lib/milvus/conf \
-v /home/$USER/milvus/logs:/var/lib/milvus/logs \
-v /home/$USER/milvus/wal:/var/lib/milvus/wal \
milvusdb/milvus:1.1.1-cpu-d061621-330cc6
```

第 5 步：安装对应版本的 Python SDK

```
pip3 install pymilvus==1.1.2
```

第 6 步：验证数据正确性

编写 Python 脚本调用相关接口验证数据正确性。# 基本概念
特征向量

什么是特征向量

向量是具有一定大小和方向的量，可以简单理解为一串数字的集合，就像一行多列的矩阵，比如：[2,0,1,9,0,6,3,0]。每一行代表一个数据项，每一列代表一个该数据项的各个属性。

特征向量是包含事物重要特征的向量。大家比较熟知的一个特征向量是 RGB (红-绿-蓝) 色彩。每种颜色都可以通过对红 (R)、绿 (G)、蓝 (B) 三种颜色的比例来得到。这样一个特征向量可以描述为：颜色 = [红，绿，蓝]。

特征向量的优势

随着现代计算机和机器学习技术的飞速发展，越来越多的多媒体数据被存储、分析和运用在房地产、制药以及金融信息服务等领域。多媒体数据通常有多维度的特征。

而特征向量则可以通过多维度数值高效准确地描述多媒体内容。

特征向量在机器学习和模式识别的多个领域都拥有重要地位。机器学习算法通常需要用数值来表示描述对象以便于进行统计分析。

应用领域

特征向量，因其能通过数值的方式高效准确地描述对象，目前被广泛运用于机器学习的各种领域。

- 图像识别

  向量特征可以是梯度幅值、颜色、灰度强度、边缘、区域等。由于图像的数值化表示更容易定义和分析，特征向量在图像识别领域有着广泛的应用。

- 语音识别

  向量特征可以是音长、噪音等级、信噪比等。

- 垃圾邮件过滤

  向量特征可以是 IP 地址、文字结构、某个词出现的频率、特定邮件标题等。

向量索引

向量索引（vector index）是指通过某种数学模型，对向量构建的一种时间和空间上更高效的数据结构。借助向量索引，我们能够高效地查询与目标向量相似的若干个向量。Milvus 的实体目前只支持一个向量字段。你可以为该向量字段指定一种索引类型以提高查询性能。Milvus 会在切换索引类型时自动删除旧索引。

Milvus 目前支持的向量索引类型大都属于 ANNS（Approximate Nearest Neighbors Search，近似最近邻搜索）。ANNS 的核心思想是不再局限于只返回最精确的结果项，而是仅搜索可能是近邻的数据项，即以牺牲可接受范围内的精度的方式提高检索效率。

根据实现方式，ANNS 向量索引可分为四大类：

- 基于树的索引
- 基于图的索引
- 基于哈希的索引
- 基于量化的索引

下表将目前 Milvus 支持的索引进行了归类：

Milvus 支持的索引

索引分类

适用场景

FLAT

N/A

查询数据规模小，对查询速度要求不高。

需要 100% 的召回率。

IVF_FLAT

基于量化的索引

高速查询，

要求尽可能高的召回率。

IVF_SQ8

基于量化的索引

高速查询，

磁盘和内存资源有限，

仅有 CPU 资源。

IVF_SQ8H

基于量化的索引

高速查询，

磁盘、内存、显存有限。

IVF_PQ

基于量化的索引

RNSG

基于图的索引

HNSW

基于图的索引

Annoy

基于树的索引

索引创建机制

数据段建索引

Milvus 数据段存储海量数据。在建立索引时，Milvus 为每个数据段单独创建索引。

用户主动创建索引

调用 `create_index()` 接口时，Milvus 会对该字段上的已有数据同步建立索引。每当后续插入的数据的大小达到系统配置的 `index_file_size` 时，Milvus 会为其在后台自动创建索引。

当插入的数据段少于 `4096` 行时，`Milvus` 不会为其建立索引。

闲时建索引

众所周知，建索引是一个比较消耗计算资源和时间的工作。当查询任务和后台建索引任务并发时，Milvus 通常把计算资源优先分配给查询任务，即用户发起的任何查询命令都会打断后台正在执行的建索引任务。之后仅当用户持续 5 秒不再发起查询任务，Milvus 才会恢复执行后台建索引任务。此外，如果查询命令指定的数据段尚未建成指定索引，Milvus 会直接在段内做全量搜索。

索引概览

FLAT

FLAT 索引类型是指对向量进行原始文件存储。搜索时，所有向量都会与目标向量进行距离计算和比较。

FLAT 索引类型提供 100% 的检索召回率。与其他索引相比，当查询数量较少时，它是最有效的索引方法。

- 查询参数

参数	说明	取值范围
`metric_type`	[可选] 距离计算方式	详见目前支持的距离计算方式。

IVF_FLAT

IVF（Inverted File，倒排文件）是一种基于量化的索引类型。它通过聚类方法把空间里的点划分成 `nlist` 个单元。查询时先把目标向量与所有单元的中心做距离比较，选出 `nprobe` 个最近单元。然后比较这些被选中单元里的所有向量，得到最终的结果。

IVF_FLAT 是最基础的 IVF 索引，存储在各个单元中的数据编码与原始数据一致。

- 建索引参数

参数	说明	取值范围
`nlist`	聚类单元数	[1, 65536]

- 查询参数

参数	说明	取值范围
`nprobe`	查询取的单元数	[1, 65536] GPU 版 Milvus 在 `nprobe` > 2048 时由 GPU 查询切换为 CPU 查询。

IVF_SQ8

IVF_SQ8 是在 IVF 的基础上对放入单元里的每条向量做一次标量量化（Scalar Quantization）。标量量化会把原始向量的每个维度从 4 个字节的浮点数转为 1 个字节的无符号整数，因此 IVF_SQ8 索引文件占用的存储空间远小于 IVF_FLAT。但是，标量量化会导致查询时的精度损失。

- 建索引参数同 IVF_FLAT
- 查询参数同 IVF_FLAT

IVF_SQ8H

IVF_SQ8H 是一种优化查询执行的 IVF_SQ8 索引类型。

在不同的 `nq`（Number of queries，查询数量）与系统参数 `gpu_search_threshold` 的关系下，查询方式如下：

- `nq` ≥ `gpu_search_threshold`：整个查询过程都在 GPU 上执行。
- `nq` < `gpu_search_threshold`：在 GPU 上执行在 IVF 里寻找 `nprobe` 个最近单元的运算，在 CPU 上执行其它运算。
- 建索引参数同 IVF_FLAT
- 查询参数同 IVF_FLAT

IVF_PQ

`PQ`（Product Quantization，乘积量化）会将原来的高维向量空间均匀分解成 `m` 个低维向量空间的笛卡尔积，然后对分解得到的低维向量空间分别做矢量量化。最终每条向量会存储在 `m` × `nbits` 个 bit 位里。乘积量化能将全样本的距离计算转化为到各低维空间聚类中心的距离计算，从而大大降低算法的时间复杂度。

IVF_PQ 是先对向量做乘积量化，然后进行 IVF 索引聚类。其索引文件甚至可以比 IVF_SQ8 更小，不过同样地也会导致查询时的精度损失。

不同版本的建索引参数和查询参数设置不同，请根据使用的 Milvus 版本查看相应的参数信息。

CPU 版 Milvus GPU 版 Milvus

- 建索引参数

参数	说明	取值范围
`nlist`	聚类单元数	[1, 65536]

参数	说明	取值范围
m	乘积量化因子个数	dim ≡ 0 (mod m)
nbits	分解后每个低维向量的存储位数 (可选)	[1, 16] (默认 8)

- 查询参数

参数	说明	取值范围
nprobe	查询取的单元数	[1, nlist]

- 建索引参数

参数	说明	取值范围
nlist	聚类单元数	[1, 65536]
m	乘积量化因子个数	m ∈ {1, 2, 3, 4, 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, 64, 96}, and (dim / m) ∈ {1, 2, 3, 4, 6, 8, 10, 12, 16, 20, 24, 28, 32}。m x 1024 的值不能超过显卡的 MaxSharedMemPerBlock。
nbits	分解后每个低维向量的存储位数 (可选)	8

如果 m 值不在指定区间，但是只要 m 值是 CPU 版 Milvus 支持的（点击上面按钮可查看 CPU 版 Milvus 支持的取值范围），Milvus 会自动由 GPU 建索引切换为 CPU 建索引。

如果 nbits 指定值不是 8 但是在 1 和 16 之间，系统会自动切换为 CPU 版 Milvus。

- 查询参数

参数	说明	取值范围
nprobe	查询取的单元数	[1, min(2048, nlist)]

如果 nprobe 值不在指定区间，但是只要 nprobe 值是 CPU 版 Milvus 支持的（点击上面按钮可查看 CPU 版 Milvus 支持的取值范围），Milvus 会自动由 GPU 查询切换为 CPU 查询。

RNSG

RNSG（Refined Navigating Spreading-out Graph）是一种基于图的索引算法。它把全图中心位置设为导航点，然后通过特定的选边策略来控制每个点的出度（小于等于 out_degree），使得搜索时既能减少内存使用，又能快速定位到目标位置附近。

RNSG 的建图流程如下：

1. 为每个点精确寻找 knng 个最近邻结点。
2. 以 knng 个最近邻结点为基础迭代至少 search_length 次，以选出 candidate_pool_size 个可能的最邻近结点。
3. 在选出的 candidate_pool_size 个结点里按择边策略构建每个点的出边。

RNSG 的查询流程与建图流程类似，以导航点为起点至少迭代 search_length 次以得到最终结果。

- 建索引参数

参数	说明	取值范围
out_degree	结点的最大出度	[5, 300]
candidate_pool_size	结点出边候选池	[50, 1000]
search_length	查询迭代次数	[10, 300]

参数	说明	取值范围
knng	预计算最近邻结点数	[5, 300]

- 查询参数

参数	说明	取值范围
search_length	查询迭代次数	[10, 300]

HNSW

HNSW（Hierarchical Small World Graph）是一种基于图的索引算法。它会为一张图按规则建成多层导航图，并让越上层的图越稀疏，结点间的距离越远；越下层的图越稠密，结点间的距离越近。搜索时从最上层开始，找到本层距离目标最近的结点后进入下一层再查找。如此迭代，快速逼近目标位置。

为了提高性能，HNSW 限定了每层图上结点的最大度数 M 。此外，建索引时可以用 efConstruction，查询时可以用 ef 来指定搜索范围。

- 建索引参数

参数	说明	取值范围
M	结点的最大度数	[4, 64]
efConstruction	搜索范围	[8, 512]

- 查询参数

参数	说明	取值范围
ef	搜索范围	[top_k, 32768]

Annoy

Annoy（Approximate Nearest Neighbors Oh Yeah）是一种用超平面把高维空间分割成多个子空间，并把这些子空间以树型结构存储的索引方式。

在查询时，Annoy 会顺着树结构找到距离目标向量较近的一些子空间，然后比较这些子空间里的所有向量（要求比较的向量数不少于 search_k 个）以获得最终结果。显然，当目标向量靠近某个子空间的边缘时，有时需要大大增加搜索的子空间数以获得高召回率。因此，Annoy 会使用 n_trees 次不同的方法来划分全空间，并同时搜索所有划分方法以减少目标向量总是处于子空间边缘的概率。

- 建索引参数

参数	说明	取值范围
n_trees	空间划分的方法数	[1, 1024]

- 查询参数

参数	说明	取值范围
search_k	搜索的结点数。-1 表示用全数据量的 5%	{-1} ∪ [top_k, n × n_trees]

选择索引

- 若要为你的使用场景选择合适的索引，请参阅 如何选择索引类型。
- 关于索引和向量距离计算方法的选择，请访问 距离计算方式。

常见问题

索引 IVF_SQ8 和 IVF_SQ8H 在召回率上有区别吗?

对于相同的数据集，IVF_SQ8 和 IVF_SQ8H 的召回率一致。

Milvus 中 FLAT 索引和 IVF_FLAT 索引的原理比较?

把 FLAT 和 IVF_FLAT 做比较，可以这么估算：

已知 IVF_FLAT 索引是把向量分成 nlist 个单元。假设用默认的 nlist = 16384，搜索的时候是先用目标向量和这 16384 个中心点计算距离，得到最近的 nprobe 个单元，再在单元里计算最近向量。而 FLAT 是每条向量和目标向量计算距离。

所以当总的向量条数约等于 nlist 时，两者的计算量相当，性能也差不多。而随着向量条数达到 nlist 的 2 倍、3 倍、n 倍之后，IVF_FLAT 的优势就越来越大。

可参阅 如何选择索引类型。

参考文献

- RNSG：Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph
- HNSW：Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs
- Annoy：Nearest neighbors and vector models　part 2　algorithms and data structures

存储相关

存储相关概念

分区和数据段

建立集合时，Milvus 根据参数 `index_file_size` 控制数据段的大小。另外，Milvus 提供分区功能，你可以根据需要将数据划分为多个分区。对数据的合理组织和划分可以有效提高查询性能。

数据段（segment）

为了能处理海量的数据，Milvus 会将数据分段，每段数据拥有数万甚至数十万个实体。每个数据段的数据又按照字段（field）分开，每个字段的数据单独存为一个数据文件。目前的版本中，实体仅包含一个 ID 字段和一个向量字段，因此每个数据段的数据文件主要包括一个 UID 文件以及一个原始向量数据文件。

数据段的大小是由创建集合时的参数 `index_file_size` 来决定的，默认为 1024 MB，上限为 128 GB。

建立索引时，集合中的每个数据段依次建立索引，并将索引单独存为一个文件。索引文件之间相互独立。索引可以显著地提高检索性能。

分区（partition）

当一个集合累积了大量数据之后，查询性能会逐渐下降。而某些场景只需查询集合中的部分数据，这时就要考虑把集合中的数据根据一定规则在物理存储上分成多个部分。这种对集合数据的划分就叫分区。每个分区可包含多个数据段。

分区以标签（tag）作为标识。插入向量数据时，你可以指定将数据插入到某个标签对应的分区中。查询向量数据时，你可以根据标签来指定在某个分区的数据中进行查询。Milvus 既支持对分区标签的精确匹配，也支持正则表达式匹配。

每个集合的分区数量上限是 4096 个。

集合、分区和数据段的关系

集合、分区以及数据段的关系如下所示：

每个集合都有一个 _default 分区。插入数据时如果没有指定分区，Milvus 会将数据插入该分区中。

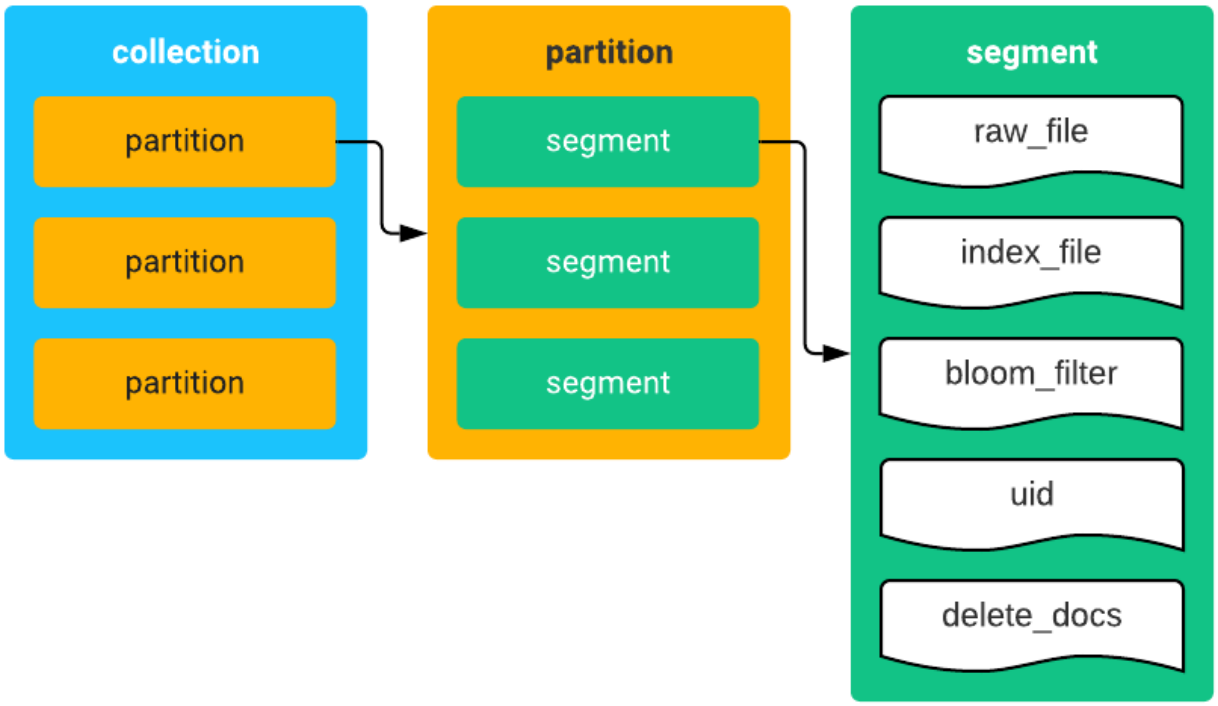

Figure 2: file

元数据

不管是分区还是段，都只是数据在物理存储中的组织形式。Milvus 进行查询操作时，必须要获知各个数据文件在物理存储上的位置以及状态信息，包括所属集合、包含的实体条数、文件的大小、全局唯一的标识、以及创建日期等等。我们将这些信息称为元数据。此外，元数据还包含集合以及分区的信息，包括集合名称、集合维度、索引类型、分区标签等等。

