数据库构建、串并行查询对比

定义

1. data向量：存在数据库中的向量，
2. query向量：搜索的向量
3. target向量：返回的k-近邻，是data向量的子集

数据库查询
data向量传入数据库——>聚类，建立ivfflat索引——>逐次输入query查询并返回回target

串行查询：
逐次输入query向量：query和聚类中心比较选出待筛选聚类——>将对应聚类的所有向量轮流和query比较——>得到query的target

并行查询（暂定）：
一次输入所有query_batch，分query_batch做查询：query_batch和聚类中心比较选出待筛选聚类——>将对应聚类的所有向量轮流传上gpu，进行比较——>得到query_batch的target

聚类中心始终常驻显存（暂定，测试集肯定能这么跑）

测试数据集

Dataset	Dimensions	Train size	Test size	Neighbors	Distance	Download
DEEP1B	96	9,990,000	10,000	100	Angular	HDF5 (3.6GB)
SIFT	128	1,000,000	10,000	100	Euclidean	HDF5 (501MB)
TEXT	-	-	-	-	-	ann-benchmarks上没有支持，需要添加

数据集不算大，可以常驻内存。如果测试集很大，要搞磁盘——>内存——>GPU，可能会更费事一些

可行性分析

假设按照ann-benchmarks中的最高标准：一个形状为[N,D]的向量，N为1M级别，D为1K级别，则一个向量4KB

我们采用如下策略：

- 1K个聚类中心常驻显存(4MB)
- 10K个query输入后常驻显存(40MB)

测试框架中，一个批次是静态的、固定的，我们知道任意一个聚类要和哪些query计算距离。所以，数据总量约等于数据集大小，所以：

- 每个cluster只输入一次，(1M vectors占4GB) \
- **总共约占显存4GB**，（甚至可以让数据集常驻显存，但这显然不可推广的做法） \

鉴于memory-bound，我们将总的数据传输时间当作总查询时间：

- 如果1s内完成的话(吞吐量10^4)，我们**只需要将双向数据带宽达到8GB/s**就可以，而我们的PCIE总线带宽有32GB。

这样的估算有如下问题：

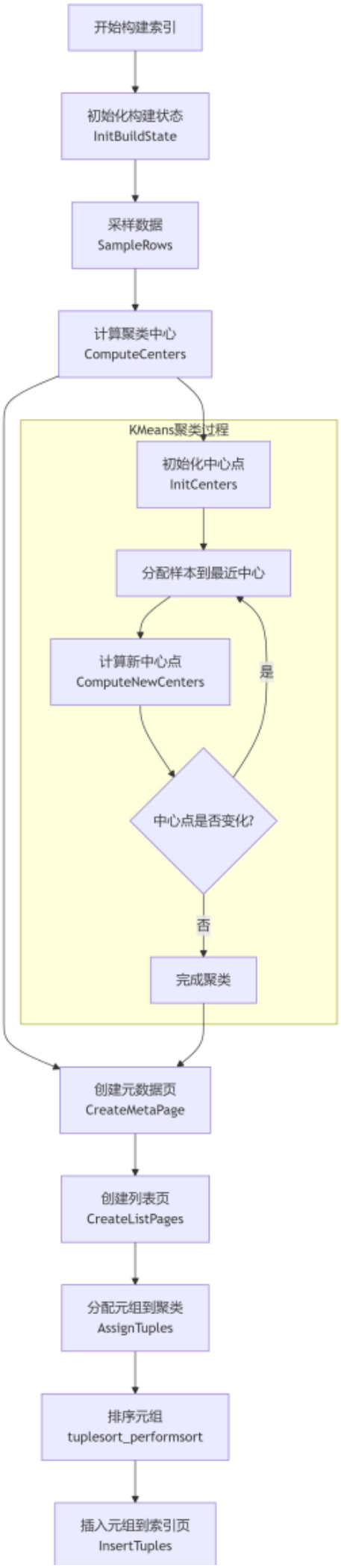
- 没有考虑cpu处理时间，仅按照数据流量估计（较轻的高估）
- 仅考虑一个cuda流，实际上面我们每次只用了几百MB的显存，单卡V100有16G显存（较严重的低估）

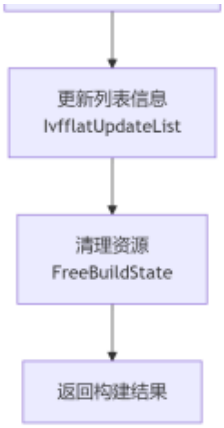
结合其他gpu向量数据库的表现，我认为**我们可以在实现尽可能简单的情况，实现合同目标**