当数据发生改变时，元数据应相应变化并且易于获取，因此使用事务型数据库来管理元数据是一个理想的选择。Milvus 提供 SQLite 或者 MySQL 作为元数据的后端服务。对于生产环境或者分布式服务来说，应当使用 MySQL 来作为元数据后端服务。

元数据后端服务不负责存储实体数据和索引。

常见问题

Milvus 的元数据存储可以使用 SQL Server 或者 PostgreSQL 吗?

不可以，目前仅支持 SQLite 和 MySQL。

存储操作

阅读本文前，请先阅读 存储相关概念。

数据插入

客户端通过调用 `insert` 接口来插入数据，单次插入的数据量不能大于 256 MB。插入数据的流程如下：

1. 服务端接收到插入请求后，将数据写入预写日志（WAL）。
2. 当预写日志成功记录后，返回插入操作。
3. 将数据写入可写缓冲区（mutable buffer）。

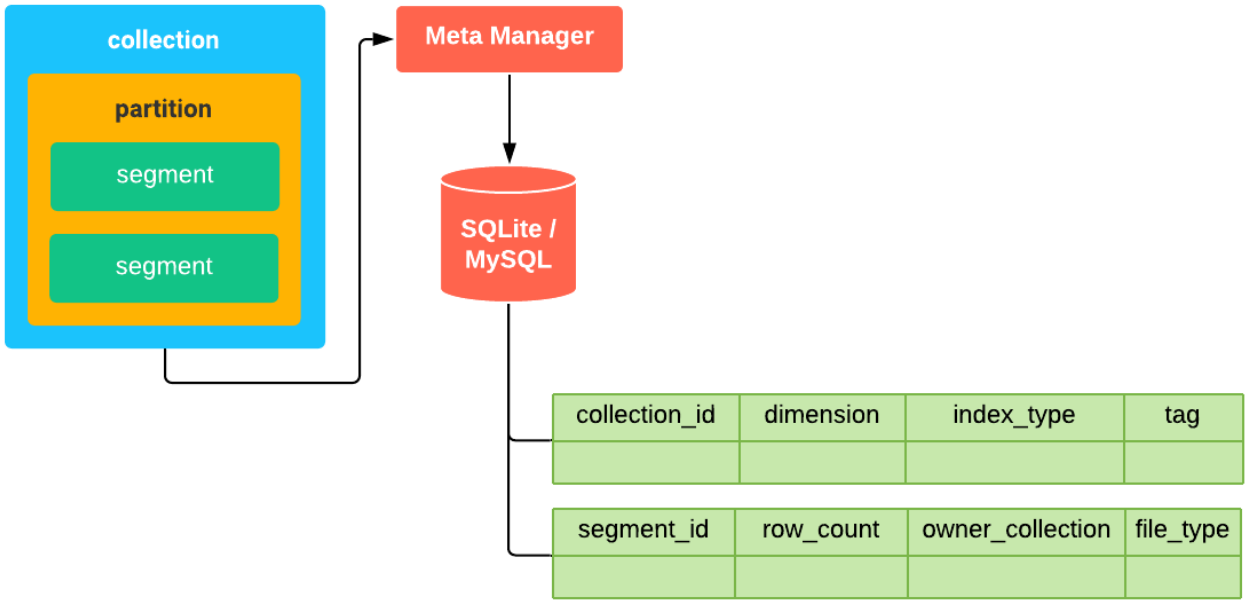

Figure 3: meta

每个集合都有独立的可写缓冲区。每个可写缓冲区的容量上限是 128 MB。所有集合的可写缓冲区总容量上限由系统参数 insert_buffer_size 决定，默认是 1 GB。

数据落盘

缓冲区中的数据落盘有三种触发机制：

定时触发

系统会定时触发落盘任务。定时间隔由系统参数 `auto_flush_interval` 决定，默认是 1 秒。

落盘操作的流程如下：

1. 系统开辟一块新的可写缓冲区，用于容纳后续插入的数据。
2. 系统将之前的可写缓冲区设为只读（immutable buffer）。
3. 系统把只读缓冲区的数据写入磁盘，并将新数据段的描述信息写入元数据后端服务。

完成以上流程后，系统就成功创建了一个数据段（segment）。

客户端触发

由客户端调用 `flush` 接口触发落盘。

缓冲区达到上限触发

累积数据达到可写缓冲区的上限（128MB）会触发落盘操作。

每个数据段的所有相关文件都被存放在以段 ID 命名的文件夹中，比如记录实体 ID 的 UID 文件、用于标记已被删除实体的 delete_docs 文件，以及用于快速查找实体的布隆过滤器（bloom-filter）文件。

段内数据文件请参考 分区和数据段 中的示意图。

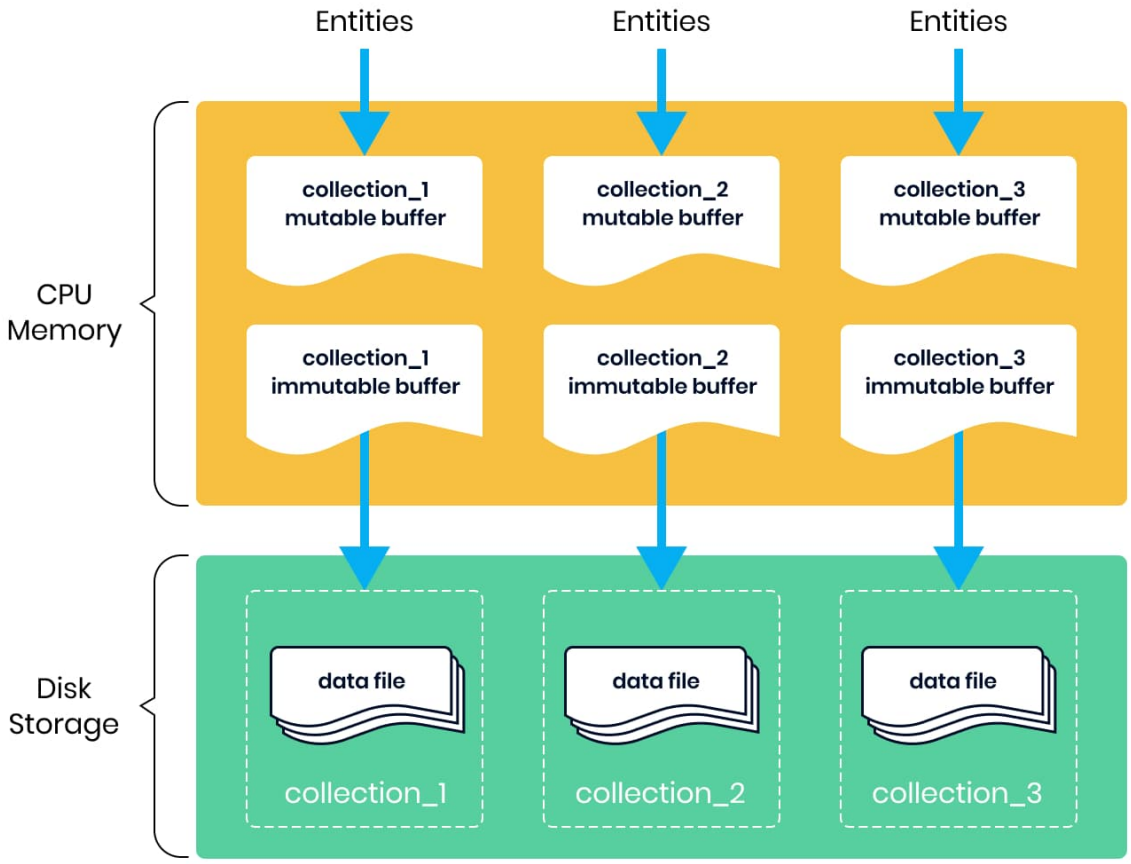

Figure 4: insert

数据合并

小数据段过多会导致查询性能低下。为了避免此问题，Milvus 会在需要的时候触发后台段合并任务，即把小数据段合并成新的数据段，并删除小数据段、更新元数据。其中，新数据段的大小不低于 `index_file_size`。

合并操作的触发时机如下：

- 启动服务时
- 完成落盘任务后
- 建索引前
- 删除索引后

已经建立了索引的数据段不会参与合并操作。

建立索引

未建立索引之前，Milvus 对集合的查询操作都是以暴力搜索（brute-force search）的方式完成的。为提高查询性能，你可以为集合建立合适的索引。索引建成后，每个数据段都会产生一个索引文件，此时元数据也会同步更新。

更多索引相关信息，请参考 向量索引。

删除

删除集合

1. 客户端调用 `drop_collection` 接口来删除一个集合。
2. 服务端接收到请求后，仅在元数据中把该集合（包括它的分区和段）标记为删除状态。对于已标记为删除状态的集合，将无法再对其进行任何新操作（比如插入和查询）。
3. 后台的清理任务将被标记为删除状态的集合（包括它的分区和段）从元数据中删除，然后将该集合的数据文件和文件夹从磁盘上删除。如果在删除操作之前已经有对该集合的操作正在执行，后台清理任务不会删除正在使用的段，直到操作完成。

删除分区

1. 客户端调用 `drop_partition` 接口来删除一个分区。
2. 服务端接收到请求后，仅在元数据中把该分区（包括它的段）标记为删除状态。
3. 后台清理任务按照删除集合的流程来删除该分区和元数据。

删除实体

Milvus 为每个数据段建立了一个 delete_docs 文件，用来记录被删除向量在段内的位置。

Milvus 使用布隆过滤器（bloom filter）来快速判断一个实体 ID 是否可能存在于某个数据段中。因此，在每个数据段下都创建了一个名为 bloom_filter 的文件。

删除实体的流程如下：

1. 客户端调用 `delete_entity_by_id` 接口删除集合中的实体。
2. 服务端接收到请求后，执行以下操作删除实体：
    - 如果该实体在插入缓冲区中，直接删除该实体。
    - 否则，根据每个数据段的布隆过滤器判断该实体所处的数据段，然后更新该数据段的 delete_docs 以及 bloom_filter 文件。

数据段整理

查询一个数据段时，Milvus 会将该数据段的实体数据以及 delete_docs 文件读入内存。虽然被删除的实体不参与计算，但它们也会被读入内存。所以，一个数据段中被删除的实体越多，浪费的内存资源和磁盘空间越多。为了减少此类不必要的资源消耗，Milvus 提供了数据段整理（compact）的操作，流程如下：

1. 客户端调用 `compact` 接口。
2. 服务端接收到请求后，根据 delete_docs 所记录的信息，将段内未被删除的实体写入一个新的数据段，并把旧数据段标记为删除状态。之后将由后台清理任务负责清理被标记为删除状态的数据段。如果旧数据段已建立索引，新数据段产生之后会重建索引。

compact 操作会忽略被删除向量占比小于 10% 的数据段。

数据读取

1. 客户端调用 `get_entity_by_id` 接口读取原始实体数据。
2. 服务端接收到请求后，通过布隆过滤器找到实体所在的段，返回该实体 ID 对应的数据。

浮点型向量在 Milvus 中以单精度（float）方式存储。

常见问题

Milvus 可以通过扩展某些接口（如 S3 接口、GlusterFS 接口）来扩展存储吗?

目前暂不支持。

Milvus 有没有数据导出的功能?

目前没有专门的工具实现该功能。你可以通过 get_entity_by_id 得到指定 ID 对应的向量。

相似度计算方式

Milvus 基于不同的距离计算方式比较向量间的距离。选择合适的距离计算方式能极大地提高数据分类和聚类性能。

以下表格列出了 Milvus 目前支持的距离计算方式与数据格式、索引类型之间的兼容关系。

浮点型向量 二值型向量

距离计算方式

索引类型

欧氏距离 (L2)

FLAT

IVF_FLAT

IVF_SQ8

IVF_SQ8H

IVF_PQ

RNSG

HNSW

Annoy

内积 (IP)

距离计算方式

索引类型

杰卡德距离 (Jaccard) 谷本距离 (Tanimoto) 汉明距离 (Hamming)

FLAT

IVF_FLAT

超结构 (superstructure) 子结构 (substructure)

FLAT

$$d(\boldsymbol{a},\boldsymbol{b}) = d(\boldsymbol{b},\boldsymbol{a}) = \sqrt{\sum_{i=1}^{n}(b_i - a_i)^2}$$

Figure 5: euclidean

欧氏距离（L2）

欧氏距离计算的是两点之间最短的直线距离。

欧氏距离的计算公式为：

其中 a = (a1, a2,⋯, an) 和 b = (b1, b2,⋯, bn) 是 n 维欧氏空间中的两个点。

欧氏距离是最常用的距离计算方式之一，应用广泛，适合数据完整，数据量纲统一的场景。

内积（IP）

两条向量内积距离的计算公式为：

$$p(A,B) = A \cdot B = \sum_{i=1}^{n} a_i \times b_i$$

Figure 6: ip

假设有 A 和 B 两条向量，则 `||A||` 与 `||B||` 分别代表 A 和 B 归一化后的值。

内积更适合计算向量的方向而不是大小。

如需使用点积计算向量相似度，则必须对向量作归一化处理。处理后点积与余弦相似度等价。

假设 X’ 是向量 X 的归一化向量：

$$X' = (x'_1, x'_2, \ldots, x'_n), X' \in^n$$

Figure 7: normalize

两者之间的关系为：

杰卡德距离

杰卡德相似系数计算数据集之间的相似度，计算方式为：数据集交集的个数和并集个数的比值。计算公式可以表示为：

$$x'_i = \frac{x_i}{\| X \|} = \frac{x_i}{\sqrt{\sum_{i=1}^{n} (x_i)^2}}$$

Figure 8: normalization

$$J(A,B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Figure 9: Jaccard similarity coefficient

杰卡德距离是用来衡量两个数据集差异性的一种指标，被定义为 1 减去杰卡德相似系数。对于二值变量，杰卡德距离等价于谷本系数。

$$d_j(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

Figure 10: Jaccard distance

杰卡德距离适合字符串相似性度量。

谷本距离

对于二值变量，谷本距离公式可表示为:

在 Milvus 中，谷本距离仅支持二值变量。

值域从 0 到正无穷。

对于二值变量，谷本系数等价于杰卡德距离:

对于二值变量，谷本系数值域为 0 到 +1（+1 的相似度最高）

汉明距离

汉明距离计算二进制字符串之间的距离。两个等长字符串之间的汉明距离定义为将其中一个变为另外一个所需要作的最小替换次数。

比如，假设有两条字符串 1101 1001 和 1001 1101。比较时，如果字符相同用 0 表示，如果字符不同则用 1 表示。

11011001 ⊕ 10011101 = 01000100

所以以上两条字符串之间的汉明距离为 2。

$$d_t = \frac{A \cdot B}{|A|^2 + |B|^2 - A \cdot B}$$

Figure 11: tanimoto distance

$$T(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

Figure 12: tanimoto coefficient

超结构

超结构主要用来计算某化学结构与其超结构的相似度。值越小则相似度越大。Milvus 目前只返回距离为 0 的结果。

超结构的公式可表示为：

$$1 - \frac{N_{A\&B}}{N_A}$$

Figure 13: superstructure

其中

- 分子式 B 是分子式 A 的超结构。
- NA 表示分子式 A 的化学指纹中二进制位的数量。
- NB 表示分子式 B 的化学指纹中二进制位的数量。
- NAB 表示分子式 A 和 B 的化学指纹中共有的二进制位的数量。

子结构

子结构主要用来计算某化学结构与其子结构的相似度。值越小则相似度越大。Milvus 目前只返回距离为 0 的结果。

子结构的公式可表示为：

其中

- 分子式 B 是分子式 A 的子结构。
- NA 表示分子式 A 的化学指纹中二进制位的数量。
- NB 表示分子式 B 的化学指纹中二进制位的数量。
- NAB 表示分子式 A 和 B 的化学指纹中共有的二进制位的数量。

常见问题

为什么向量距离计算方式是内积时，搜索出来的 top1 不是目标向量本身？

向量距离计算方式用内积时，如果向量未归一化，会出现这样的情况。

什么是归一化？Milvus 中为什么有时候需要归一化？

$$1 - \frac{N_{A\&B}}{N_B}$$

Figure 14: substructure

归一化指的是通过数学变换将向量的模长变为 1 的过程。如需使用点积计算向量相似度，则必须对向量作归一化处理。处理后点积与余弦相似度等价。

可参阅文章 向量搜索的简明数学基础。

为什么欧氏距离和内积在计算向量相似度时的结果不一致?

如果欧氏距离和内积返回不一致的结果，需要检查数据是否已经归一化。如果没有，请先对数据进行归一化。理论上可以证明，对于未归一化的数据，欧氏距离和内积的结果是不一致的。

预写式日志

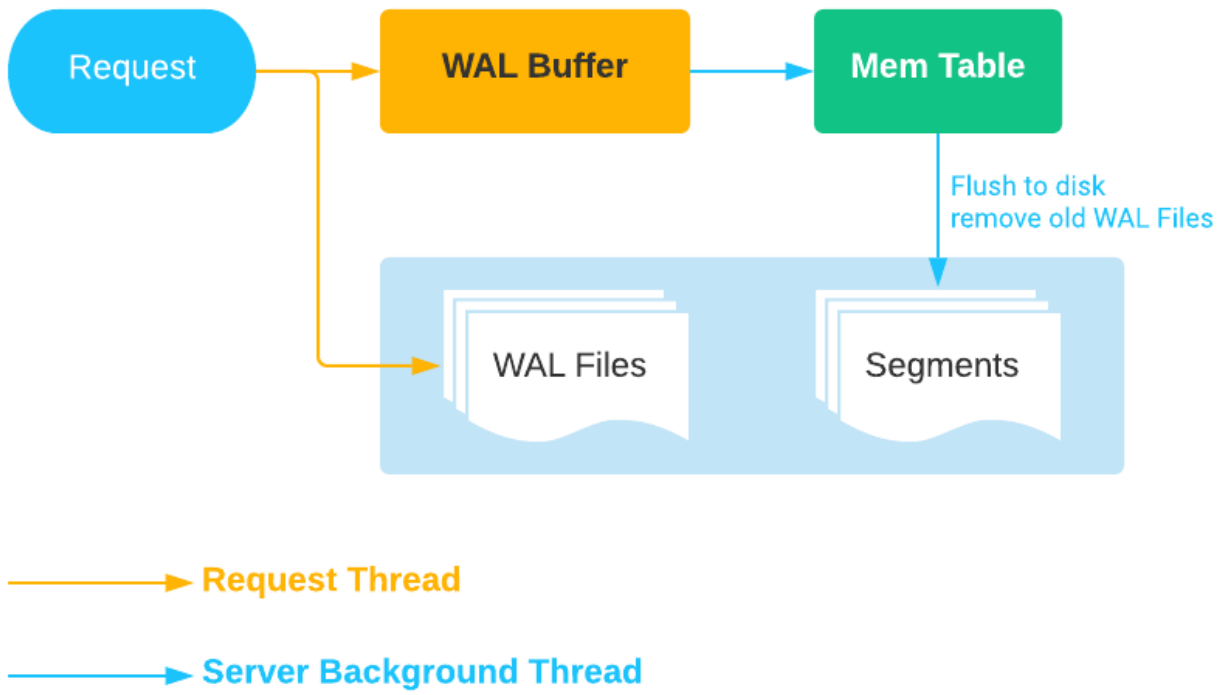

Figure 15: wal_structure

预写式日志首先把用户的插入和删除请求记入日志文件，然后由后台线程写入系统。一旦将用户请求成功写入日志，服务端即会返回成功。开启该功能可以增强数据的可靠性，并减少对客户端的阻塞。

数据可靠性

预写式日志能保证修改请求的原子性。所有返回成功的请求都会被完整地写入系统。对于因系统意外退出或者链接意外断开而没有响应的请求，操作只可能全部成功或者全部失败。操作是否成功可以通过调用其它接口来确认。此外，在系统重启时，日志中还未被应用到系统状态的请求将被重新执行。

缓冲区设置

预写式日志使用的缓冲区大小由系统参数 `wal.buffer_size` 决定。为保证预写式日志的写入性能，建议把缓冲区大小设为单批次导入数据量大小的 2 倍以上。

关于如何设置系统参数 `<code>wal.buffer_size</code>`，请见 `<a href="milvus_config.md">Milvus 配置</a>`。

旧日志删除

Milvus 会自动删除那些已经应用到系统的日志。

Milvus 术语

- Collection: 包含一组 entity，可以等价于关系型数据库系统（RDBMS）中的表。
- Segment: Milvus 在数据插入时通过合并数据自动创建的数据文件。一个 collection 可以包含多个 segment。一个 segment 可以包含多个 entity。在搜索中，Milvus 会搜索每个 segment，过滤被删除的数据，并返回合并后的结果。
- Entity: 包含一组 field。field 与实际对象相对应。field 可以是代表对象属性的结构化数据，也可以是代表对象特征的向量。
- Entity ID: 用于指代一个 entity 的唯一值。

  目前，Milvus 不支持 ID 去重，因此有可能在一个 segment 内出现重复 ID。
- Field: entity 的组成部分。field 可以是结构化数据，例如数字和字符串，也可以是向量。
- Vector: 一种类型的 field，代表对象的特征。

  目前，一个实体最多只能包含一个向量。
- Index: 索引基于原始数据构建，可以提高对 collection 数据搜索的速度。
- Mapping: 一系列规则用来定义 collection 如何组织数据。# 快速开始
  ## 安装概述

在开始前，请确保已经阅读发行版本以了解 Milvus 的两个发行版本的性能差异和适用场景。

我们推荐使用 Docker 启动 Milvus 服务。请根据你的 Milvus 版本选择相应的安装指南：

- 安装 CPU 版本的 Milvus
- 安装 GPU 版本的 Milvus

如果你没有安装 Docker 环境，你可以参考从源代码编译 Milvus。

安装、启动 Milvus 服务

CPU 版 MilvusGPU 版 Milvus

安装前提

操作系统

操作系统	版本
CentOS	7.5 或以上
Ubuntu LTS	18.04 或以上

硬件

硬件	建议配置
CPU	Intel CPU Sandy Bridge 或以上
CPU 指令集	
内存	8 GB 或以上（取决于具体向量数据规模）
硬盘	SATA 3.0 SSD 或以上

软件

软件	版本
Docker	19.03 或以上

请确保可用内存大于你在 server_config.yaml 文件中设置的 cache.insert_buffer_size 和 cache.cache_size 之和。

确认 Docker 状态

确认 Docker daemon 正在运行：

```
$ sudo docker info
```

如果无法正常打印 Docker 相关信息，请启动 Docker daemon。

在 Linux 上需要使用 sudo 执行 Docker 命令。若要在没有 sudo 的情况下运行 Docker 命令，请创建 docker 组并添加用户，详见 Linux 安装步骤。

拉取 Milvus 镜像

拉取 CPU 版本的 Milvus 镜像：

```
$ sudo docker pull milvusdb/milvus:1.1.1-cpu-d061621-330cc6
```

如果你的主机由于网络限制无法在线获得 Docker 镜像和配置文件，请从其他主机在线获取镜像，保存为 TAR 文件传输回本地，传输完成后重新加载为 Docker 镜像：

点击查看离线传输相关代码示例。

将 Docker 镜像保存为 TAR 文件再使用合适的方式传输。

$ docker save milvusdb/milvus > milvus_image.tar

将 TAR 文件传输完成后使用以下命令重新加载成 Docker 镜像。

$ docker load < milvus_image.tar

如果拉取镜像的速度过慢或一直失败，请参考 部署运维问题 中提供的解决办法。

下载配置文件

```
$ mkdir -p /home/$USER/milvus/conf
$ cd /home/$USER/milvus/conf
$ wget https://raw.githubusercontent.com/milvus-io/milvus/v1.1.1/core/conf/demo/server_config.yaml
```

如果无法通过 wget 命令下载配置文件，你也可以在 /home/$USER/milvus/conf 目录下创建 server_config.yaml 文件，然后将 server config 文件 的内容复制到你创建的配置文件中。

启动 Milvus Docker 容器

启动 Docker 容器，将本地的文件路径映射到容器中：

```
$ sudo docker run -d --name milvus_cpu_1.1.1 \
-p 19530:19530 \
-p 19121:19121 \
-v /home/$USER/milvus/db:/var/lib/milvus/db \
-v /home/$USER/milvus/conf:/var/lib/milvus/conf \
-v /home/$USER/milvus/logs:/var/lib/milvus/logs \
-v /home/$USER/milvus/wal:/var/lib/milvus/wal \
milvusdb/milvus:1.1.1-cpu-d061621-330cc6
```

上述命令中用到的参数定义如下：

- `-d`: 在后台运行容器。
- `--name`: 为容器指定一个名字。
- `-p`: 指定端口映射。
- `-v`: 将宿主机路径挂载至容器。

确认 Milvus 运行状态：

```
$ sudo docker ps
```

如果 Milvus 服务没有正常启动，执行以下命令查询错误日志：

```
$ sudo docker logs milvus_cpu_1.1.1
```

常见问题

可以在 Windows 上安装 Milvus 吗?

理论上只要能够支持 Docker 的操作系统都可以运行 Milvus。

为什么 Milvus 在启动时返回 Illegal instruction?

如果你的 CPU 不支持 SSE42、AVX、AVX2、AVX512 其中任何一个指令集，则 Milvus 无法正常启动。可以通过 cat /proc/cpuinfo 查看 CPU 支持的指令集。

Milvus 中如何实现数据迁移?

详见数据迁移。

注意：不同版本之间，数据可能会不兼容。目前数据格式兼容到 Milvus v0.7.0。

Milvus 只能使用 Docker 部署吗?

Milvus 还支持源码编译，该方法仅支持 Linux 系统。详见 从源代码编译 Milvus。

应如何设置 IVF 索引的 nlist 和 nprobe 参数?

IVF 索引的 nlist 值需要根据具体的使用情况去设置。一般来说，推荐值为 $4 \times \text{sqrt(n)}$，其中 n 为 segment 内的 entity 总量。

`nprobe` 的选取需要根据数据总量和实际场景在速度性能和准确率之间进行取舍。建议通过多次实验确定一个合理的值。

以下是使用公开测试数据集 sift50m 针对 `nlist` 和 `nprobe` 的一个测试。以索引类型 IVF_SQ8 为例，针对不同 `nlist`/`nprobe` 组合的搜索时间和召回率分别进行对比。

因 CPU 版 Milvus 和 GPU 版 Milvus 测试结果类似，此处仅展示基于 GPU 版 Milvus 测试的结果。

在本次测试中，`nlist` 和 `nprobe` 的值成比例增长，召回率随 `nlist`/`nprobe` 组合增长呈现上升的趋势。

在 `nlist` 为 4096 和 `nprobe` 为 128 时，速度性能最佳。

接下来你可以

- 如果你刚开始了解 Milvus：
  - 运行示例程序
  - 了解更多 Milvus 基础操作
  - 体验 Milvus 在线训练营
- 如果你已准备好在生产环境中部署 Milvus：
  - 创建 监控与报警系统 实时查看系统表现
  - 设置 Milvus 参数
- 如果你想使用针对大数据集搜索的 GPU 加速版 Milvus：
  - 安装支持 GPU 加速版 Milvus

Hello Milvus

在 Milvus 服务端成功启动之后，你可以运行一个 Python 示例程序。这个示例程序将创建一个向量数据表，向其中插入 10 条向量，然后运行向量相似度查询。

1. 请确保系统已经安装了 Python 3.6 和 pip。
2. 安装 Milvus Python SDK。

   ```
   ### Install Milvus Python SDK
   $ pip3 install pymilvus==1.1.2
   ```

   如果需要进一步了解 Milvus Python SDK，请参考 Milvus Python SDK Readme。
3. 下载 Python 示例代码。

   ```
   ### Download Python example
   $ wget https://raw.githubusercontent.com/milvus-io/pymilvus/v1.1.2/examples/example.py
   ```

   万一你遇到无法通过 wget 命令正常下载示例代码的情况，你也可以创建 example.py 文件并复制粘贴 示例代码 的内容。
4. 运行示例代码。

   ```
   ### Run Milvus Python example
   $ python3 example.py
   ```
5. 确认程序正确运行。

   ```
   Query result is correct.
   ```

   恭喜你！你已经成功完成了在 Milvus 上的第一次向量相似度查询。

接下来你可以

- 了解更多 Milvus 基础操作
- 体验 Milvus 在线训练营，发现更多解决方案。

连接服务端

该页面将向你展示如何从 Python 客户端连接 Milvus 服务端。

关于详细的 API 使用方法，请参考 Python API 文档。

建议你使用 资源评估工具 来估算数据所需的硬件资源。

1. 导入 pymilvus：

   ```
   ### Import pymilvus.
   >>> from milvus import Milvus, IndexType, MetricType, Status
   ```

2. 使用以下任意一种方法连接 Milvus 服务端:

```
### Connect to the Milvus server.
>>> milvus = Milvus(host='localhost', port='19530')
```

在上面的代码中，host 和 port 都使用了默认值。你可以将其更改为自己设定的 IP 地址和端口。

```
>>> milvus = Milvus(uri='tcp://localhost:19530')
```

常见问题

Milvus 的 Python SDK 有连接池吗?

Milvus v0.9.0 及更高版本对应的 Python SDK 有连接池。连接池的默认连接数量没有上限。

在 Windows 安装 pymilvus 报错，如何解决?

可以尝试在 Conda 环境下安装。

创建、删除集合

本页提供创建或删除集合的 Python 示例代码。

参考 示例程序 获取更详细的使用方式。

创建集合

1. 准备创建集合所需参数:

```
### Prepare collection parameters.
>>> param = {'collection_name':'test01', 'dimension':256, 'index_file_size':1024, 'metric_type':Metric
```

2. 创建集合名为 test01，维度为 256，自动创建索引的数据文件大小为 1024 MB，距离度量方式为欧氏距离（L2）的集合:

```
### Create a collection.
>>> milvus.create_collection(param)
```

删除集合

```
### Drop a collection.
>>> milvus.drop_collection(collection_name='test01')
```

常见问题

创建集合时 index_file_size 如何设置能达到性能最优?

使用客户端创建集合时有一个 index_file_size 参数，用来指定数据存储时单个文件的大小，其单位为 MB，默认值为 1024。当向量数据不断导入时，Milvus 会把数据增量式地合并成文件。当某个文件达到 index_file_size 所设置的值之后，这个文件就不再接受新的数据，Milvus 会把新的数据存成另外一个文件。这些都是原始向量数据文件，如果建立了索引，则每个原始文件会对应生成一个索引文件。Milvus 在进行搜索时，是依次对每个索引文件进行搜索。

根据我们的经验，当 index_file_size 从 1024 改为 2048 时，搜索性能会有 30% ~ 50% 左右的提升。但要注意如果该值设得过大，有可能导致大文件无法加载进显存（甚至内存）。比如显存只有 2 GB，该参数设为 3 GB，显存明显放不下。

如果向集合中导入数据的频率不高，建议将 index_file_size 的值设为 1024 MB 或者 2048 MB。如果后续会持续地向集合中导入增量数据，为了避免查询时未建立索引的数据文件过大，建议这种情况下将该值设置为 256 MB 或者 512 MB。

可参阅 如何设置 Milvus 客户端参数。

建立集合后，index_file_size 和 metric_type 还支持修改吗?

不支持。

Milvus 对集合和分区的总数有限制吗?

collection 数量没有限制。每个 collection 内的 partition 总数不能超过 4096 个。

Milvus 支持的向量维度的最大值是多少?

Milvus 最多能够支持 32,768 向量维度。

创建、删除分区

本页提供创建或删除分区的 Python 示例代码。

创建分区

你可以通过标签将集合分割为若干个分区，从而提高搜索效率。每个分区实际上也是一个集合。

```
### Create a partition.
>>> milvus.create_partition('test01', 'tag01')
```

删除分区

```
>>> milvus.drop_partition(collection_name='test01', partition_tag='tag01')
```

插入、删除向量

你可以在集合或集合的分区中进行向量操作，本页提供以下内容：

- 在集合中插入向量
- 在分区中插入向量
- 通过 ID 删除向量

在集合中插入向量

1. 随机生成 20 个 256 维的向量：

   ```
   >>> import random
   ### Generate 20 vectors of 256 dimensions.
   >>> vectors = [[random.random() for _ in range(256)] for _ in range(20)]
   ```

2. 插入向量列表。如果你不指定向量 ID，Milvus 自动为向量分配 ID。

   ```
   ### Insert vectors.
   >>> milvus.insert(collection_name='test01', records=vectors)
   ```

   你也可以自己定义向量 ID：

   ```
   >>> vector_ids = [id for id in range(20)]
   >>> milvus.insert(collection_name='test01', records=vectors, ids=vector_ids)
   ```

在分区中插入向量

```
>>> milvus.insert('test01', vectors, partition_tag="tag01")
```

通过 ID 删除向量

假设你的集合中存在以下向量 ID：

```
>>> ids = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

你可以通过以下命令删除向量：

```
>>> milvus.delete_entity_by_id(collection_name='test01', id_array=ids)
```

在调用 delete 接口后，用户可以选择再调用 flush，保证新增的数据可见，被删除的数据不会再被搜到。

常见问题

Milvus 中自定义 ID 有没有长度限制?

ID 类型是非负的 64 位整型。

Milvus 可以插入重复 ID 的向量吗?

可以，这样在 Milvus 中会存在相同 ID 的多条向量。

Milvus 是否支持 “边插入边查询”?

支持。

Milvus 中单次插入数据有上限吗?

单次插入数据不能超过 256 MB。

Milvus 支持的向量维度的最大值是多少?

Milvus 最多能够支持 32,768 向量维度。

创建、删除索引

本页提供创建或删除索引的 Python 示例代码。

创建索引

目前，一个集合只支持一种索引类型，切换索引类型会自动删除旧的索引文件。在创建其它索引前，FLAT 作为集合的默认索引类型。

create_index() 会指定该集合的索引类型，并同步为之前插入的数据建立索引，后续插入的数据在大小达到 index_file_size 时，索引会在后台自动建立。在实际生产环境中，如果是流式数据，建议在插入向量之前先创建索引，以便后续系统自动建立；如果是静态数据，建议导入所有数据后再一次性创建索引。更多索引用法请参考 索引示例程序。

1. 准备创建索引所需参数（以 IVF_FLAT 为例）。索引参数是一个 JSON 字符串，在 Python SDK 中以字典来表示。

   ```
   ### Prepare index param.
   >>> ivf_param = {'nlist': 16384}
   ```

   对于不同的索引类型，创建索引所需参数也有区别。所有的索引参数都必须赋值。详细信息请参考 Milvus 索引类型。

2. 为指定集合创建索引：

   ```
   ### Create an index.
   >>> milvus.create_index('test01', IndexType.IVF_FLAT, ivf_param)
   ```

使用 IVF_SQ8H 建索引或查询时必须启用 GPU。

删除索引

删除索引后，集合再次使用默认索引类型 FLAT。

```
>>> milvus.drop_index('test01')
```

常见问题

建索引参数 nlist 的大小该如何选择?

该值需要根据具体的使用情况去设置。详见 性能调优 > 索引 和 如何设置 Milvus 参数。

Milvus 可以在同一个集合中的不同分区上建立不同索引吗?

不可以。同一个集合在某一刻只能有一种索引。

Milvus 中支持新增向量后再建索引吗?

支持。Milvus 中数据是分文件存储的，后续新增向量会存在新的数据文件中。该文件达到一定量后会自动触发建立索引，生成一个新的索引文件，不会影响之前已经建立过的索引。

查询向量

Milvus 支持在集合或分区中查询向量。

在集合中查询向量

1. 创建搜索参数。搜索参数是一个 JSON 字符串，在 Python SDK 中以字典来表示。

   ```
   >>> search_param = {'nprobe': 16}
   ```

   对于不同的索引类型，搜索所需参数也有区别。所有的搜索参数都必须赋值。详细信息请参考 Milvus 索引类型。

2. 创建随机向量作为 query_records 进行搜索：

   ```
   ### Create 5 vectors of 256 dimensions.
   >>> q_records = [[random.random() for _ in range(256)] for _ in range(5)]
   >>> milvus.search(collection_name='test01', query_records=q_records, top_k=2, params=search_param)
   ```

   top_k 指的是向量空间中距离目标向量最近的 k 个向量。

   top_k 的范围为：[1, 1048576]。

在分区中查询向量

```
### Create 5 vectors of 256 dimensions.
>>> q_records = [[random.random() for _ in range(256)] for _ in range(5)]
>>> milvus.search(collection_name='test01', query_records=q_records, top_k=1, partition_tags=['tag01'], pa
```

如果你不指定 partition_tags，Milvus 会在整个集合中搜索。

常见问题

为什么 Milvus 查询召回率一直不理想?

在调用 SDK 进行向量搜索时，可以增大函数中 nprobe 参数的值。值越大，结果越精确，但耗时也越久。详见 如何设置 Milvus 客户端参数。

Milvus 是否支持 “边插入边查询”?

支持。

对集合分区的查询是否会受到集合大小的影响，尤其在集合数据量高达一亿数据量时?

不会。如果你在搜索时指定了分区，Milvus 只会在相应分区进行搜索。

如果只是搜索集合中的部分分区，整个集合的数据会全部加载到内存吗?

不会，只加载指定的分区里的数据。

各个数据段的检索是并行处理的吗?

一般而言，Milvus 对单个数据段内的查询是并行的，多个数据段的处理根据发行版本略有不同。

假设一个集合存在多个数据段，当查询请求到达时：

CPU 版 Milvus 会对数据段读取任务和段内查询任务进行流水线处理。

GPU 版 Milvus 会在 CPU 版的基础上，将多个数据段分配给各个 GPU 处理。

可参阅文章：Milvus 开源向量数据库 ANNS。

批量搜索时，用多线程的收益大吗?

多线程查询，如果是小批量（nq $<$ 64）的话，后台会合并查询请求。如果是大批量查询的话，就不会有什么优势。

为什么搜索的速度非常慢?

请首先检查 server_config.yaml 的 cache.cache_size 参数是否大于集合中的数据量。

创建索引立即查询，为什么内存会突然增长?

这是因为 Milvus 在进行搜索时会将新生成的索引文件加载到内存，由于加载的索引文件和用于生成索引文件的原始向量文件总和小于 cache.cache_size 的上限，原始向量数据暂未被系统从内存释放。

为什么重启 Milvus 服务端之后，第一次搜索时间非常长?

重启后第一次搜索时，会将数据从磁盘加载到内存，所以这个时间会比较长。可以在 server_config.yaml 中开启 preload_collection，在内存允许的情况下尽可能多地加载集合。这样在每次重启服务端之后，数据都会先载入到内存中，可以解决第一次搜索耗时很长的问题。或者在查询前，调用方法 load_collection() 将该集合加载到内存。

数据落盘

在进行有关数据更改的操作时，你可以将集合中的数据从内存中进行 flush 操作使数据落盘。Milvus 也会执行自动落盘。自动落盘会在固定的时间周期（1 秒）对所有现存集合的数据进行落盘操作。

```
>>> milvus.flush(collection_name_array=['test01'])
```

在调用 delete 接口后，用户可以选择再调用 flush，保证新增的数据可见，被删除的数据不会再被搜到。

常见问题

为什么数据插入后不能马上被搜索到?