代码结构

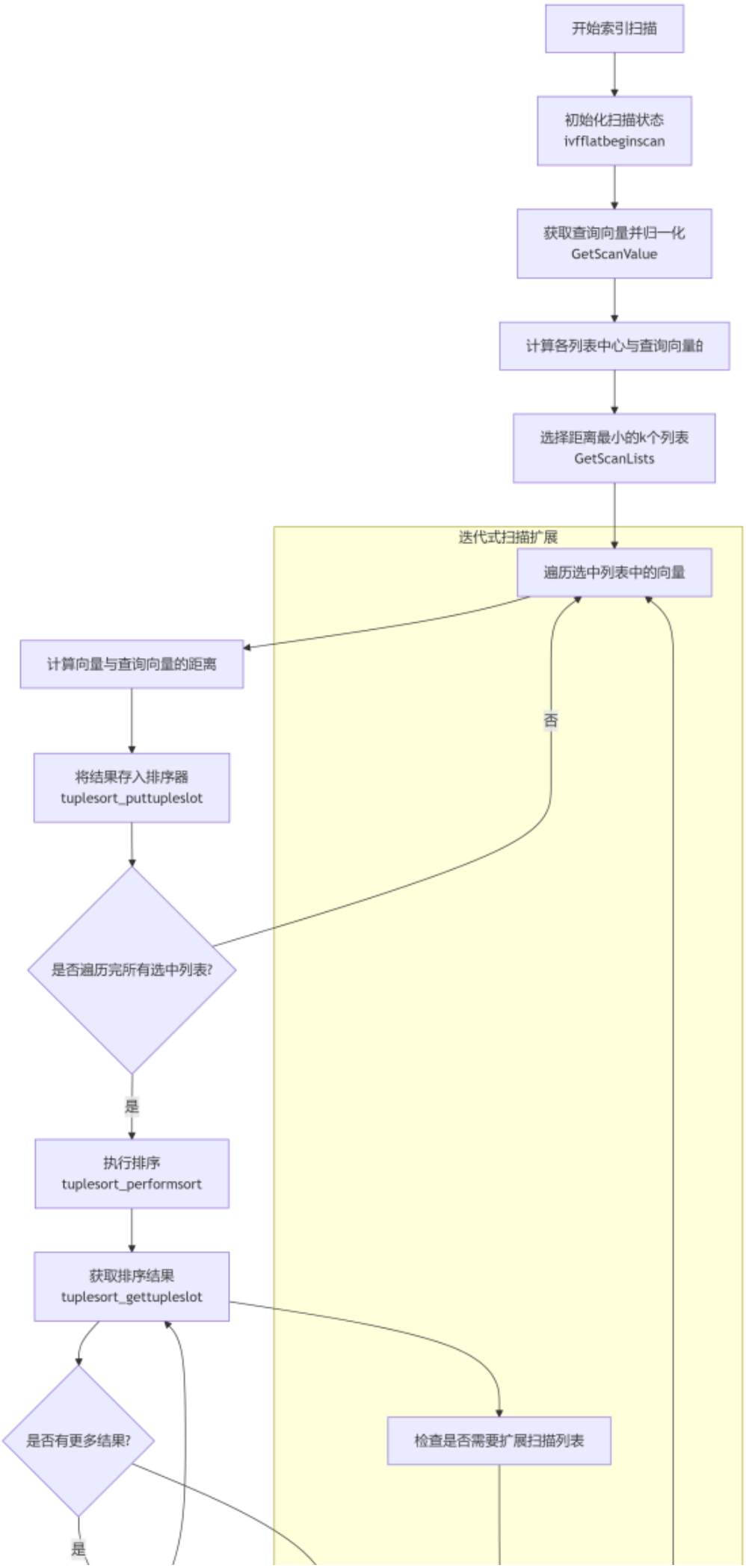
项目结构 by 章毅

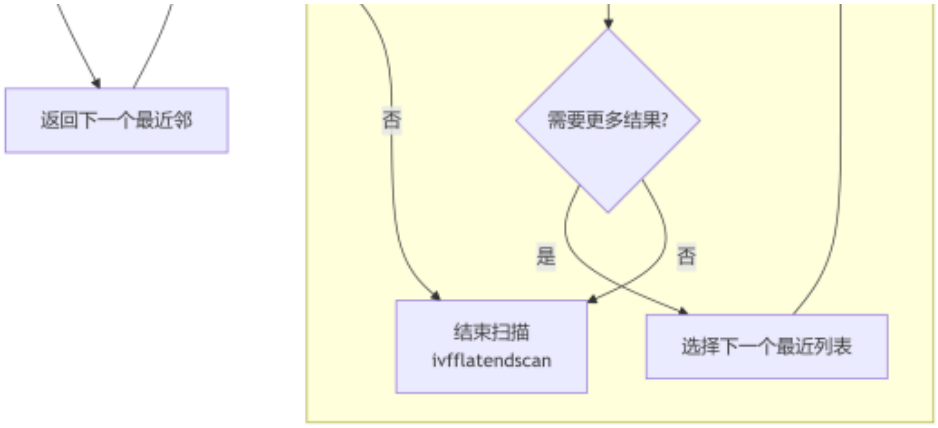
聚类流程图：


开始构建索引
初始化构建状态
InitBuildState
采样数据
SampleRows
计算聚类中心
ComputeCenters
KMeans聚类过程
初始化中心点
InitCenters
分配样本到最近中心
计算新中心点
ComputeNewCenters
是
中心点是否变化?
否
完成聚类
创建元数据页
CreateMetaPage
创建列表页
CreateListPages
分配元组到聚类
AssignTuples
排序元组
tuplesort_performsort
插入元组到索引页
InsertTuples