因为数据还没有落盘。要确保数据插入后立刻能搜索到，可以手动调用 flush 接口。但是频繁调用 flush 接口可能会产生大量小数据文件，从而导致查询变慢。

数据段整理

数据段是 Milvus 自动将插入的向量数据合并所获得的数据文件。一个集合可包含多个数据段。如果一个数据段中的向量数据被删除，其所占据的空间并不会自动释放。你可以对集合中的数据段进行 compact 操作以释放多余空间。

```
>>> milvus.compact(collection_name='test01', timeout=1)
```

关闭客户端

本页介绍如何关闭 Python 客户端。

```
>>> milvus.close()
```

参考手册

服务端配置

配置概述

以下配置说明可同时应用于单机或者分布式场景。

Milvus 文件结构

成功启动 Milvus 服务后，你可以在 home/$USER/milvus 的路径下看到 Milvus 的文件夹。其中包含以下文件:

- milvus/db（数据库存储）
- milvus/logs（日志存储）
- milvus/conf（设置文件）
  - server_config.yaml（服务设置）

配置修改

编辑配置文件

你可以直接编辑配置文件对配置进行修改。如果修改了配置文件，你必须重启 Milvus 服务来启用新的更改。

```
$ docker restart <container_id>
```

下面以 Milvus 的系统配置文件 server_config.yaml 为例，演示如何修改日志级别和日志存储路径：

```
logs:
  level: info
  path: /var/lib/milvus/logs
```

运行时修改

你可以使用 Milvus 客户端对 server_config.yaml 的配置进行运行时修改。详情请参考客户端参考。

对以下参数的运行时修改是立即生效的：

- cache 区域
  - cache_size
  - insert_buffer_size
- gpu 区域
  - enable
  - cache_size
  - gpu_search_threshold
  - search_devices
  - build_index_devices

对于其它参数，你必须重新启动 Milvus 才能使改动生效。

server_config.yaml 参数说明

若有任何疑问，欢迎在 GitHub 上给我们 创建 issue 或是 加入 Slack 社区讨论。

cluster 区域

参数	说明	类型	默认值
enable	是否开启集群模式。	Boolean	false
role	节点的运行模式，可选配置：	Role	rw

general 区域

参数	说明	类型	默认值
timezone	使用 UTC-x 或 UTC+x 来指定时区。比如，可以使用 UTC+8 来代表中国标准时间。	Timezone	UTC+8
meta_uri	元数据存储的 URI。可以使用 SQLite（Milvus 单机版本）或者 MySQL（Milvus 分布式版本）来存储元数据。URI 格式为 dialect://username:password@host:port/database。其中，dialect 可以是 sqlite 或者 mysql，其他文字需要替换成实际值。	URI	sqlite://:@:/

network 区域

参数	说明	类型	默认值
bind.address	Milvus 服务端监听的 IP 地址。	IP	0.0.0.0
bind.port	Milvus 服务端监听的端口号，范围：[1025, 65534]。	Integer	19530
http.enable	是否开启 HTTP 服务。	Boolean	true

参数	说明	类型	默认值
`http.port`	Milvus HTTP 服务监听的端口号，范围：[1025, 65534]。	Integer	`19121`

`storage` 区域

参数	说明	类型	默认值
`path`	Milvus 数据文件（向量文件、索引文件和元数据）的存储路径。	Path	`/var/lib/milvus`
`auto_flush_interval`	Milvus 定期将缓冲区数据落盘的时间间隔（单位为秒），范围：[0, 3600]。如果该值为 0，则关闭定期落盘功能。	Integer	`1`

`wal` 区域

参数	说明	类型	默认值
`enable`	是否开启 WAL（Write-Ahead Logging，预写式日志）。如果启用 WAL，Milvus 会在更改数据之前将所有数据更改写入日志文件。WAL 确保 Milvus 操作的原子性和耐用性。	Boolean	`true`
`recovery_error_ignore`	在通过 WAL 执行恢复操作时，是否忽略出现错误的日志。	Boolean	`false`
`buffer_size`	WAL 缓冲区大小，范围：64MB ~ 4096MB。建议该值大于单次插入的数据量两倍，以获取更好的性能。格式请见空间大小的格式。	String	`256MB`
`path`	预写式日志文件存储路径。	String	`/var/lib/milvus/wal`

`cache` 区域

参数	说明	类型	默认值
`cache_size`	内存中用于驻留搜索数据的缓存空间大小。`cache_size` 和 `insert_buffer_size` 之和不能超过内存总量。格式请见空间大小的格式。	String	`4GB`
`insert_buffer_size`	用于数据导入的缓冲区所使用的内存量。`cache_size` 和 `insert_buffer_size` 之和不能超过内存总量。格式请见空间大小的格式。	String	`1GB`
`preload_collection`	在 Milvus 开启时，需要加载的集合列表。	StringList	N/A

`gpu` 区域

在该区域选择是否在 Milvus 里启用 GPU 用于搜索和索引创建。同时使用 CPU 和 GPU 可以达到资源的最优利用，在特别大的数据集里做搜索时性能更佳。

若要切换到 CPU-only 模式，只要将 `enable` 设置为 `false`。

参数	说明	类型	默认值
`enable`	是否开启 GPU 用于搜索和索引创建。	Boolean	`false`
`cache_size`	显存中用于驻留搜索数据的缓存空间大小。该值不能超过显存总量。格式请见空间大小的格式。	String	`1GB`
`gpu_search_threshold`	使用 GPU 查询的阈值。用 `nq` 表示单批次查询的向量数，则搜索计算方式如下：	Integer	`1000`
`search_devices`	用于搜索的 GPU 设备，格式为“gpux”。其中“x”是 GPU 的序号，例如“gpu0”。	DeviceList	`gpu0`
`build_index_devices`	用于创建索引的 GPU 设备，格式为“gpux”。其中“x”是 GPU 的序号，例如“gpu0”。	DeviceList	`gpu0`

在 Milvus 里，创建索引和搜索是两个独立分开的过程，可以只在 CPU，或同时在 CPU 和 GPU 里进行。通过将 GPU 添加至 search_devices 或者 build_index_devices 下方，你可以指定多个 GPU 设备来进行创建索引或搜索。请参考下面的 YAML 示例代码：

```yaml
search_devices:
  - gpu0
  - gpu1
build_index_devices:
  - gpu0
  - gpu1
```

`logs` 区域

参数	说明	类型	默认值
`level`	日志打印的最低级别。日志级别：`debug` < `info` < `warning` < `error` < `fatal`。	String	`debug`
`trace.enable`	是否开启 trace 级别日志打印。	Boolean	`true`
`path`	日志存储路径。	String	`/var/lib/milvus/logs`
`max_log_file_size`	单个日志文件的大小限制，范围：1024MB ~ 4096MB。格式请见空间大小的格式。	Integer	`1024MB`
`log_rotate_num`	每个日志级别最多保存的文件数量，范围：[0, 1024]。0 代表日志文件数无限制。	Integer	`0`

`metric_config` 区域

参数	说明	类型	默认值
`enable`	是否开启 Prometheus 监控。	Boolean	`false`
`address`	访问 Prometheus Pushgateway 的 IP 地址。	IP	`127.0.0.1`
`port`	访问 Prometheus Pushgateway 的端口号。范围：[1025, 65534]。	Integer	`9091`

在 Milvus 配置文件中，空间大小的格式为“数字 + 单位”，如“4GB”。

数字和单位之间没有空格。

数字必须是整数。

可选单位为 GB、MB、KB。

常见问题

除了配置文件外，怎样可以判断我确实在使用 GPU 做搜索?

有以下三种方式:

使用 nvidia-smi 命令查看 GPU 使用情况。

用 Prometheus 配置，详见 使用 Grafana 展示监控指标 > 系统运行指标。

使用 Milvus 服务端的日志。

如果设置了 preload_collection，必须是等集合全部加载到内存，服务才能访问吗?

是的。如果是在 server_config.yaml 里设置的，那在启动 Milvus 时就会加载好数据后才开始提供服务。

为什么查询时 GPU 一直空闲?

此时应该是在用 CPU 进行查询。如果要用 GPU 进行查询，需要在配置文件中将 gpu_search_threshold 的值设置为小于 nq (每次查询的向量条数) 。可以将 gpu_search_threshold 的值调整为期望开启 GPU 搜索的 nq 数。若 nq 小于该值，则用 CPU 查询，否则将使用 GPU 查询。不建议在查询批量较小时使用 GPU 搜索。

为什么我的日志文件时间与系统时间不一致?

Docker 镜像内部的日志文件默认使用 UTC 时间。如果宿主机未使用 UTC 时间，就会出现日志文件时间与系统时间不一致的情况。建议在宿主机上挂载日志文件，这样可以保证宿主机上的日志文件和系统时间是一致的。

使用 MySQL 管理元数据

Milvus 默认使用 SQLite 管理元数据，SQLite 具有易于使用、高鲁棒的特性，且无需启动额外服务。但是在生产环境中，基于可靠性的考虑，我们强烈建议你使用 MySQL。

Milvus 在 CentOS 系统中不支持 MySQL 8.0 或更高版本。

请参考以下步骤使用 MySQL 作为元数据管理服务:

1. 拉取 MySQL 最新镜像。

   ```
   $ docker pull mysql:5.7
   ```

2. 启动 MySQL 服务（密码和端口可自行设置）。

   ```
   $ docker run -p 3306:3306 -e MYSQL_ROOT_PASSWORD=123456 -d mysql:5.7
   ```

3. 使用 root 账号和 MySQL 服务所在的主机 IP （`<MySQL_server_host IP>`）登录 MySQL，回车后系统提示输入密码。输入上一步设置的密码。

   ```
   $ mysql -h<MySQL_server_host IP> -uroot -p
   ```

4. 进入 MySQL 客户端命令行，创建一个 database，名称可自行设定，这里使用 `milvus`。

   ```
   mysql> create database milvus;
   ```

5. 退出 MySQL 客户端, 修改 server_config.yaml 文件的 `meta_uri` 参数。使用 MySQL 服务所在的主机 IP 作为 IP 地址（`<MySQL_server_host IP>`）。注意密码、IP 地址、端口以及 database 名称要和以上几步的设置一致。

   ```
   meta_uri: mysql://root:123456@<MySQL_server_host IP>:3306/milvus
   ```

6. 使用修改过的 server_config.yaml 启动 Milvus 服务。

常见问题

出现 database is locked 的报错怎么解决?

如果元数据管理用的是 SQLite，在有数据频繁写入的情况下会出现该错误。建议将 SQLite 更换为 MySQL。如何更换请参考文档 使用 MySQL 管理元数据。

为什么我在 SQLite / MySQL 找不到向量数据?

SQLite / MySQL 只是存放原始向量数据的元数据。向量和索引直接以文件的形式存在磁盘上，不存放在 SQLite 或 MySQL 里。详见 存储相关概念。

Milvus 的元数据存储可以使用 SQL Server 或者 PostgreSQL 吗?

不可以，目前仅支持 SQLite 和 MySQL。

数据管理相关博客

从数据导入，数据存储到数据查询和调度，请参阅我们的博客深入了解 Milvus 数据管理方案。

- 数据管理策略
- 数据文件清理机制的改进
- 查看元数据
- 元数据表的字段
- 如何通过元数据管理数据文件

监控与报警

监控与报警方案概述

监控与报警

方案概述

在生产环境下，我们需要对数据库系统进行积极全面的性能监控，以及给需要调查干预的突发情况创建报警规则以触发用户通知。

Milvus 使用的监控与报警方案如下：

- 使用 Prometheus 监控和存储性能指标存储：
  - Prometheus Server：收集和存储时序数据。
  - Client 代码库：定制监控指标。
  - Pushgateway：推送指标数据，确保生命周期短且难以被及时提取的监控指标能够被 Prometheus 获取。
  - Alertmanager：实现报警机制。
- 使用开源的时序数据分析及可视化平台 Grafana 展示各项性能指标。

工作流程

Milvus 会收集监控数据并将其推送到 Pushgateway。同时，Prometheus 服务器会定期从 Pushgateway 中拉取数据并将其保存到它的时序数据库。具体工作流程如下：

proxy

配置、启动 Prometheus

本页面介绍如何配置、启用 Prometheus 搜集 Milvus 监控指标，以及如何将 Alertmanager 连接到 Prometheus 实现数据可视化的展示和报警机制。

安装 Prometheus

1. 下载 Prometheus 二进制文件的压缩包。
2. 确保 Prometheus 已经成功安装：

```
$ prometheus --version
```

你可以将 Prometheus 的路径添加到 PATH，以便在任意 Shell 上都能快速启动 Prometheus。

配置、启动 Prometheus

1. 启动 Pushgateway：

   ```
   ./pushgateway
   ```

   必须在启动 Milvus Server 之前启动 Pushgateway 进程。

2. 在 server_config.yaml 中开启 Prometheus 监控，并设置 Pushgateway 的地址和端口号。

   ```
   metric:
     enable: true                    ### 将值设为 true 以开启 Prometheus 监控。
     address: <your_IP_address>      ### 设置 Pushgateway 的 IP 地址。
     port: 9091                      ### 设置 Pushgateway 的端口号。
   ```

   如果是在 Kubernetes 集群中，你需要为每个需要监控的 Milvus 节点配置 server_config.yaml。

3. 下载 Milvus Prometheus 配置文件：

   ```
   $ wget https://raw.githubusercontent.com/milvus-io/docs/master/v1.1.1/assets/monitoring/prometheus.yml
   ```

4. 下载 Milvus 报警规则文件 到 Prometheus 根目录：

   ```
   wget -P rules https://raw.githubusercontent.com/milvus-io/docs/master/v1.1.1/assets/monitoring/alert_r
   ```

5. 根据实际需求编辑 Prometheus 配置文件：

   - global：配置 scrape_interval 和 evaluation_interval 等参数。

   ```
   global:
     scrape_interval:     2s ### 设置抓取时间间隔为 2s。
     evaluation_interval: 2s ### 设置评估时间间隔为 2s。
   ```

   - alerting：设置 Alertmanager 的地址和端口。

   ```
   alerting:
   alertmanagers:
   - static_configs:
     - targets: ['localhost:9093']
   ```

   - rule_files：设置报警规则文件。

   ```
   rule_files:
     - "alert_rules.yml"
   ```

   - scrape_configs：设置抓取数据的 job_name 及 targets 等信息。

   ```
   scrape_configs:
   - job_name: 'prometheus'
     static_configs:
     - targets: ['localhost:9090']

   - job_name: 'pushgateway'
     honor_labels: true
     static_configs:
     - targets: ['localhost:9091']
   ```

   关于 Prometheus 的高级配置和功能的详细信息请见 配置 Prometheus。

6. 启动 Prometheus：

   ```
   ./prometheus --config.file=prometheus.yml
   ```

启动 Prometheus 后，你可以在 Prometheus 界面上配置 Milvus 提供的监控指标。详见：Milvus 监控指标。

配置 Alertmanager

需要报警的事件

积极的监控帮助及早发现问题，但创建报警规则以便在出现突发事件时触发用户通知也非常有必要。

以下主要介绍需要创建报警规则的事件。

服务器宕机

- 报警规则：当 Milvus 服务器宕机时发送报警信息。
- 如何判断：当 Milvus 服务器宕机时，监控仪表盘上各个指标会显示 No Data。

CPU/GPU 温度过高

- 报警规则：当 CPU/GPU 温度超过 80 摄氏度时发送报警信息。
- 如何判断：查看监控仪表盘上的 CPU Temperature 和 GPU Temperature 两个指标。

配置步骤

1. 下载 Alertmanager 二进制文件的压缩包。
2. 确保 Alertmanager 已经成功安装：

   ```
   $ alertmanager --version
   ```

   你可以将 Alertmanager 的路径并添加到 PATH，以便在任意 Shell 上都能快速启动 Alertmanager。
3. 根据 配置 Alertmanager 创建配置文件 alertmanager.yml，指定接受报警通知的邮箱或微信账号，并将配置文件添加到 Alertmanager 根目录下。
4. 启动 Alertmanager 服务并指定配置文件：

   ```
   ./alertmanager --config.file=alertmanager.yml
   ```
5. 通过浏览器登录 http://:9093，进入 Alertmanager 用户交互页面。你可以在此定义 报警的条件。

常见问题

在多个 Milvus 节点接入 Pushgateway 的情况下如何进行区分数据来源?

在 prometheus.yaml 里面加一个 Prometheus 的实例就可以。最后在 Prometheus 或者 Grafana 里面显示监控的时候，会指明数据是来自哪个 Milvus 实例。

使用 Grafana 展示监控指标

使用 Grafana 展示 Milvus 监控指标

启动并配置 Grafana

1. 运行 Grafana：

```
docker run -i -p 3000:3000 grafana/grafana
```

2. 在浏览器中打开 http://:3000 网址，并登录 Grafana 用户交互页面。

Grafana 的默认用户名和密码都是 admin。你也可以在此创建新的 Grafana 账号。

3. 将 Prometheus 添加为数据源。
4. 在 Grafana 用户交互页面中，点击 Configuration > Data Sources > Prometheus，然后设置以下数据源属性：

名称	值
Name	Prometheus
Default	`True`
URL	http://:9090
Access	Browser

5. 下载 Grafana 配置文件。
6. 将配置文件导入 Grafana。

   prometheus.png

配置监控指标

你可以通过 Milvus 提供的 Grafana 配置文件 配置 Milvus 提供的各项监控指标。详见：Milvus 监控指标。

Milvus 监控指标

Milvus 会生成关于系统运行状态的详细时序 metrics。你可以通过 Prometheus、Grafana 或任何可视化工具展现以下指标：

- Milvus 性能指标
- 系统运行指标：CPU/GPU 使用状况，网络、磁盘读取等指标。
- 硬件存储指标：数据大小，数据文件及存储情况等指标。

Milvus 性能指标

指标	说明
Insert per Second	每秒钟插入的向量数量（实时显示）。
Queries per Minute	每分钟运行的查询数量（实时显示）。
Query Time per Vector	单条向量查询时间 = 查询使用时间 / 向量数量
Query Service Level	查询服务级别 = 一定时间阈值内的查询数量/总查询数量 一般建议设置 3 个时间阈值来跟踪查询服务级别。
Uptime	Milvus 服务器正常运行的时长（分钟）。

系统运行指标

指标	说明
GPU Utilization	GPU 利用率（%）。
GPU Memory Usage	Milvus 当前使用的显存量（GB）。
CPU Utilization	CPU 利用率（%）= 服务器任务执行时间 / 服务器总运行时间
Memory Usage	Milvus 当前使用的内存量（GB）。
Cache Utilization	缓存利用率（%）。
Network IO	网口的读写速度（GB/s）。
Disk Read Speed	磁盘读取速度（GB/s）。
Disk Write Speed	磁盘写入速度（GB/s）。

硬件存储指标

指标	说明
Data Size	Milvus 所存向量条数。
Total File	Milvus 所存数据总量（GB）。

性能调优

插入性能调优

“数据插入”到“数据写入磁盘”的基本流程请参考 存储操作。

如果数据量小于单次插入上限（256 MB），批量插入比单条插入要高效得多。

系统配置中的两个参数对插入性能有影响：

- `wal.enable`

该参数用于开启或关闭 预写日志（WAL, Write Ahead Log） 功能（默认开启）。开启和关闭预写日志功能时，插入数据的流程分别如下：

- 开启预写日志功能时，预写日志模块先将数据写入磁盘，然后返回插入操作。
- 关闭预写日志功能时，数据插入速度更快。系统直接将数据写入内存中的可写缓冲区，并立即返回插入操作。

但是对于 删除操作 来说，打开预写日志功能时速度更快。为了保证数据的可靠性，我们建议打开预写日志。

- `storage.auto_flush_interval`

该参数是指后台落盘任务的间隔时间，默认值为 1 秒。根据 Milvus 数据段合并策略，增大该值可减少段合并的次数，减少磁盘 I/O，提高插入操作的吞吐量。

Milvus 无法搜索到在该时间间隔内未落盘的数据。

另外，建立集合时的参数 `index_file_size` 也对插入性能有影响。该参数的默认值为 1024 MB，最大值为 128 GB。该参数越大，将文件合并到该值设定的大小所需的次数就越多，影响插入操作的吞吐量。该参数越小，则产生的数据段越多，查询性能可能会变差。

除了软件层面的因素外，网络带宽和存储介质对插入操作性能也有影响。

查询性能调优

影响查询性能的因素包括硬件环境、系统参数、索引、查询规模等。

硬件环境

- 使用 CPU 计算时，查询性能取决于 CPU 的主频、核心数和支持的指令集。

Milvus 在支持 AVX 指令集的 CPU 上的查询性能较好。

- 使用 GPU 计算时，查询性能取决于 GPU 的并行计算能力以及传输带宽。

系统参数

系统参数配置请参考 Milvus 服务端配置。

- `cache.cache_size`

该参数是指用于驻留查询数据的缓存空间大小，默认值为 4 GB。如果该缓存空间不足以容纳所需的数据，查询时会从磁盘临时加载数据，严重影响查询性能。因此，`cache_size` 应当大于查询所需的数据量。

- 浮点型原始向量的数据量可根据“向量总条数 × 维度 × 4”来估算。
- 二进制型原始向量的数据量可根据“向量总条数 × 维度 ÷ 8”来估算。

索引建立完成后（不包括 FLAT），索引文件需要额外占用磁盘空间，查询只需加载索引文件。

- IVF_FLAT 索引的数据量和其原始向量总数据量基本相等。
- IVF_SQ8 / IVF_SQ8H 索引的数据量相当于其原始向量总数据量的 25% ～30%。
- IVF_PQ 索引的数据量根据其参数变化，一般低于其原始向量总数据量的 10%。
- HNSW ／ RNSG ／ Annoy 索引的数据量都大于其原始向量总数据量。

通过调用 get_collection_stats 接口，可准确获知查询一个集合所需的数据总量。

- `gpu.gpu_search_threshold`

在 GPU 版本中，当目标向量数量大于等于该参数设定的值，将会启用 GPU 查询。该参数的默认值为 1000。

GPU 查询的性能取决于 CPU 将数据加载进显存的速度以及 GPU 的计算能力。在目标向量数量较少时，无法充分发挥出 GPU 并行计算的优势。只有当目标向量数量达到某个阈值后，GPU 的查询性能才会优于 CPU。在实际使用中，可根据实验对比得出该参数的理想值。

- `gpu.search_resources`

指定用于查询的 GPU 设备。对于数据插入和查询并发的场景，使用 GPU 建索引可避免后台建索引任务和查询任务争抢 CPU 资源。

- `gpu.build_index_resources`

指定用于建索引的 GPU 设备。对于查询目标向量数量较大的场景，使用多 GPU 可显著提高查询效率。

索引

向量索引的基本概念请参考 向量索引概述。

选择合适的索引需要在存储空间、查询性能、查询召回率等多个指标中权衡。

- FLAT 索引

FLAT 是对向量的暴力搜索（brute-force search），速度最慢，但召回率最高（100%），磁盘空间占用最小。

随着目标向量数量增多，使用 CPU 做 FLAT 查询的耗时呈线形上升关系；而使用 GPU 查询时，批量查询的效率高，目标向量数量增加对查询耗时影响不大。

- IVF 系列索引

IVF 系列索引包括 IVF_FLAT、IVF_SQ8 ／ IVF_SQ8H 和 IVF_PQ。IVF_SQ8 ／ IVF_SQ8H 和 IVF_PQ 索引对向量数据做了有损压缩，磁盘占用量较少。

IVF 索引都有两个相同的参数：`nlist` 和 `nprobe`。`nlist` 是建索引参数，`nprobe` 是搜索参数。选取 `nlist` 和 `nprobe` 的推荐值，详见 性能优化问题 > 应如何设置 IVF 索引的 nlist 和 nprobe 参数?