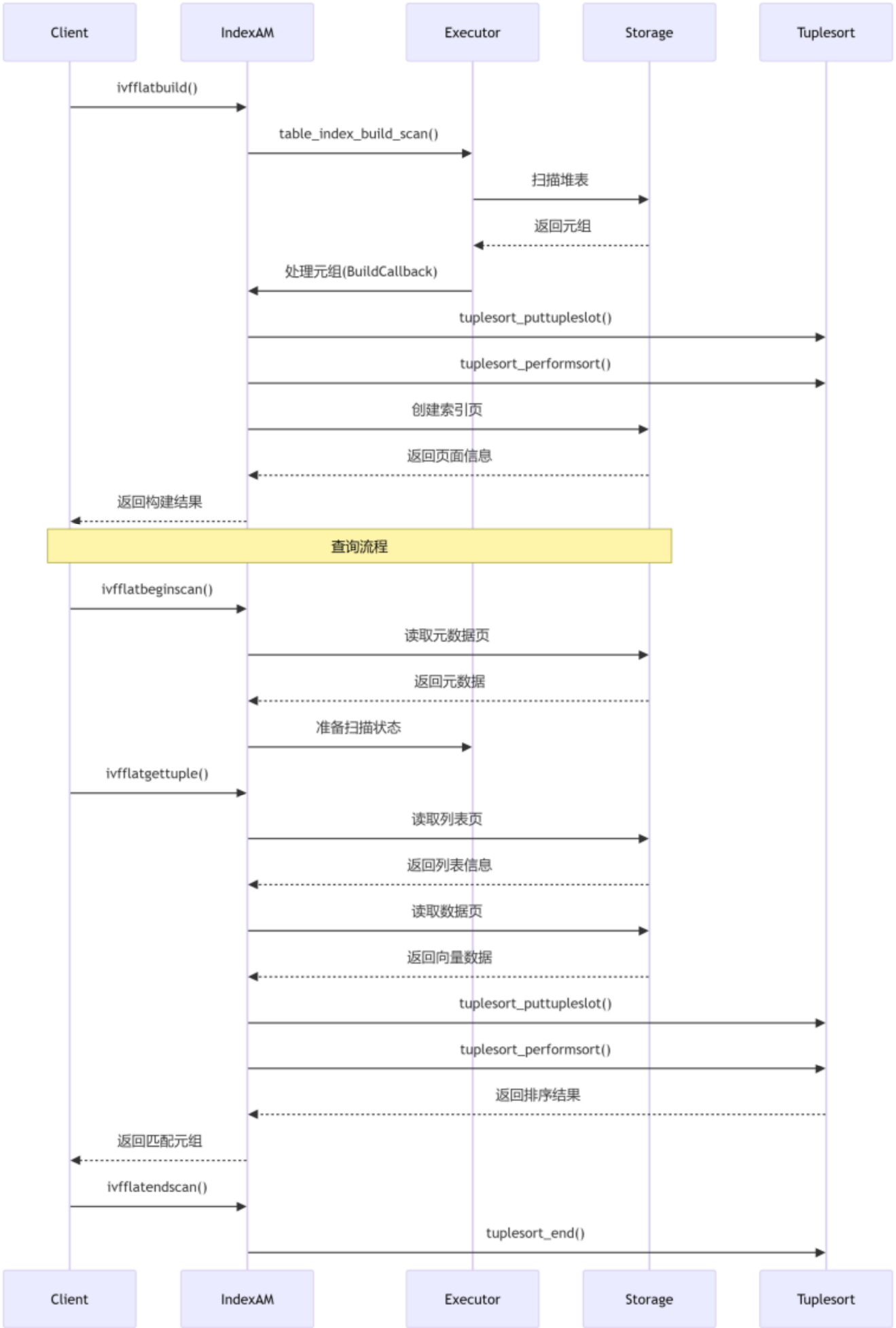


扫描流程图：

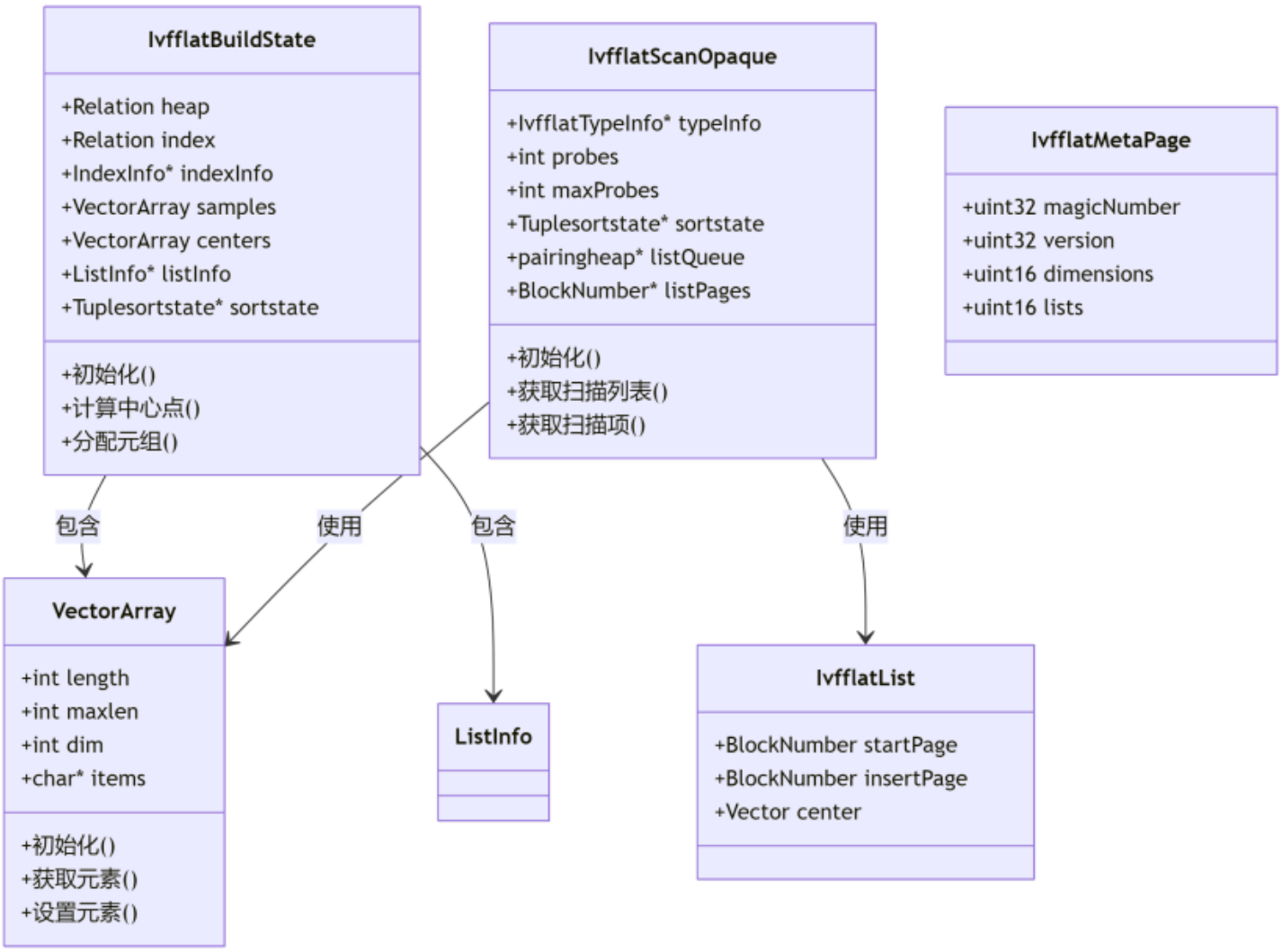

开始索引扫描
初始化扫描状态
ivfflatbeginscan
获取查询向量并归一化
GetScanValue
计算各列表中心与查询向量d
选择距离最小的k个列表
GetScanLists
迭代式扫描扩展
遍历选中列表中的向量
计算向量与查询向量的距离
将结果存入排序器
tuplesort_puttupleslot
是否遍历完所有选中列表?
否
是
执行排序
tuplesort_performsort
获取排序结果
tuplesort_gettupleslot
是否有更多结果?
检查是否需要扩展扫描列表
是



类图：


Client
IndexAM
Executor
Storage
Tuplesort
ivfflatbuild()
table_index_build_scan()
扫描堆表
返回元组
处理元组(BuildCallback)
tuplesort_puttupleslot()
tuplesort_performsort()
创建索引页
返回页面信息
返回构建结果
查询流程
ivfflatbeginscan()
读取元数据页
返回元数据
准备扫描状态
ivfflatgettuple()
读取列表页
返回列表信息
读取数据页
返回向量数据
tuplesort_puttupleslot()
tuplesort_performsort()
返回排序结果
返回匹配元组
ivfflatendscan()
tuplesort_end()
Client
IndexAM
Executor
Storage
Tuplesort

数据流图：



pgvector ivfflat源码阅读

向量数据结构

Vector类型： `vector.h`
最基础的向量类型，各种操作都支持

```
typedef struct Vector
{
    int32       vl_len_;        /* varlena header */
    int16       dim;            /* 维度数 */
    int16       unused;         /* 保留字段 */
    float       x[FLEXIBLE_ARRAY_MEMBER]; /* 向量数据 */
} Vector;
```

向量存储

IvfflatListData 在List中按照链表的方式组织页面

```
typedef struct IvfflatListData
{
    BlockNumber startPage;    // 聚类起始页面
    BlockNumber insertPage;   // 当前插入页面
    Vector      center;       // 聚类中心
} IvfflatListData;
```

IvfflatPageOpaqueData 每个页面都有

```
typedef struct IvfflatPageOpaqueData
{
    BlockNumber nextblkno;    // 指向下一个页面的块号
    uint16      unused;
    uint16      page_id;      // 页面标识
} IvfflatPageOpaqueData;
```

向量压缩：TOAST机制

在数据大小超过一定阈值（默认单向量2KB）时触发，所以可以设较大的TOAST阈值先绕过

向量加载过程

1. 候选聚类选择阶段（`GetScanLists`）

```
cbuf = ReadBuffer(scan->indexRelation, nextblkno);
LockBuffer(cbuf, BUFFER_LOCK_SHARE);
cpage = BufferGetPage(cbuf);
```

- 从磁盘读取聚类中心页面
- 计算查询向量与各聚类中心的距离
- 选择最近的n_probes个聚类

2. 向量数据加载阶段（`GetScanItems`）

```
buf = ReadBufferExtended(scan->indexRelation, MAIN_FORKNUM, searchPage, RBM_NORMAL, so->bas);
LockBuffer(buf, BUFFER_LOCK_SHARE);
page = BufferGetPage(buf);
```

- 遍历选中的聚类页面
- 从每个页面读取向量数据
- 计算与查询向量的距离并排序

Postgresql的Buffer+Page系统

设计目的

1. **减少磁盘I/O**：将频繁访问的页面缓存在内存中
2. **提高性能**：避免重复从磁盘读取相同页面

3. **并发控制**：通过锁机制确保数据一致性
4. **内存管理**：有效管理有限的内存资源

Buffer(Buffer ID)的基本定义

文件位置：`/usr/include/postgresql/16/server/storage/buf.h`（第20行）

```
typedef int Buffer;（后文中为避免歧义称为Buffer ID）
```