根据相关原理，估算使用 IVF 索引进行查询时的计算量：

- 单个数据段计算量可估算为：目标向量数量 × (`nlist` + （段内向量数 ÷ `nlist`）× `nprobe`)
- 数据段的数量可估算为：集合数据总量 ÷ `index_file_size`
- 对集合查询所需的计算总量则为：单个数据段计算量 × 数据段数量

通过估算得出的计算总量越大，查询耗时越长。实际使用中可根据以上公式确定合理的参数，在满足召回率的前提下获得较高的查询性能。

在持续插入数据的场景下，由于对大小未达到 index_file_size 的数据段未建立索引，对其使用的查询方式是暴力搜索。计算量为：目标向量数量 x 该数据段向量总数。

- HNSW / RNSG / Annoy 索引

HNSW、RNSG、Annoy 的索引参数对查询性能的影响较为复杂，建议参考 索引概览。

其他

- 结果集

结果集的规模取决于目标向量数量和 `topk`。`topk` 的大小对计算的影响不大。但在目标向量数量和 `topk` 都较大的情况下，结果集序列化和网络传输的耗时会相应增加。

- MySQL

Milvus 使用 MySQL 作为元数据后端服务。Milvus 在查询数据时会多次访问 MySQL 以获取元数据信息，因此 MySQL 服务的响应速度对 Milvus 的查询性能有较大影响。

- 预加载

首次查询需要先将数据从磁盘读入缓存，因此耗时较长。为避免首次查询加载数据，可预先调用 `load_collection` 接口，或使用系统参数 `preload_collection` 指定启动 Milvus 时要预先加载的集合。

- 段数据整理

在 数据段整理 中提到，Milvus 在查询数据时将 delete_docs 读入内存以过滤被删除的实体。调用 `compact` 接口可清理被删除的实体，减少过滤操作，从而提高查询性能。

存储优化

- 数据段整理

在 数据段整理 中提到，被删除的实体不参与计算，并且占用磁盘空间。如果有大量的实体已被删除，你可以调用 `compact` 接口来释放磁盘空间。

常见问题

为什么查询时 GPU 一直空闲?

此时应该是在用 CPU 进行查询。如果要用 GPU 进行查询，需要在配置文件中将 gpu_search_threshold 的值设置为小于 nq (每次查询的向量条数) 。可以将 gpu_search_threshold 的值调整为期望开启 GPU 搜索的 nq 数。若 nq 小于该值，则用 CPU 查询，否则将使用 GPU 查询。不建议在查询批量较小时使用 GPU 搜索。

为什么搜索的速度非常慢?

请首先检查 server_config.yaml 的 cache.cache_size 参数是否大于集合中的数据量。

创建集合时 index_file_size 如何设置能达到性能最优?

使用客户端创建集合时有一个 index_file_size 参数，用来指定数据存储时单个文件的大小，其单位为 MB，默认值为 1024。当向量数据不断导入时，Milvus 会把数据增量式地合并成文件。当某个文件达到 index_file_size 所设置的值之后，这个文件就不再接受新的数据，Milvus 会把新的数据存成另外一个文件。这些都是原始向量数据文件，如果建立了索引，则每个原始文件会对应生成一个索引文件。Milvus 在进行搜索时，是依次对每个索引文件进行搜索。

根据我们的经验，当 index_file_size 从 1024 改为 2048 时，搜索性能会有 30% ~ 50% 左右的提升。但要注意如果该值设得过大，有可能导致大文件无法加载进显存（甚至内存）。比如显存只有 2 GB，该参数设为 3 GB，显存明显放不下。

如果向集合中导入数据的频率不高，建议将 index_file_size 的值设为 1024 MB 或者 2048 MB。如果后续会持续地向集合中导入增量数据，为了避免查询时未建立索引的数据文件过大，建议这种情况下将该值设置为 256 MB 或者 512 MB。

可参阅 如何设置 Milvus 客户端参数。

为什么同样的数据量，用 GPU 查询比 CPU 查询慢?

一般来说，当 nq（每次查询的向量条数）较小时，用 CPU 查询比较快。只有当 nq 较大（约大于 500）时，用 GPU 查询才会更有优势。

因为在 Milvus 中，每次用 GPU 查询都需要将数据从内存加载到显存。只有当 GPU 查询节省的计算时间能抵消掉数据加载的时间，才能体现出 GPU 查询的优势。

为什么有时候小的数据集查询时间反而更长?

如果数据文件的大小小于创建集合时 index_file_size 参数的值，Milvus 则不会为此数据文件构建索引。因此，小的数据集有可能查询时间会更长。你还可以调用 create_index 建立索引。

Mishards: 集群分片中间件

Mishards：集群分片中间件

Mishards 是什么

Mishards 是一个用 Python 开发的 Milvus 集群分片中间件，其内部处理请求转发、读写分离、水平扩展和动态扩容，为用户提供内存和算力可以扩容的 Milvus 实例。

Mishards 简单工作原理

Mishards 负责将上游请求拆分，并路由到内部各细分子服务，最后将子服务结果汇总，返回给上游。

Mishards 目标场景

场景分类	并发度	延迟	数据规模	是否适合 Mishards
1	低	低	中等、小	否

场景分类	并发度	延迟	数据规模	是否适合 Mishards
2	高	低	中等、小	否
3	低	高	大	是
4	低	低	大	是
5	高	低	大	是

Mishards 适合大数据规模下的搜索场景。那么，怎么判断数据规模的大小呢？这个问题没有标准答案，因为这取决于实际生产环境中使用的硬件资源。这里提供一个简单的判断数据规模的思路：

1. 如果你不在意延迟，当数据规模大于单台服务器上硬盘的可用容量时，你可以认为这种场景具有大的数据规模。例如，每次批处理 5000 条查询请求，服务端计算时间已经大于数据从硬盘到内存的加载时间，此时将硬盘的可用容量作为判断数据规模的标准。
2. 如果你在意延迟，当数据规模大于单台服务器上的可用内存时，你也可以认为这种场景具有大的数据规模。

基于 Mishards 的集群方案

总体架构

主要构件

- 服务发现：获取读写节点的服务地址。
- 负载均衡器
- Mishards 节点：无状态, 可扩容。
- Milvus 写节点：单节点，不可扩容。为避免单点故障，需为该节点部署高可用 HA 方案。
- Milvus 读节点：有状态，可扩容。
- 共享存储服务：Milvus 读写节点通过共享存储服务共享数据，可选择 NAS 或 NFS。
- 元数据服务：目前只支持 MySQL。生产环境下需要部署 MySQL 高可用方案。

Mishards 配置

全局配置

参数	是否必填	类型	默认值	说明
`Debug`	否	Boolean	`True`	选择是否启用 Debug 工作模式。目前 Debug 模式只会影响日志级别。
`TIMEZONE`	否	String	`UTC`	时区
`SERVER_PORT`	否	Integer	`19530`	定义 Mishards 的服务端口。
`WOSERVER`	是	String		Milvus 写节点地址，参考格式：`tcp://127.0.0.1:19530`

元数据

元数据记录了底层数据的组织结构信息。在分布式系统中，Milvus 写节点是元数据唯一的生产者，而 Mishards 节点、Milvus 写节点和读节点都是元数据的消费者。目前版本的 Milvus 只支持 MySQL 和 SQLite 作为元数据的存储后端。

在分布式系统中，元数据的存储后端只能是 MySQL。

参数	是否必填	类型	默认值	说明
`SQLALCHEMY_DATABASE_URI`	是	String		定义元数据存储的数据库地址，格式标准为 RFC-738-style。例如：`mysql+pymysql://root:root@127.0.0.1:3306/milvus?charset=u`
`SQL_ECHO`	否	Boolean	`False`	是否打印 SQL 详细语句。

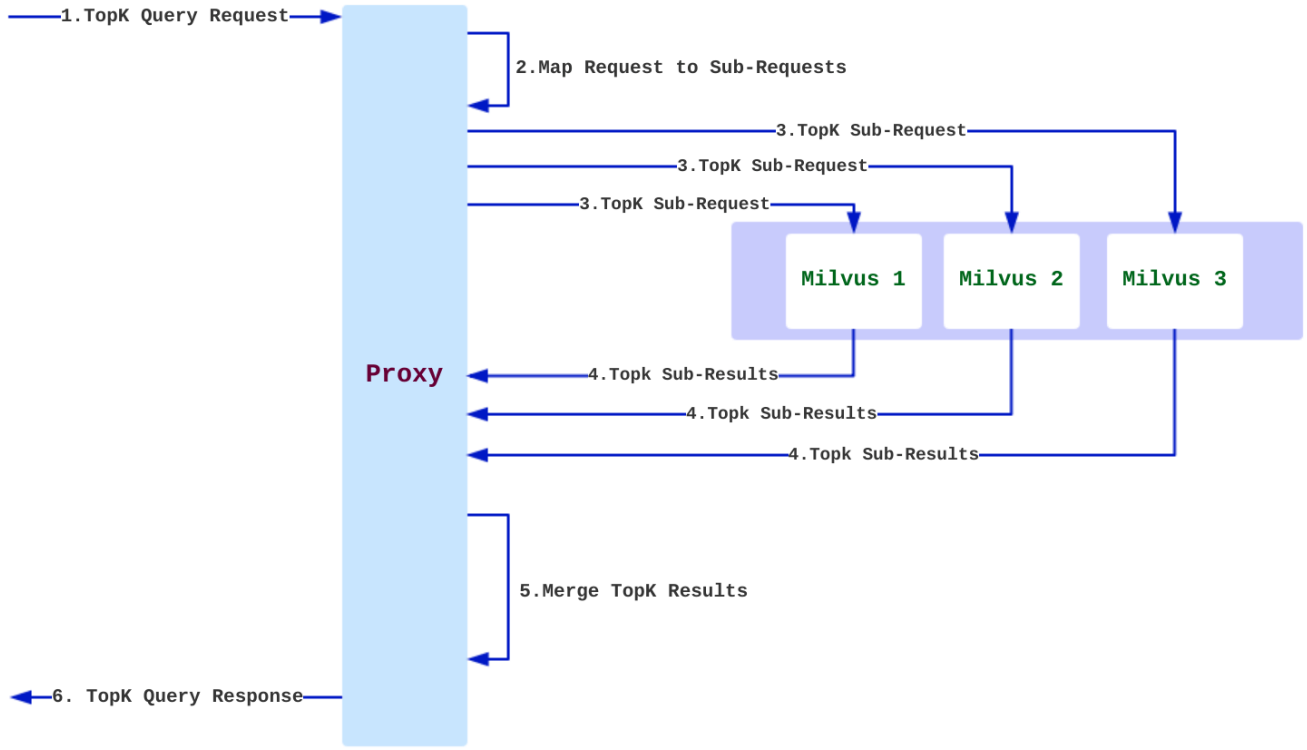

Figure 16: proxy

服务发现

服务发现为 Mishards 提供所有 Milvus 读写节点的地址信息。Mishards 定义了相关的服务发现接口 `IServiceRegistryProvider`，并通过插件模式提供了扩展的可能性。目前默认提供了两种插件：KubernetesProvider 对应 Kubernetes 集群；StaticProvider 对应静态配置。你可以仿照这两种插件的实现，定制自己的服务发现插件。

参数	是否必填	类型	默认值	说明
DISCOVERY_STATIC_HOSTS	否	List	[]	DISCOVERY_CLASS_NAME 为 static 时，定义服务地址列表。列表中的地址以逗号隔开，例如 192.168.1.188,192.168.1.190。
DISCOVERY_STATIC_PORT	否	Integer	19530	DISCOVERY_CLASS_NAME 为 static 时，定义服务地址监听端口。
DISCOVERY_PLUGIN_PATH	否	String		用户自定义服务发现插件的搜索路径，默认使用系统搜索路径。
DISCOVERY_CLASS_NAME	否	String	static	在插件搜索路径下，根据类名搜索类，并将其实例化。目前系统提供 static 和 kubernetes 两种类，默认使用 static。
DISCOVERY_KUBERNETES_NAMESPACE	否	String		DISCOVERY_CLASS_NAME 为 kubernetes 时，定义 Milvus 集群的 namespace。
DISCOVERY_KUBERNETES_IN_CLUSTER	否	Boolean	False	DISCOVERY_CLASS_NAME 为 kubernetes 时，选择服务发现是否在集群中运行。
DISCOVERY_KUBERNETES_POLL_INTERVAL	否	Integer	5	DISCOVERY_CLASS_NAME 为 kubernetes 时，定义服务发现监听周期（单位：秒）。
DISCOVERY_KUBERNETES_POD_PATT	否	String		DISCOVERY_CLASS_NAME 为 kubernetes 时，匹配 Milvus Pod 名字的正则表达式。
DISCOVERY_KUBERNETES_LABEL_SELECTOR	否	String		SD_PROVIDER 为 kubernetes 时，匹配 Milvus Pod 的标签，例如：tier=ro-servers。

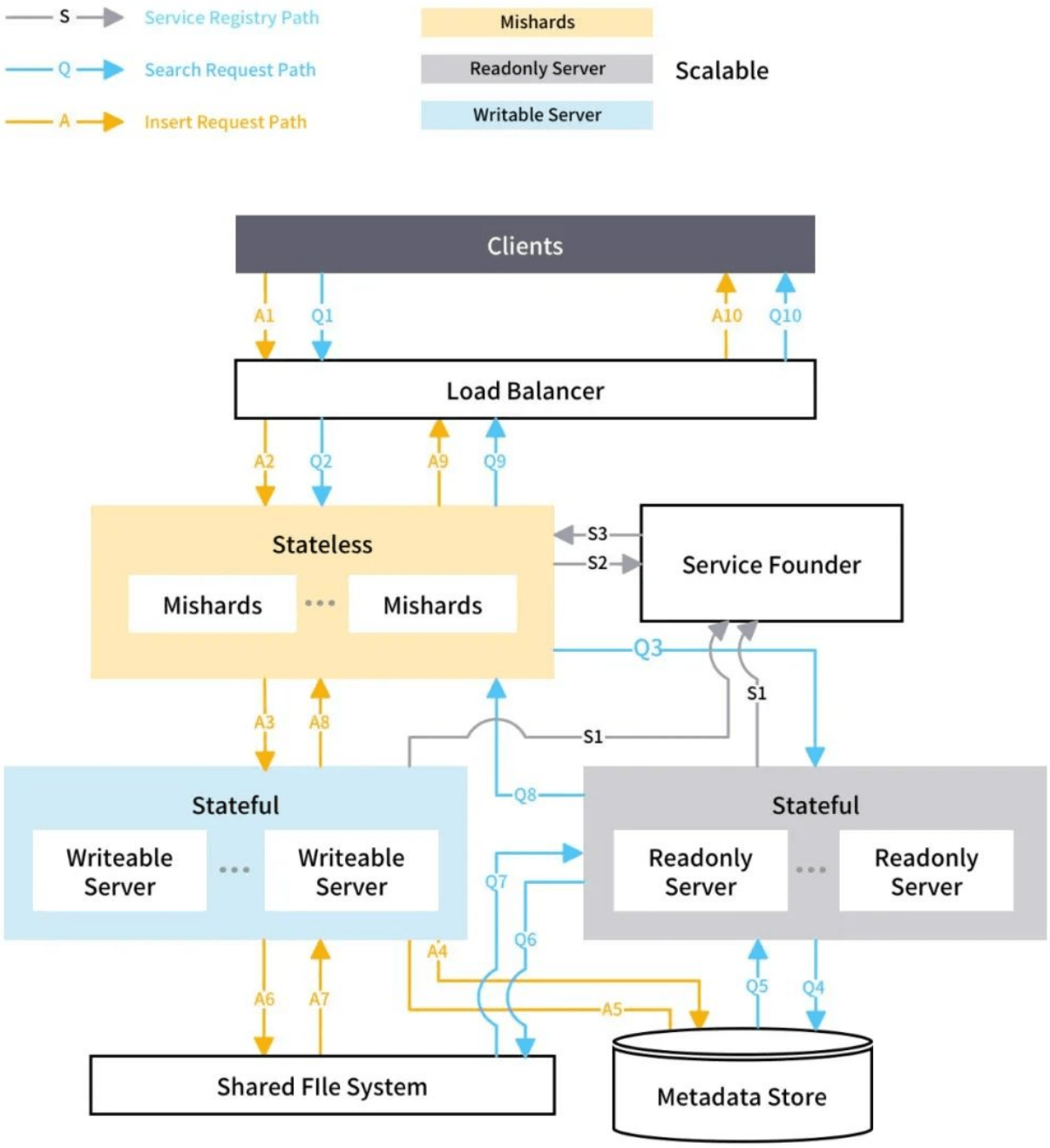

Figure 17: structure

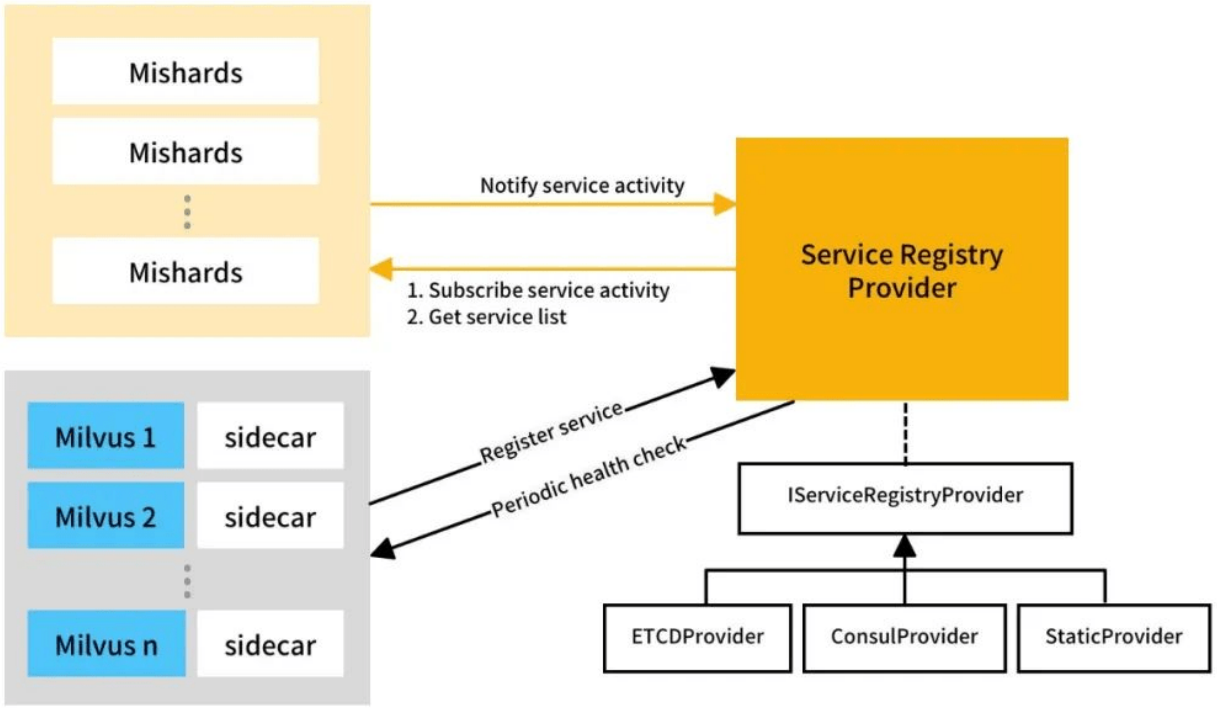

Figure 18: discovery

链路追踪

分布式系统错综复杂，请求往往会分发给内部多个服务调用。为了方便问题的定位，我们需要跟踪内部的服务调用链。系统的复杂性越高，一个可行的链路追踪系统带来的好处就越明显。我们选择了已进入 CNCF 的 OpenTracing 分布式追踪标准，OpenTracing 提供与平台和厂商无关的 API，便于开发人员实现链路跟踪系统。