Buffer ID实际上是一个整数标识符，用整数索引postgresql内部的缓冲区数组：

- 0 = InvalidBuffer (无效缓冲区)
- 正数 = 共享缓冲区索引 (1..NBuffers)
- 负数 = 本地缓冲区索引 (-1..-NLocBuffer)

为什么使用整数标识符而不是直接页面索引

1. 如果直接使用页面索引，您需要知道每个页面在1. 内存中的确切位置
2. 页面可能不在内存中（需要从磁盘加载）
3. 页面可能被其他进程修改或移动
4. 内存是有限的，不能同时加载所有页面

如何使用Buffer(Buffer ID)处理页面

1. 构建一个双重映射系统 1. BufferTag->Buffer ID 和 2. Buffer ID->实际内存地址
2. 用bufferTag查找哈希表来确定缓冲区，如果找到缓冲区则返回，没找到就分配一个
3. 使用引用计数来确定缓冲区状态
4. 异步读取/写回磁盘

共享内存区域：

+------------------+
| BufferDescriptors | <- 缓冲区描述符数组
+------------------+
| BufferBlocks | <- 实际数据缓冲区
+------------------+
| BufferMapping | <- 哈希表
+------------------+\

为什么需要BufferTag

1. 核心问题：Buffer ID不是固定的，同一个页面在不同时间可能被分配到不同的缓冲区！
   需要BufferTag唯一标识页面（和内存地址不同，是面向数据库的抽象）

```
typedef struct buftag
{
    Oid         spcOid;         // 表空间OID
    Oid         dbOid;          // 数据库OID
    RelFileNumber relNumber;    // 关系文件号
    ForkNumber  forkNum;        // 分支号
    BlockNumber blockNum;       // 块号
} BufferTag;
```

Buffer访问的核心函数

1. ReadBufferExtended 从磁盘读取指定块到缓冲区

文件位置: `/usr/include/postgresql/16/server/storage/bufmgr.h` (第173-175行)

```
extern Buffer ReadBufferExtended(Relation reln, ForkNumber forkNum,
                                 BlockNumber blockNum, ReadBufferMode mode,
                                 BufferAccessStrategy strategy);
```

- `reln`：关系对象
- `forkNum`：文件分支号 (MAIN_FORKNUM = 0)
- `blockNum`：块号
- `mode`：读取模式 (RBM_NORMAL = 0)
- `strategy`：缓冲区访问策略

2. LockBuffer 获取缓冲区的锁，确保并发安全

文件位置: `/usr/include/postgresql/16/server/storage/bufmgr.h` (第242行)

```
extern void LockBuffer(Buffer buffer, int mode);
```

锁模式定义 (第155-157行):

```
#define BUFFER_LOCK_UNLOCK      0
#define BUFFER_LOCK_SHARE       1    // 共享锁
#define BUFFER_LOCK_EXCLUSIVE   2    // 排他锁
```

3. BufferGetPage 从Buffer获取Page指针，用于访问页面内容

文件位置: `/usr/include/postgresql/16/server/storage/bufmgr.h` (第280-283行)

```
static inline Page
BufferGetPage(Buffer buffer)
{
    return (Page) BufferGetBlock(buffer);
}
```

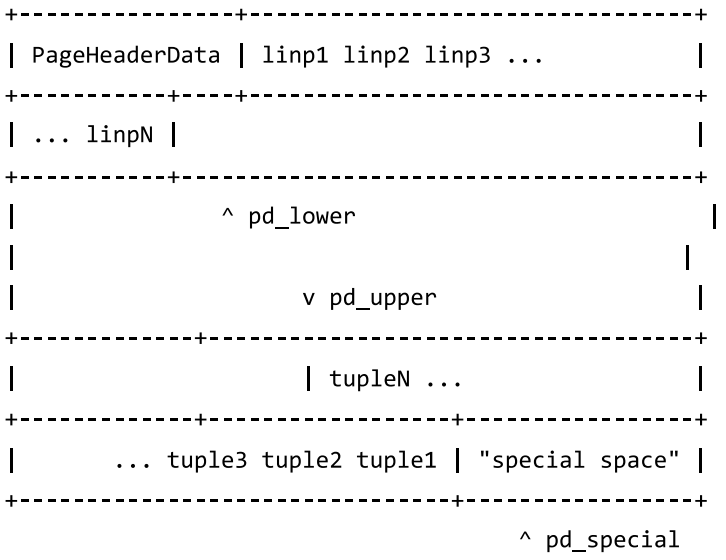
Page结构

PageHeaderData结构

文件位置: /usr/include/postgresql/16/server/storage/bufpage.h (第108-120行)

```
typedef struct PageHeaderData
{
    PageXLogRecPtr pd_lsn;          // LSN
    uint16          pd_checksum;    // 校验和
    uint16          pd_flags;       // 标志位
    LocationIndex pd_lower;         // 空闲空间起始偏移
    LocationIndex pd_upper;         // 空闲空间结束偏移
    LocationIndex pd_special;       // 特殊空间起始偏移
    uint16          pd_pagesize_version;
    TransactionId pd_prune_xid;     // 最老的可清理XID
    ItemIdData      pd_linp[FLEXIBLE_ARRAY_MEMBER]; // 行指针数组
} PageHeaderData;
```

页面布局



Page访问函数

1. PageGetMaxOffsetNumber 返回页面中最大的偏移号即页面中的项目数量

文件位置: /usr/include/postgresql/16/server/storage/bufpage.h (第250-260行)

```
static inline OffsetNumber
PageGetMaxOffsetNumber(Page page)
{
    PageHeader      pageheader = (PageHeader) page;

    if (pageheader->pd_lower <= SizeOfPageHeaderData)
        return 0;
    else
        return (pageheader->pd_lower - SizeOfPageHeaderData) / sizeof(ItemIdData);
}
```

2. PageGetItemId函数 根据偏移号获取行指针(ItemId)

文件位置: /usr/include/postgresql/16/server/storage/bufpage.h (第240-245行)

```
static inline ItemId
PageGetItemId(Page page, OffsetNumber offsetNumber)
{
    return &((PageHeader) page)->pd_linp[offsetNumber - 1];
}
```

3. PageGetItem函数 根据行指针获取实际的数据项

文件位置: /usr/include/postgresql/16/server/storage/bufpage.h (第320-327行)

```
static inline Item
PageGetItem(Page page, ItemId itemId)
{
    Assert(page);
    Assert(ItemIdHasStorage(itemId));

    return (Item) (((char *) page) + ItemIdGetOffset(itemId));
}
```

GPU支持实现方案

优化思路

1. 针对Ivfflat当中的精筛过程，将原本的CPU串行处理改为GPU并行处理，优化向量距离计算和排序
2. CPU、GPU本身性能很高，访存是性能瓶颈。要让CPU多拿、早拿数据给DMA做传输，让数据带宽尽可能高
3. GPU擅长做“整块”的工作，如果有长短不一的工作，最好划分成一块块较短的工作

向量存储：内存

新增VectorBatch类型：

`vector_batch.h`
用来传入一批次向量

```
typedef struct VectorBatch
{
        int32           vl_len_;        /* varlena header */
        int16           count;                  /* 向量数量 */
        int16           dim;                    /* 向量维度 */
        int32           unused;                 /* 保留字段 */
        Vector          vectors[FLEXIBLE_ARRAY_MEMBER];  /* 向量数据 */
} VectorBatch;
```

1. 支持在ann-benchmark-sql-c各个环节使用vector-batch
2. 对比 `VectorArray`：这是一个在 `ivfflat.h` 中实现的数据类型，用于高效建立ivfflat索引，鉴于我们不动建索引的过程，就不使用这个数据结构了
3. 现在向量数据体还是使用顺序存储，将来可能要变成交错存储

新增GPU读取list的方法

确定要查找的list之后，CPU将这些list的页面锁定，DMA挨个上传到GPU上

向量存储：磁盘

GPU通过GPU Direct Storage技术，能够绕过CPU直接从SSD读取数据到显存，但是需要满足如下条件：

1. 支持GPUDirect的GPU（如NVIDIA A100、H100或RTX 30系列及以上）
2. NVMe SSD（需支持PCIe 3.0/4.0/5.0协议），且推荐高性能型号（如PCIe 4.0/5.0 SSD）以匹配GPU带宽需求

这次是用不上了，看上去非常复杂。而且rummy所有数据都在内存里，也不涉及磁盘操作

cuda算子

尽量32个向量为一组并行处理，

Things to do:

1. 更改批量查询的order_by查询，支持对每个query按照距离排序，再整合成一张表输出
2. 添加pgvector-python库支持，**包括将ndarray输入到数据库中**，将数据库中的表格变成可供ann-benchmark识别的结果