Mishards 定义了相关的链路追踪接口，并通过插件模式提供了扩展的可能性。目前默认提供了基于 Jaeger 的插件。

查阅 Jaeger Doc 了解怎样集成 Jaeger。

参数	是否必填	类型	默认值	说明
TRACER_PLUGIN_PATH	否	String		用户自定义链路追踪插件的搜索路径，默认使用系统搜索路径。
TRACER_CLASS_NAME	否	String		在插件搜索路径下，根据类名搜索类，并将其实例化。目前只支持 Jaeger，默认不使用。
TRACING_SERVICE_NAME	否	String	mishards	TRACING_CLASS_NAME 为 Jaeger 时，链路追踪的服务。
TRACING_SAMPLER_TYPE	否	String	const	TRACING_CLASS_NAME 为 Jaeger 时，链路追踪的 采样类型。
TRACING_SAMPLER_PARAM	否	Integer	1	TRACING_CLASS_NAME 为 Jaeger 时，链路追踪的 采样频率。
TRACING_LOG_PAYLOAD	否	Boolean	False	TRACING_CLASS_NAME 为 Jaeger 时，链路追踪是否采集 Payload。

日志

集群服务日志文件分布在不同的服务节点上，因此你在排查问题需要登录到相关服务器获取日志。建议使用 ELK 日志分析组件来协同分析多个日志文件、排查问题。

参数	是否必填	类型	默认值	说明
LOG_LEVEL	否	String	DEBUG	日志记录级别：DEBUG < INFO < WARNING < ERROR。
LOG_PATH	否	String	/tmp/mishards	日志文件的存储路径
LOG_NAME	否	String	logfile	日志文件的名称

路由

Mishards 从服务发现中心获取 Milvus 读写节点的地址信息，通过元数据服务获取底层数据元信息。Mishards 的路由策略就是对于这些素材的一种消费。如图有 10 个数据段（s1, s2, s3, ⋯, s10）。现在选择基于数据段名字的一致性哈希路由策略（FileNameHashRingBased），Mishards 会将涉及 s1、s4、s6、s9 数据段的请求路由到 Milvus 1 节点，将涉及 s2、s3、s5 数据段的请求路由到 Milvus 2 节点，将涉及 s7、s8、s10 数据段的请求路由到 Milvus 3 节点。

Mishards 定义了路由策略相关的接口，并通过插件提供扩展。你可以仿照默认的一致性哈希路由插件，根据自己的业务特点定制个性化路由。

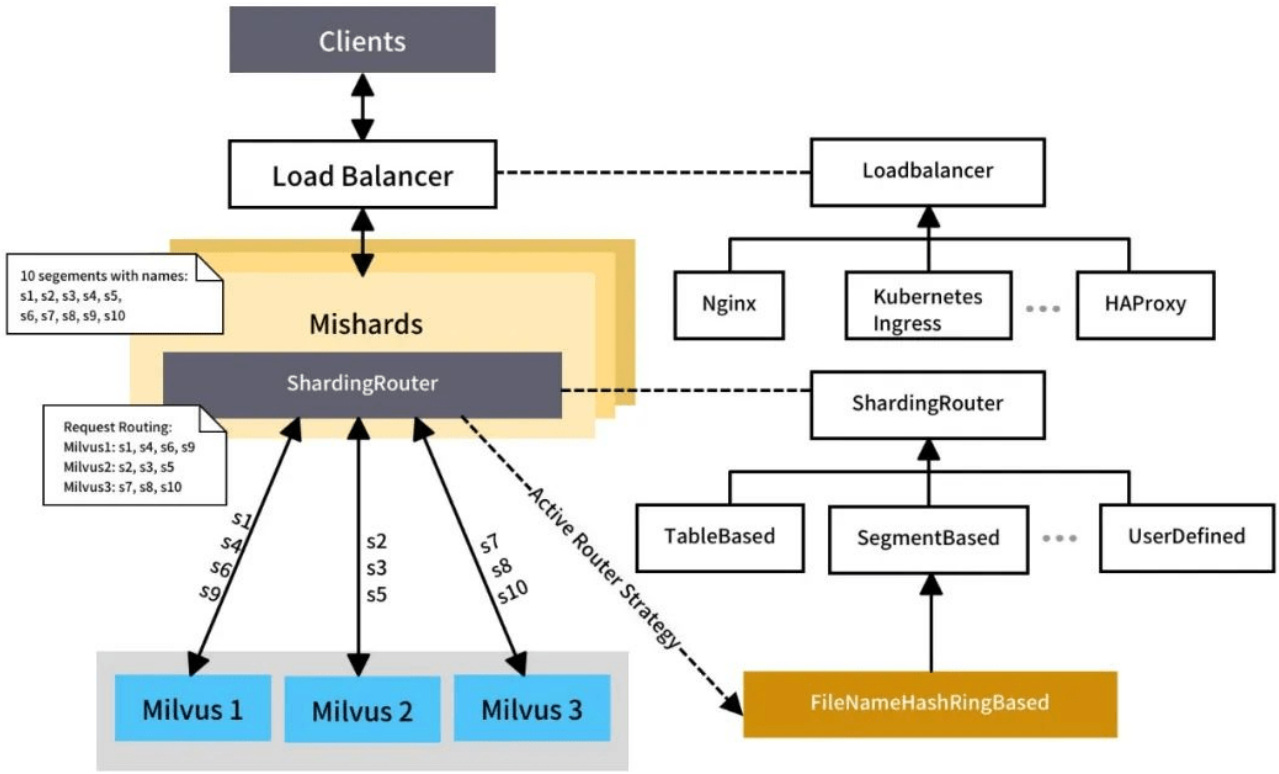

Figure 19: router

参数	是否必填	类型	默认值	说明
ROUTER_CLASS_NAME	否	String	FileBasedHashRingRouter	在插件搜索路径下，根据类名搜索路由的类，并将其实例化。目前系统只提供了基于数据段名字的一致性哈希路由策略 FileBasedHashRingRouter。
ROUTER_PLUGIN_PATH	否	String		用户自定义路由插件的搜索路径，默认使用系统搜索路径。

Mishards 简单示例

启动示例

前提条件

- 安装 Milvus
- Python 版本 3.6 及以上

启动 Milvus 和 Mishards 实例

请按照以下步骤在单机上启动单个 Milvus 实例和 Mishards 服务：

1. 将 Milvus 仓库复制到本地：

   ```
   git clone https://github.com/milvus-io/milvus -b 1.1
   ```

2. 安装 Mishards 的依赖库：

   ```
   $ cd milvus/shards
   $ pip install -r requirements.txt
   ```

3. 启动 Milvus 服务：

   - 如果 Docker 版本低于 v19.03，运行如下命令：

   ```
   $ sudo docker  run --runtime=nvidia --rm -d -p 19530:19530 -v /tmp/milvus/db:/var/lib/milvus/db milvus
   ```

   - 否则，运行如下命令：

   ```
   $ sudo docker run --gpus all --rm -d -p 19530:19530 -v /tmp/milvus/db:/var/lib/milvus/db milvusdb/milv
   ```

4. 更改目录权限：

   ```
   $ sudo chown -R $USER:$USER /tmp/milvus
   ```

5. 配置 Mishards 环境变量：

   ```
   $ cp mishards/.env.example mishards/.env
   ```

6. 启动 Mishards 服务：

   ```
   $ python mishards/main.py
   ```

使用 docker-compose 启动示例

`all_in_one` 使用 Docker 容器启动 2 个 Milvus 实例、1 个 Mishards 中间件实例和 1 个 Jaeger 链路追踪实例。

1. 安装 Docker Compose。
2. 将 Milvus 仓库复制到本地：

   ```
   $ git clone https://github.com/milvus-io/milvus -b 1.1
   $ cd milvus/shards
   ```

3. 启动所有服务：

   ```
   $ make deploy
   ```

4. 检查确认服务状态：

   ```
   $ make probe_deploy
   Pass ==> Pass: Connected
   Fail ==> Error: Fail connecting to server on 127.0.0.1:19530. Timeout
   ```

若要查看服务踪迹，使用浏览器打开 Jaeger 页面。

若要清理所有服务，请使用如下命令：

```
$ make clean_deploy
```

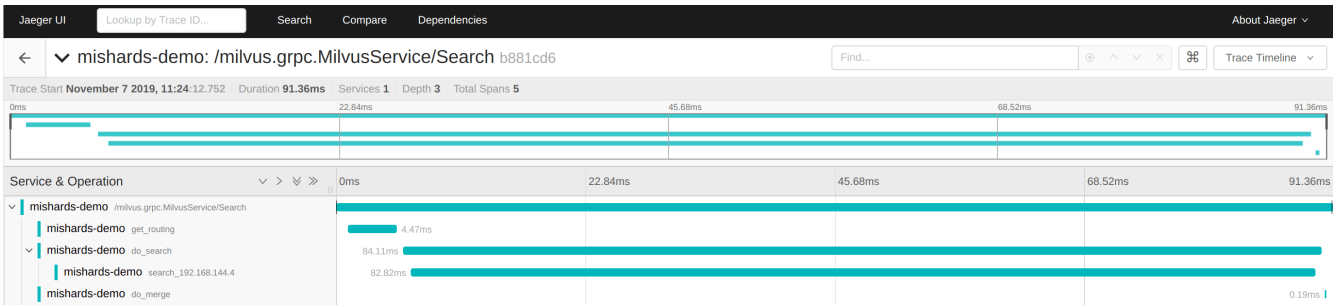

Figure 20: jaegerui

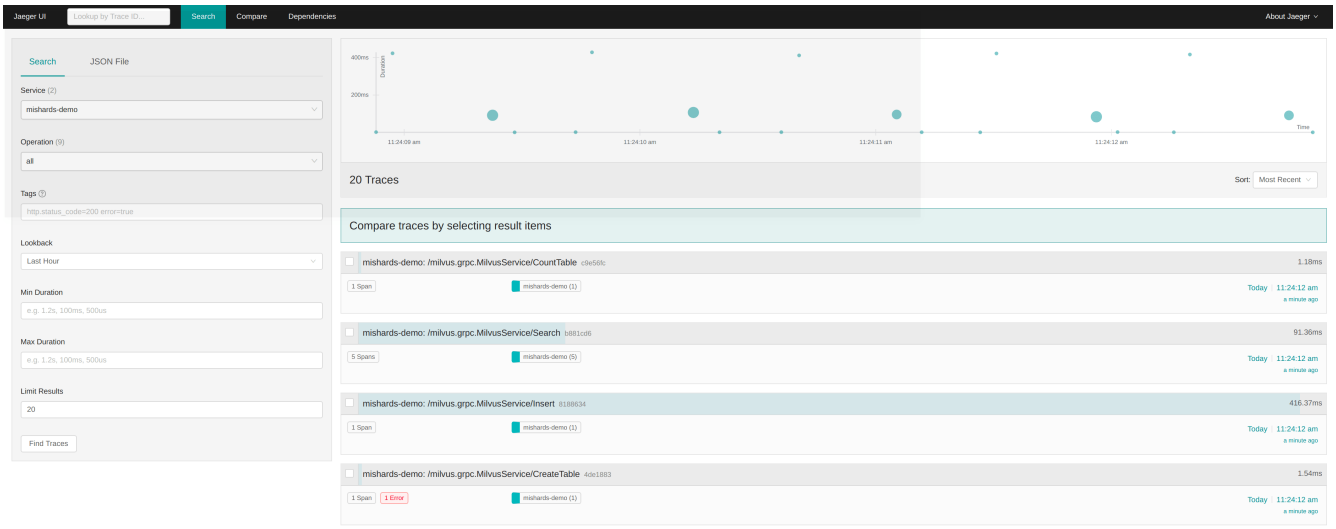

Figure 21: jaegertraces

在 Kubernetes 中部署 Mishards 集群

安装前提

- Kubernetes 版本 1.10 及以上
- Helm 版本 2.12.0 及以上

关于 Helm 的使用请参考 Helm 使用指南。

安装流程

1. 添加 Helm Chart 仓库：

```
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com
```

2. 安装 Chart 依赖：

```
$ git clone https://github.com/milvus-io/milvus-helm.git
$ cd milvus-helm/charts/milvus
$ helm dep update
```

3. 部署 Mishards：

```
$ helm install --set cluster.enabled=true --set persistence.enabled=true milvus-release  .
```

4. 检查部署状态：

```
$ helm list -f "milvus-release"
```

卸载 Mishards

- 使用 Helm v2.x 卸载 Mishards：

```
$ helm delete milvus-release
```

- 使用 Helm v3.x 卸载 Mishards：

```
$ helm uninstall milvus-release
```

从单机升级到 Mishards 集群

Milvus-Helm 支持从单机服务升级到 Mishards 集群。

1. 部署单机 Milvus：

```
$ helm install --set persistence.enabled=true milvus-release .
```

2. 升级到 Mishards 集群：

```
$ helm upgrade --set cluster.enabled=true --set persistence.enabled=true milvus-release .
```

注意事项

Mishards 依赖共享存储，因此 Kubernetes 集群中必须有可用的 PV（Persistent Volumes），并且保证 PV 能够同时被多个 Pod 共同使用。这里通过 `persistence.enabled` 来开启 Persistent Volumes。

1. 为了共享数据，PV 访问模式必须被设置为 `ReadOnlyMany` 或者 `ReadWriteMany`。
2. 选择文件存储系统：
   - 如果集群部署在 AWS，可以使用 Elastic File System (EFS)。
   - 如果集群部署在 Azure，可以使用 Azure File Storage (AFS)。

关于 PersistentVolume 的申请及管理，请参阅 Persistent Volumes。

关于 PersistentVolume 的访问模式，请参阅 Access Modes。

基本案例

你可以在 Milvus Helm Charts 找到 Milvus-Helm 支持的所有参数。

1. 配置有多个读节点和多个 Mishards 分片中间件的集群。

   我们通常配置多个节点来保证服务的可用性以及提高吞吐率。下面的例子部署的 Mishards 集群包括：2 个分片中间件、2 个读节点、1 个写节点。

   ```
   $ helm install
      --set cluster.enabled=true      \
      --set persistence.enabled=true  \
      --set mishards.replicas=2        \
      --set readonly.replicas=2        \
      milvus-release .
   ```

   这里通过 `mishards.replicas` 和 `readonly.replicas` 控制副本集数量，默认值为 1。

   Mishards 集群中的写节点暂时不支持扩展。

2. 使用外部已配置好的 MySQL 集群作为元数据数据库。

   为了配合本地部署，有些情况需要支持外部的 MySQL。Milvus-Helm 内部的 MySQL 服务不保证高可用，但你可以通过外部的 MySQL 集群来提升可用性。下面的例子就是基于外部 MySQL 的部署。

   ```
   $ helm install
      --set cluster.enabled=true             \
      --set persistence.enabled=true         \
      --set mysql.enabled=false              \
      --set externalMysql.enabled=true       \
      --set externalMysql.ip=192.168.1.xx    \
      --set externalMysql.port=3306          \
      --set externalMysql.user=root          \
      --set externalMysql.password=root      \
      --set externalMysql.database=milvus    \
      milvus-release .
   ```

   若使用外部的 MySQL，则不再需要 Helm 内置的 MySQL 服务。这里通过 `mysql.enabled` 来关闭 Helm 内置的 MySQL 服务。

3. 读写节点的 Milvus 分别使用不同的配置。

   为了更合理地使用资源，我们希望读节点和写节点拥有不一样的配置。下面的例子配置了一个拥有 16 GB 内存的读节点和 8 GB 内存的写节点。

   ```
   $ helm install
      --set cluster.enabled=true                    \
      --set persistence.enabled=true                \
      --set cache.cpuCacheCapacity=8                \
      --set readonly.cache.cpuCacheCapacity=16      \
      milvus-release .
   ```

   更多 Milvus 配置参数，请参阅 Milvus 服务端配置。

   更多 Milvus-Helm 配置参数，请参阅 Milvus Helm Charts。

4. 配置 GPU 资源。

   使用 GPU 可以有效提升 Milvus 的性能。下面的例子通过 `gpu.enabled=true` 允许写节点使用 GPU 资源，`readonly.gpu.enabled=false` 禁止读节点使用 GPU 资源。

   ```
   $ helm install
      --set cluster.enabled=true             \
      --set persistence.enabled=true         \
      --set gpu.enabled=true                 \
   ```

```
--set readonly.gpu.enabled=false        \
milvus-release .
```

Kubernetes 集群中必须有可访问的 GPU 资源。关于 Kubernetes 中的 GPU 资源管理及调度，请参阅 Schedule GPUs。

常见问题

性能优化问题

为什么重启 Milvus 服务端之后，第一次搜索时间非常长?

重启后第一次搜索时，会将数据从磁盘加载到内存，所以这个时间会比较长。可以在 server_config.yaml 中开启 `preload_collection`，在内存允许的情况下尽可能多地加载集合。这样在每次重启服务端之后，数据都会先载入到内存中，可以解决第一次搜索耗时很长的问题。或者在查询前，调用方法 `load_collection()` 将该集合加载到内存。

为什么搜索的速度非常慢?

请首先检查 server_config.yaml 的 `cache.cache_size` 参数是否大于集合中的数据量。

如何进行性能调优?

- 确保配置文件中的参数 `cache.cache_size` 值大于集合中的数据量。
- 确保所有数据文件都建立了索引。
- 检查服务器上是否有其他进程在占用 CPU 资源。
- 调整参数 `index_file_size` 和 `nlist` 的值。
- 如果检索性能不稳定，可在启动 Milvus 时添加参数 `-e OMP_NUM_THREADS=NUM`，其中 `NUM` 为 CPU 逻辑核数的 2/3。

详见 性能调优。

应如何设置 IVF 索引的 **`nlist`** 和 **`nprobe`** 参数?

IVF 索引的 nlist 值需要根据具体的使用情况去设置。一般来说，推荐值为 $4 \times \text{sqrt(n)}$，其中 n 为 segment 内的 entity 总量。

`nprobe` 的选取需要根据数据总量和实际场景在速度性能和准确率之间进行取舍。建议通过多次实验确定一个合理的值。

以下是使用公开测试数据集 sift50m 针对 `nlist` 和 `nprobe` 的一个测试。以索引类型 IVF_SQ8 为例，针对不同 `nlist`/`nprobe` 组合的搜索时间和召回率分别进行对比。

因 CPU 版 Milvus 和 GPU 版 Milvus 测试结果类似，此处仅展示基于 GPU 版 Milvus 测试的结果。

在本次测试中，`nlist` 和 `nprobe` 的值成比例增长，召回率随 `nlist`/`nprobe` 组合增长呈现上升的趋势。

在 `nlist` 为 4096 和 `nprobe` 为 128 时，速度性能最佳。

为什么有时候小的数据集查询时间反而更长?

如果数据文件的大小小于创建集合时 `index_file_size` 参数的值，Milvus 则不会为此数据文件构建索引。因此，小的数据集有可能查询时间会更长。你还可以调用 `create_index` 建立索引。

为什么查询时 GPU 一直空闲?

此时应该是在用 CPU 进行查询。如果要用 GPU 进行查询，需要在配置文件中将 gpu_search_threshold 的值设置为小于 nq (每次查询的向量条数) 。可以将 gpu_search_threshold 的值调整为期望开启 GPU 搜索的 nq 数。若 nq 小于该值，则用 CPU 查询，否则将使用 GPU 查询。不建议在查询批量较小时使用 GPU 搜索。

为什么数据插入后不能马上被搜索到?

因为数据还没有落盘。要确保数据插入后立刻能搜索到，可以手动调用 `flush` 接口。但是频繁调用 `flush` 接口可能会产生大量小数据文件，从而导致查询变慢。

为什么我的 CPU 利用率始终不高?

nq = 100 以下，且数据量也不大的时候确实会出现这个现象。Milvus 在计算时，批量内的查询是并行处理的，如果批量不大且数据量也不大的话，并行度不高，CPU 利用率也就不高了。

创建集合时 **index_file_size** 如何设置能达到性能最优?

使用客户端创建集合时有一个 **index_file_size** 参数，用来指定数据存储时单个文件的大小，其单位为 MB，默认值为 1024。当向量数据不断导入时，Milvus 会把数据增量式地合并成文件。当某个文件达到 **index_file_size** 所设置的值之后，这个文件就不再接受新的数据，Milvus 会把新的数据存成另外一个文件。这些都是原始向量数据文件，如果建立了索引，则每个原始文件会对应生成一个索引文件。Milvus 在进行搜索时，是依次对每个索引文件进行搜索。

根据我们的经验，当 **index_file_size** 从 1024 改为 2048 时，搜索性能会有 30% ~ 50% 左右的提升。但要注意如果该值设得过大，有可能导致大文件无法加载进显存（甚至内存）。比如显存只有 2 GB，该参数设为 3 GB，显存明显放不下。

如果向集合中导入数据的频率不高，建议将 **index_file_size** 的值设为 1024 MB 或者 2048 MB。如果后续会持续地向集合中导入增量数据，为了避免查询时未建立索引的数据文件过大，建议这种情况下将该值设置为 256 MB 或者 512 MB。

可参阅 如何设置 Milvus 客户端参数。

Milvus 的导入性能如何?

客户端和服务端在同一台物理机上时，10 万条 128 维的向量导入需要约 0.8 秒（基于 SSD 磁盘）。这个具体也要看磁盘的 I/O 速度。

边插入边搜索会影响搜索速度吗?

- 当插入向量没有达到建索引条件时，新插入向量在初次被搜索时需要从磁盘加载到内存。
- 当插入向量达到建索引条件时，Milvus 开始为新增向量创建索引。v0.9.0 之后，新出现的搜索请求会打断建索引任务，这需要 1 秒左右的延时。

批量搜索时，用多线程的收益大吗?

多线程查询，如果是小批量（nq < 64）的话，后台会合并查询请求。如果是大批量查询的话，就不会有什么优势。

为什么同样的数据量，用 GPU 查询比 CPU 查询慢?

一般来说，当 nq（每次查询的向量条数）较小时，用 CPU 查询比较快。只有当 nq 较大（约大于 500）时，用 GPU 查询才会更有优势。

因为在 Milvus 中，每次用 GPU 查询都需要将数据从内存加载到显存。只有当 GPU 查询节省的计算时间能抵消掉数据加载的时间，才能体现出 GPU 查询的优势。

仍有问题没有得到解答?

如果仍有其他问题，你可以：

- 在 GitHub 上访问 Milvus，提问、分享、交流，帮助其他用户。
- 加入我们的 Slack 社区，与其他用户讨论交流。

产品问题

Milvus 会收费吗?

Milvus 会坚持开源路线，现有的版本都不会收费。

项目开源，请遵循 Apache 2.0 协议 使用。

Milvus 支持非 x86 平台吗?

目前不支持。

Milvus 支持对向量的插入、删除、更改和查询操作吗?

支持。其中，对向量的修改可以通过先删除再插入来实现。

Milvus 可以处理百亿或千亿级数据吗?

Milvus 提供了集群分片中间件 Mishards，可以实现集群分片部署，满足百亿或者千亿级数据的处理需求。

Milvus 数据存储在哪里?

向量数据导入 Milvus 后，将自动存储在本地磁盘的 milvus/db/tables/ 这个路径下。

元数据可以存储在 MySQL 或 SQLite 上。详见 使用 MySQL 管理元数据。

为什么我在 SQLite / MySQL 找不到向量数据?

SQLite / MySQL 只是存放原始向量数据的元数据。向量和索引直接以文件的形式存在磁盘上，不存放在 SQLite 或 MySQL 里。详见 存储相关概念。

Milvus 的元数据存储可以使用 SQL Server 或者 PostgreSQL 吗?

不可以，目前仅支持 SQLite 和 MySQL。

Milvus 的 Python SDK 有连接池吗?

Milvus v0.9.0 及更高版本对应的 Python SDK 有连接池。连接池的默认连接数量没有上限。

Milvus 是否支持“边插入边查询”?

支持。

Milvus 是否有可视化管理工具?

从 Milvus v0.7.0 开始有 Milvus Enterprise Manager 提供图形管理功能。

Milvus 有没有数据导出的功能?

目前没有专门的工具实现该功能。你可以通过 `get_entity_by_id` 得到指定 ID 对应的向量。

为什么我通过 `get_entity_by_id` 获取的向量精度损失了?

Milvus 将向量的每个维度以单精度（精确到小数点后 7 位）存储和计算。所以如果原始数据为双精度（精确到小数点后 16 位），经过 Milvus 的处理后就会出现精度损失。

向 Milvus 中导入数据时，应该自己指定 entity ID 还是由 Milvus 自动生成 entity ID?

两种方法均可。但是，在一个集合内的向量必须全部使用用户指定的 entity ID 或者全部使用自动生成的 entity ID。

Milvus 可以插入重复 ID 的向量吗?

可以，这样在 Milvus 中会存在相同 ID 的多条向量。

Milvus 中自定义 ID 有没有长度限制?

ID 类型是非负的 64 位整型。

Milvus 中单次插入数据有上限吗?

单次插入数据不能超过 256 MB。

为什么向量距离计算方式是内积时，搜索出来的 **top1** 不是目标向量本身?

向量距离计算方式用内积时，如果向量未归一化，会出现这样的情况。

对集合分区的查询是否会受到集合大小的影响，尤其在集合数据量高达一亿数据量时?

不会。如果你在搜索时指定了分区，Milvus 只会在相应分区进行搜索。

如果只是搜索集合中的部分分区，整个集合的数据会全部加载到内存吗?

不会，只加载指定的分区里的数据。

对各个数据段的检索是并行处理的吗?

一般而言，Milvus 对单个数据段内的查询是并行的，多个数据段的处理根据发行版本略有不同。

假设一个集合存在多个数据段，当查询请求到达时：

- CPU 版 Milvus 会对数据段读取任务和段内查询任务进行流水线处理。
- GPU 版 Milvus 会在 CPU 版的基础上，将多个数据段分配给各个 GPU 处理。

可参阅文章：Milvus 开源向量数据库 ANNS。

Milvus 中如何选择向量索引的类型?

索引需要根据具体场景和需求去选择。详见 如何选择索引类型。

Milvus 可以在同一个集合中的不同分区上建立不同索引吗?

不可以。同一个集合在某一刻只能有一种索引。

Milvus 中支持新增向量后再建索引吗?

支持。Milvus 中数据是分文件存储的，后续新增向量会存在新的数据文件中。该文件达到一定量后会自动触发建立索引，生成一个新的索引文件，不会影响之前已经建立过的索引。

索引 IVF_SQ8 和 IVF_SQ8H 在召回率上有区别吗?

对于相同的数据集，IVF_SQ8 和 IVF_SQ8H 的召回率一致。

Milvus 中 FLAT 索引和 IVF_FLAT 索引的原理比较?

把 FLAT 和 IVF_FLAT 做比较，可以这么估算：

已知 IVF_FLAT 索引是把向量分成 **nlist** 个单元。假设用默认的 **nlist** = 16384，搜索的时候是先用目标向量和这 16384 个中心点计算距离，得到最近的 **nprobe** 个单元，再在单元里计算最近向量。而 FLAT 是每条向量和目标向量计算距离。

所以当总的向量条数约等于 **nlist** 时，两者的计算量相当，性能也差不多。而随着向量条数达到 **nlist** 的 2 倍、3 倍、n 倍之后，IVF_FLAT 的优势就越来越大。

可参阅 如何选择索引类型。

创建索引立即查询，为什么内存会突然增长?

这是因为 Milvus 在进行搜索时会将新生成的索引文件加载到内存，由于加载的索引文件和用于生成索引文件的原始向量文件总和小于 **cache.cache_size** 的上限，原始向量数据暂未被系统从内存释放。

建立集合后，**index_file_size** 和 **metric_type** 还支持修改吗?

不支持。

向量插入后自动落盘间隔时间是多少?

1 秒。

如果设置了 `preload_collection`，必须是等集合全部加载到内存，服务才能访问吗?

是的。如果是在 server_config.yaml 里设置的，那在启动 Milvus 时就会加载好数据后才开始提供服务。

Milvus 的数据落盘逻辑是怎样的?

插入时把数据写到内存，定时地把缓存里的数据落盘。如果调用 `flush` 方法，也会触发落盘的动作。

详见 存储操作 > 数据落盘。

Mishards 推荐的配置是什么?

推荐写节点用 GPU 版 Milvus，读节点用 CPU 版 Milvus。比如现在只能用单个写节点，这个写节点可以配置 GPU 资源用来建索引，读节点都配置成 CPU 节点。

Mishards 支持 RESTful API 吗?

目前不支持。

什么是归一化? Milvus 中为什么有时候需要归一化?

归一化指的是通过数学变换将向量的模长变为 1 的过程。如需使用点积计算向量相似度，则必须对向量作归一化处理。处理后点积与余弦相似度等价。

可参阅文章 向量搜索的简明数学基础。

为什么欧氏距离和内积在计算向量相似度时的结果不一致?

如果欧氏距离和内积返回不一致的结果，需要检查数据是否已经归一化。如果没有，请先对数据进行归一化。理论上可以证明，对于未归一化的数据，欧氏距离和内积的结果是不一致的。

Milvus 对集合和分区的总数有限制吗?

collection 数量没有限制。每个 collection 内的 partition 总数不能超过 4096 个。

为什么搜索 `topk` 向量，结果不到 k 条向量?

在 Milvus 支持的索引类型中，IVF_FLAT 和 IVF_SQ8 是基于 k-means 空间划分的分单元搜索算法。空间被分为 `nlist` 个单元，导入的向量被分配存储在基于 `nlist` 划分的文件结构中。搜索发生时，只搜索最近似的 `nprobe` 个单元。

如果 `nlist` 和 k 比较大，而 `nprobe` 又足够小，有可能出现 `nprobe` 文件中的所有向量总数小于 k。当搜索 `topk` 向量时，就会出现搜索结果小于 k 条向量的情况。

想要避免这种情况，可以尝试将 `nprobe` 设置为更大值，或者把 `nlist` 和 `k` 设置为更小值。

详见 索引类型。

Milvus 支持的向量维度的最大值是多少?

Milvus 最多能够支持 32,768 向量维度。

为什么 Milvus 对单次可插入的数据大小设置了 256MB 的上限?

插入 Milvus 中的数据会被先写入内存。这个限制是为了避免插入操作过度占用内存资源。

Milvus 会批量合并多线程查询请求。如何取消这个机制?

你可以将以下内容添加至配置文件 server_config.yaml 中:

```
engine_config:
  search_combine_nq: 1
```

仍有问题没有得到解答?

如果仍有其他问题，你可以:

- 在 GitHub 上访问 Milvus，提问、分享、交流，帮助其他用户。
- 加入我们的 Slack 社区，与其他用户讨论交流。

部署运维问题

如果在安装 Milvus 时，从 Docker Hub 拉取镜像总是失败怎么办?

某些地区的用户可能无法快速访问 Docker Hub。如果拉取镜像失败，可以从其它的镜像源拉取镜像。比如中国镜像源的网址为 registry.docker-cn.com。可以在 /etc/docker/daemon.json 文件的 `registry-mirrors` 组添加 `"https://registry.docker-cn.com"` 命令，这样就可以默认从中国镜像源拉取镜像了。

```
{
  "registry-mirrors": ["https://registry.docker-cn.com"]
}
```

Milvus 只能使用 Docker 部署吗?

Milvus 还支持源码编译，该方法仅支持 Linux 系统。详见 从源代码编译 Milvus。

为什么 Milvus 返回 **`config check error`** 的错误?

Milvus 和服务端配置文件的版本不对应。

为什么在导入数据时 Milvus 显示 **`no space left on device`** 的错误?

可能是没有为导入数据预留足够的磁盘空间。

为什么 Milvus 查询召回率一直不理想?

在调用 SDK 进行向量搜索时，可以增大函数中 **`nprobe`** 参数的值。值越大，结果越精确，但耗时也越久。详见 如何设置 Milvus 客户端参数。

为什么更新过的设置没有生效?

每次更新配置文件之后必须重启 Milvus Docker 才能让改动生效。详见 服务端配置 > 配置修改。

如何得知我的 Milvus 已经成功启动?

使用 `sudo docker logs <container ID>` 命令检查 Milvus 的运行状态。

为什么我的日志文件时间与系统时间不一致?

Docker 镜像内部的日志文件默认使用 UTC 时间。如果宿主机未使用 UTC 时间，就会出现日志文件时间与系统时间不一致的情况。建议在宿主机上挂载日志文件，这样可以保证宿主机上的日志文件和系统时间是一致的。

如何确认 Milvus 是否支持我的 CPU?

目前，Milvus 支持的指令集有：SSE42、AVX、AVX2、AVX512。你的 CPU 必须支持其中任意一个指令集才能保证 Milvus 正常工作。

为什么 Milvus 在启动时返回 `Illegal instruction`?

如果你的 CPU 对于 SSE42、AVX、AVX2、AVX512 这四种指令集都不支持，则 Milvus 在启动时会返回上述报错信息。可以通过 `cat /proc/cpuinfo` 查看 CPU 支持的指令集。

如何确认 Milvus 是否支持我的 GPU?

Milvus 支持 CUDA 6.0 架构以后的显卡。关于 Milvus 支持的架构，详见 Wikipedia。

Milvus 镜像里面启动 server 的脚本在哪?

启动 server 的脚本在 Milvus 容器内的 /var/lib/milvus/script/ 路径下。

除了配置文件外，怎样可以判断我确实在使用 GPU 做 search?

有以下三种方式:

- 使用 `nvidia-smi` 命令查看 GPU 使用情况。
- 用 Prometheus 配置，详见 使用 Grafana 展示监控指标 > 系统运行指标。
- 使用 Milvus 服务端的日志。

可以在 Windows 上安装 Milvus 吗?

理论上只要能够支持 Docker 的操作系统都可以运行 Milvus。

在 Windows 安装 pymilvus 报错，如何解决?

可以尝试在 Conda 环境下安装。

内网环境，即离线方式，能否部署 Milvus 服务?

Milvus 是以 Docker 镜像形式发行的，是可以离线部署的:

1. 在有网的环境中拉取最新的 Milvus 镜像;
2. 使用 `docker save` 将镜像保存为 TAR 文件;
3. 拷贝该镜像到无网的环境中;
4. 用 `docker load` 命令导入该镜像。

关于 Docker 的使用详见 docs.docker.com。

在多个 Milvus 节点接入 Pushgateway 的情况下如何进行区分数据来源?

在 prometheus.yaml 里面加一个 Prometheus 的实例就可以。最后在 Prometheus 或者 Grafana 里面显示监控的时候，会指明数据是来自哪个 Milvus 实例。

我应该使用 SQLite 还是 MySQL 进行元数据管理?

生产环境下，推荐使用 MySQL 。

如何根据数据量计算需要多大的内存?

不同的索引所需内存不同。可以使用 Milvus 的 sizing 工具 去计算查询时所需要的内存。

Milvus 中如何实现数据迁移?

把原有的 Milvus 服务的整个 db 目录拷贝到新的路径下，启动新的 Milvus 服务时，将该 Milvus 服务的 db 目录映射为刚拷贝过来的 db 目录。> 注意：不同版本之间，数据可能会不兼容。目前数据格式兼容到 0.7.0。

Milvus 可以通过扩展某些接口 (如 S3 接口或 GlusterFS 接口) 来扩展存储吗?

目前暂不支持。

Milvus 日志中为什么会出现这个警告 **WARN: increase temp memory to avoid cudaMalloc, or decrease query/add size (alloc 307200000 B, highwater 0 B)**?

在 Milvus 中，如果单次申请的显存量大于它预先开辟的一段显存空间，就会报这个警告。不过没有影响，Milvus 中会扩大它使用的显存空间来满足这个显存的申请。这个警告的意思就是要使用更多显存空间了。

出现 **database is locked** 的报错怎么解决?

如果元数据管理用的是 SQLite，在有数据频繁写入的情况下会出现该错误。建议将 SQLite 更换为 MySQL。如何更换请参考文档 使用 MySQL 管理元数据。

PyMilvus 返回 **Segmentation Fault** 错误，如何解决?

PyMilvus v1.1.0 允许下载最新版本的 grpcio library，而 Milvus v1.x 的服务端则基于早期的定制版本 gRPC 搭建，该版本 gRPC 与最新版本 grpcio library 不兼容。解决此问题需将 PyMilvus 升级至 v1.1.1 或将 grpcio library 回滚至 1.37.0:

```
pip install grpcio==1.37.0
pip install grpcio-tools==1.37.0
```

是否可以指定 **Milvus()** 接口的超时时间?

PyMilvus v1.1.2 已支持指定服务器连接超时时间。

如何为建立索引和检索分配 GPU?

在 /home/$USER/milvus/conf 路径下的 server_config.yaml 文件中，你可以为建立索引以及检索分配 GPU。如下代码所示:

```
gpu:
  enable: true
  cache_size: 10GB
  gpu_search_threshold: 0
  search_devices:
    - gpu0
    - gpu1
  build_index_devices:
    - gpu2
    - gpu3
```

gpu 0, 1, 2, 3 是分配给 Docker 容器的 GPU 中的前四个。

仍有问题没有得到解答?

如果仍有其他问题，你可以:

- 在 GitHub 上访问 Milvus，提问、分享、交流，帮助其他用户。
- 加入我们的 Slack 社区，与其他用户讨论交流。

故障诊断

该页面主要描述使用 Milvus 时会遇到的常见问题。这些问题主要分为以下几类:

服务启动问题

服务启动时发生的故障，通常会导致服务无法正常启动。你可以通过以下命令来查看相关错误信息:

```
$ docker logs <milvus container id>
```

服务运行问题

服务运行期间发生的故障有可能导致服务瘫痪。遇到此类故障时，请先检查系统版本和所使用的客户端版本是否兼容，然后再查询相关错误信息。服务运行的错误信息将被记录在 /home/$USER/milvus/logs 文件中。

API 问题

通过 API 使用 Milvus 时发生的故障。这类错误信息将实时返回给客户端。

对于你无法自己解决的问题，你可以：

- 加入我们的 Slack 社区，提问并与 Milvus 团队及其它社区成员交流讨论。
- 在 GitHub 上 创建 issue，详细描述你的问题。

API 参考

开发工具

概述

开发者工具概述

你可以通过以下工具优化你的 Milvus 体验：

- Milvus Enterprise Manager: Milvus 服务端图形化交互与管理工具
- Milvus Sizing Tool: 硬件配置估算工具
- MilvusDM：Milvus 数据迁移工具 ## MilvusDM 数据迁移工具

Milvus 数据迁移工具 MilvusDM

简介

MilvusDM 是一款针对 Milvus 研发的数据迁移工具，支持 Milvus 数据传输以及数据文件的导入与导出：

- Faiss to Milvus：将未压缩的 Faiss 文件导入 Milvus
- HDF5 to Milvus：将 HDF5 格式的文件导入 Milvus
- Milvus to Milvus：支持 Milvus 之间的数据迁移
- Milvus to HDF5：将 Milvus 数据批量备份为 HDF5 格式的本地文件

开发者使用 MilvusDM 可以提升数据管理效率，降低运维成本。

功能介绍

在之前发布的文章《Milvus 迁移升级攻略》中，我们介绍了如何将 Milvus 数据从有网环境迁移到无网环境以及不同版本 Milvus 之间的数据迁移。数据迁移工具 MilvusDM 可以通过指定 Milvus 中的集合或分区，帮助用户更智能地迁移所需数据。MilvusDM 十分易于使用，只需要运行一句 `pip3 install pymilvusdm` 指令即可快速安装。此外，您还可在 github 上获取本项目的开源代码。本文将介绍 MilvusDM 工具的使用方法：

Faiss to Milvus

- 使用示例

  1. 下载 yaml 文件

  ```
  $ wget https://raw.githubusercontent.com/milvus-io/milvus-tools/main/yamls/F2M.yaml
  ```

  2. 配置参数

  通过指定文件路径 `data_path` 读取 Faiss 的数据，并将读取到的向量和 ids 导入 Milvus 中。导入时需要指定参数 `dest_host`、`dest_port`、`mode`、`dest_collection_name`、`dest_partition_name` 和 `collection_parameter`。

  ```
  F2M:
  milvus_version: 1.1.1
  data_path: '/home/data/faiss.index'
  dest_host: '127.0.0.1'
  dest_port: 19530
  ```

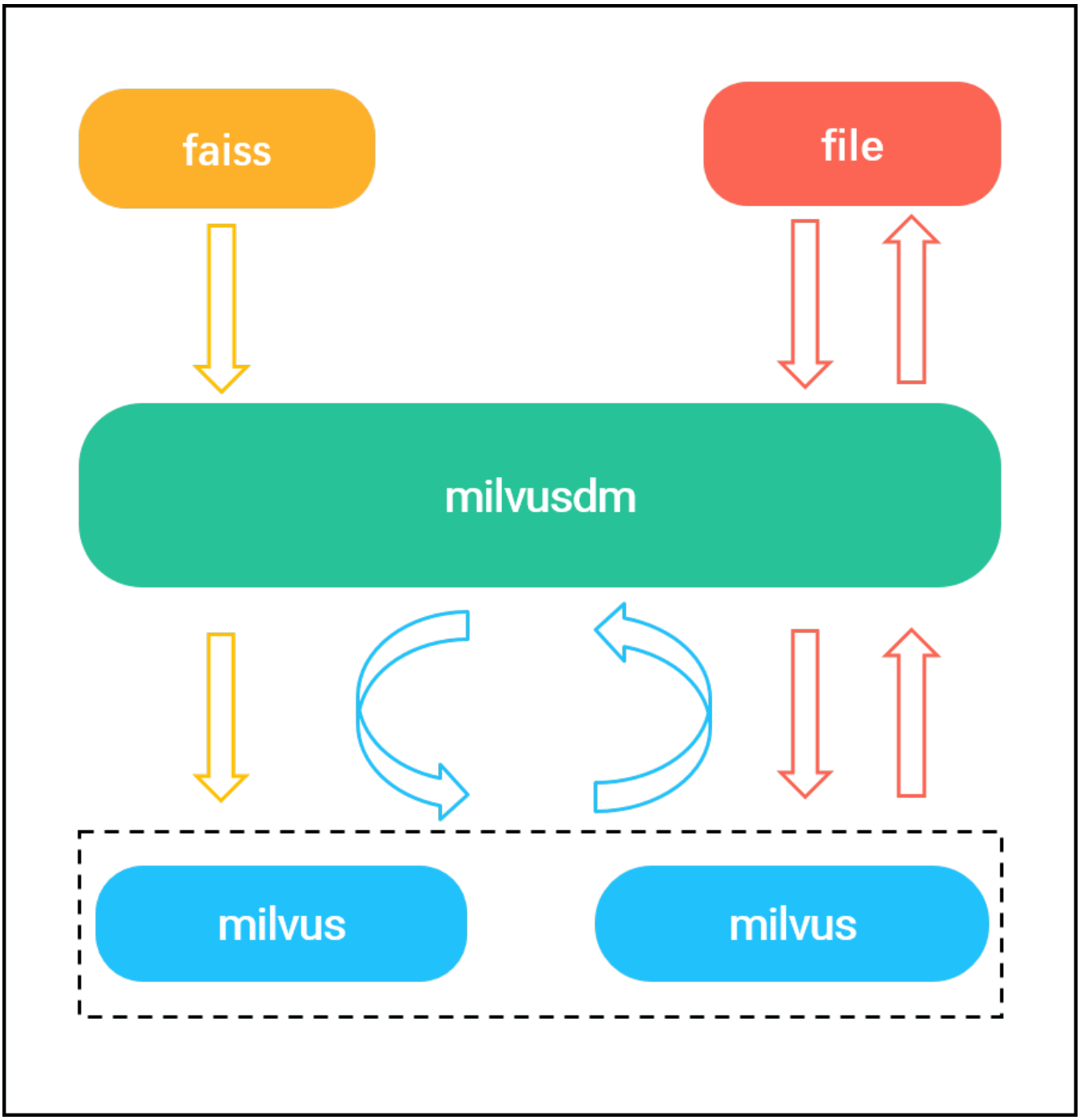

Figure 22: milvusdm blog 1.png

```
mode: 'append'
dest_collection_name: 'test'
dest_partition_name: ''
collection_parameter:
dimension: 256
index_file_size: 1024
metric_type: 'L2'
```

3. 运行

```
$ milvusdm --yaml F2M.yaml
```

- 具体实现

读取 Faiss 的文件，返回特征向量和对应的 ids 并导入 Milvus。

```
ids, vectors = faiss_data.read_faiss_data()
insert_milvus.insert_data(vectors, self.dest_collection_name, self.collection_parameter, self.mode, ids, s
```

HDF5 to Milvus

- 使用示例

1. 下载 yaml 文件

```
$ wget https://raw.githubusercontent.com/milvus-io/milvus-tools/main/yamls/H2M.yaml
```

2. 配置参数

通过指定 `data_path` 或 `data_dir` 读取 HDF5 格式的数据并导入 Milvus 中。导入时需要指定参数 `dest_host`、`dest_port`、`mode`、`dest_collection_name`、`dest_partition_name` 和 `collection_parameter`。

`data_path` 参数可指定多个文件路径，`data_dir` 参数指定文件目录。两个参数只能配置一个。

```
H2M:
milvus-version: 1.1.1
data_path:
- /Users/zilliz/float_1.h5
- /Users/zilliz/float_2.h5
data_dir:
dest_host: '127.0.0.1'
dest_port: 19530
mode: 'overwrite'        ### 'skip/append/overwrite'
dest_collection_name: 'test_float'
dest_partition_name: 'partition_1'
collection_parameter:
dimension: 128
index_file_size: 1024
metric_type: 'L2'
```

3. 运行

```
$ milvusdm --yaml H2M.yaml
```

- 具体实现

读取 HDF5 格式的文件，返回特征向量和对应的 ids 并导入 Milvus。

```
embeddings, ids = self.file.read_hdf5_data()
ids = insert_milvus.insert_data(embeddings, self.c_name, self.c_param, self.mode, ids,self.p_name)
```

Milvus to Milvus

- Improve 使用示例

1. 下载 yaml 文件

```
$ wget https://raw.githubusercontent.com/milvus-io/milvus-tools/main/yamls/M2M.yaml
```

2. 配置参数

通过指定 source_milvus_path、mysql_parameter 和 source_collection 读取源 Milvus 的向量和 ids 数据并导入 Milvus 中。导入时需要指定参数 dest_host、dest_port 和 mode。

如果源 Milvus 没有使用 MySQL 进行元数据管理，mysql_parameter 参数则为空。

```
M2M:
milvus_version: 1.1.1
source_milvus_path: '/home/user/milvus'
mysql_parameter:
host: '127.0.0.1'
user: 'root'
port: 3306
password: '123456'
database: 'milvus'
source_collection:
test:
  - 'partition_1'
  - 'partition_2'
dest_host: '127.0.0.1'
dest_port: 19530
mode: 'skip' ### 'skip/append/overwrite'
```

3. 运行

```
$ milvusdm --yaml M2M.yaml
```

- 具体实现

读取指定集合或分区的 meta 信息，根据 meta 信息读取本地 milvus/db 下的数据文件，返回特征向量和对应的 ids 并导入 Milvus。

```
collection_parameter, _ = milvus_meta.get_collection_info(collection_name)
r_vectors, r_ids, r_rows = milvusdb.read_milvus_file(self.milvus_meta, collection_name, partition_tag)
milvus_insert.insert_data(r_vectors, collection_name, collection_parameter, self.mode, r_ids, partitio
```

Milvus to HDF5

- 使用示例

1. 下载 yaml 文件

```
$ wget https://raw.githubusercontent.com/milvus-io/milvus-tools/main/yamls/M2H.yaml
```

2. 修改参数

通过指定 source_milvus_path、mysql_parameter 和 source_collection 读取源 Milvus 的数据，将 HDF5 格式的向量和 ids 保存在 data_dir 路径下。

```
M2H:
milvus_version: 1.1.1
source_milvus_path: '/home/user/milvus'
mysql_parameter:
host: '127.0.0.1'
user: 'root'
port: 3306
password: '123456'
database: 'milvus'
source_collection: ### specify the 'partition_1' and 'partition_2' partitions of the 'test' collectio
```

```
test:
  - 'partition_1'
  - 'partition_2'
data_dir: '/home/user/data'
```

3. 运行

```
$ milvusdm --yaml M2H.yaml
```

- 具体实现

  读取指定集合或分区的 meta 信息，根据 meta 信息读取本地 milvus/db 下的数据文件，返回特征向量和对应的 ids 并存入本地的 HDF5 文件中。

```
collection_parameter, version = milvus_meta.get_collection_info(collection_name)
r_vectors, r_ids, r_rows = milvusdb.read_milvus_file(self.milvus_meta, collection_name, partition_tag)
data_save.save_yaml(collection_name, partition_tag, collection_parameter, version, save_hdf5_name)
```

MilvusDM 代码结构

我们十分欢迎大家为开源项目 MilvusDM 贡献代码。你可以通过代码文件结构了解 MilvusDM 工具的设计构思。如有新的数据迁移需求，你还可以通过修改源码，为社区贡献代码。

MilvusDM 项目地址：milvus-io/milvus-tools

欢迎贡献代码 ✍，也请给本项目点 star ☆

使用 MilvusDM 时会根据传入的 yaml 文件执行对应的任务，如下图所示：

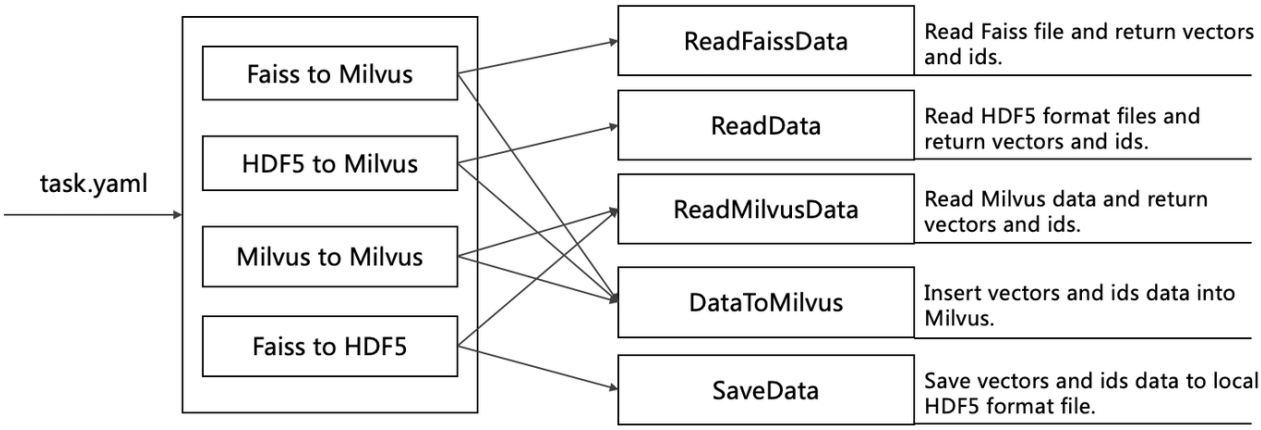

Figure 23: milvusdm blog 2.png

- pymilvusdm
  - core
    - milvus_client.py，Milvus 客户端相关的操作
    - read_data.py，读取本地 HDF5 格式的数据文件（如果有读取其他文件格式的需求，可在此处添加代码）
    - read_faiss_data.py，读取 Faiss 的数据文件
    - read_milvus_data.py，读取 Milvus 的数据文件
    - read_milvus_meta.py，读取 Milvus 的 meta 信息
    - data_to_milvus.py，根据 yaml 文件配置参数，建立集合或分区，并将向量和 ids 导入 Milvus
    - save_data.py，将读取到的数据保存为 HDF5 格式的文件

        * write_logs.py，在执行操作时写 debug/info/error 日志
    - faiss_to_milvus.py，实现将 Faiss 文件数据导入 Milvus
    - hdf5_to_milvus.py，实现将 HDF5 格式的文件数据导入 Milvus
    - milvus_to_milvus.py，实现将 Milvus 的数据拷贝到另一个 Milvus
    - milvus_to_hdf5.py，实现将 Milvus 的数据导出为 HDF5 格式的文件
    - main.py，根据 yaml 文件执行相关任务
    - setting.py，执行代码时的相关配置参数
- setup.py，将 pymilvusdm 打包并上传到 pypi

写在最后

本文介绍了 MilvusDM 工具的使用方法和开源代码。MilvusDM 主要支持以下四个功能：Faiss to Milvus、HDF5 to Milvus、Milvus to Milvus、Milvus to HDF5。如果有任何问题和建议，欢迎给本项目提 issue 或贡献代码。我们计划在下个版本中添加以下功能：

- 支持将 Faiss 的 binary 数据文件导入 Milvus
- Milvus to Milvus 时支持指定黑白名单
- Milvus to Milvus 时支持将多个集合或分区的数据合并导入至一个集合中
- 支持 Milvus 数据备份和数据恢复

如果你对本项目感兴趣的话欢迎加入我们。

发版说明

v1.1.1

发布时间：2021-06-18

版本兼容

Milvus 版本	Python SDK 版本	Java SDK 版本	Go SDK 版本
1.1.1	1.1.0	1.1.1	1.1.0

新增功能

- #1434 支持 S3 存储（由紫光华智实现）。
- #5142 支持在显存中缓存索引。

问题修复

- #4897 已删除 entity 在后继查询中仍然能被查到。
- #5164 在不存在的 partition 上插入或删除 entity 未引发异常报错。
- #5191 一段时间内连续执行多次插入和查询操作后，Mishards 会返回 `index out of range` 错误。
- #5398 系统执行请求后不定时崩溃。
- #5537 电源突然关闭后重启无法加载布隆过滤器。
- #5574 不能在多个 GPU 上同时构建 IVF_SQ8 和 IVF_PQ 索引。
- #5747 查询时 `nq` 或 `TopK` 乘积过大会导致 Milvus 停止响应。

主要改进

- #5115 将使用 CPU 查询的 `TopK` 上限从 16384 提高至 1048576。
- #5204 在未删除任何 entity 的情况下，优化了 IVF 索引在 GPU 版 Milvus 上的查询性能。
- #5544 将 `index_file_size` 上限提高至 128 GB。

v1.1.0

发布时间：2021-05-07

版本兼容

Milvus 版本	Python SDK 版本	Java SDK 版本	Go SDK 版本
1.1.0	1.1.0	1.1.0	1.1.0

新增功能

- #4564 支持在调用 `get_entity_by_id()` 方法时指定分区。
- #4806 支持在调用 `delete_entity_by_id()` 方法时指定分区。
- #4905 添加 `release_collection()` 方法以从缓存中卸载集合。

主要改进

- #4756 优化了 `get_entity_by_id()` 方法调用的性能。
- #4856 将 hnswlib 升级至 v0.5.0。
- #4958 优化了 IVF 索引训练的性能。

问题修复

- #4778 Mishards 中无法读取向量索引。
- #4797 合并不同 `topK` 查询返回结果错误。
- #4838 在空白集合中构建索引时，服务器未立即响应。
- #4858 在 GPU 版 Milvus 上执行大于 2048 的 `topK` 查询请求时系统崩溃退出。
- #4862 只读节点在启动时合并数据段。
- #4894 布隆过滤器的容量与所在数据段的行数不一致。
- #4908 删除集合后未清除 GPU 缓存。
- #4933 Milvus 为小数据段建索引耗时长。
- #4952 无法设置时区为”UTC + 5:30”。
- #5008 连续同时执行删除、插入和搜索操作时，Milvus 不定时崩溃退出。
- #5010 `nbits` 不等于 8 时，IVF_PQ 无法在 GPU 版 Milvus 上查询。
- #5050 对于未完成索引构建的数据段，`get_collection_stats()` 返回错误的索引类型。
- #5063 空白数据落盘导致 Milvus 崩溃退出。
- #5078 在 GPU 版 Milvus 上为 2048、4096 或 8192 维向量构建 IVF 索引时系统崩溃。

v1.0.0

发布时间：2021-03-09

版本兼容

Milvus 版本	Python SDK 版本	Java SDK 版本	Go SDK 版本
1.0.0	1.0.x	1.0.x	1.0.x

新增功能

- 支持将日志写入 stdout。#3977

主要改进

- 减小了 C++ SDK 使用的 grpc-milvus 的包体积。#4754

问题修复

- 修复了建索引或查询操作时产生的内存泄漏问题。#4749、#4757、#4765、#4766
    - 更多有关新增功能的信息，详见 Milvus 1.0 新增功能。
    - 关于产品路线图，详见 Milvus 1.0 产品路线图。