本科生课程设计报告

题目： B+ 树 Bulkloading 多核并行设计

组　　长	冼子婷（18338072）
组　　员	周启昀（19335296） 廖雨轩（18322043） 胡文浩（18346019）
院　　系	计算机学院
专　　业	软件工程

2022 年 1 月 1 日

目录

一、Bulkloading 过程

1. 对项目代码的理解

1）结点的数据结构

基础结点：

在 `b_node.h` 中定义了 B+ 树的结点基类 `BNode`，索引节点 `BIndexNode`，叶子结点 `BLeafNode`，`b_node.cc` 实现相应的函数与方法。

`BNode` 结点作为基础类，具有基础属性：

- 表示当前结点层数 `level_`
- 当前结点数据数量 `num_entries_`
- 兄弟节点的地址 `left_sibling_` 和 `right_sibling_`
- 键值数组 `int *key_`
- 是否脏写标志 `dirty_`
- 当前结点的硬盘地址 `block_`，表示节点存储到文件的区域编号
- 结点最大数据容量 `capacity_`
- 当前结点的 B+ 树 `BTree* btree_`

除了构造函数初始化各个属性和析构函数释放空间外，`BNode` 还具有基础虚函数，用于在派生类实现：

- `init(int level, Btree *btree)` 在 B 树的一层中初始化结点。
- `init_restore(BTree *btree, int block)` 从文件中初始化一颗 B 树。
- `write_to_buffer(char *buf)` 和 `read_from_buffer(char *buf)` 用于在缓冲区读写。
- `get_entry_size()` 获得当前结点数据大小。
- `find_position_by_key(float key)` 通过键值查找数据位置。
- `get_key(int index)` 通过下标访问键值数组 `key_` 中的数据。
- `get_left_sibling()` 和 `get_right_sibling` 访问左右兄弟结点以及 `set_left_sibling(int l)` 和 `set_right_sibling(int r)` 设置左右兄弟结点的地址。
- `get_block()` 获得当前硬盘地址，` set_block(int *block)` 设置当前结点硬盘地址。
- `get_num_entries()` 获取当前结点的数据数量。
- `get_level()` 获取当前结点在 B+ 树的高度。
- `isFull()` 判断当前结点数据是否超出了容量。
- `get_header_size()` 获取 B+ 树结点的头部大小 SIZECHAR + SIZEINT * 3，即层数 CHAR ，左右兄弟节点地址以及当前结点数据数量。
- `get_key_of_node()` 返回当前结点键值数组种的第一个 `key`

索引结点：

`BIndexNode` 索引节点继承于 `BNode` 基础结点类，不仅具有相同的属性与函数，新增了属性：

- `int *son` 表示孩子结点的地址数组。

同时实现了父类 `BNode` 中的虚函数：

- `init(int level, BTree *btree)`，初始化一个新结点，通过读取读取 `btree->file` 文件获得当前地址以及可分配磁盘大小 `btree_->file_->get_blocklength()`，计算出当前结点最大容量 `capacity_`，用于分配孩子结点地址数组 `son_` 以及键值数组 `key_` 的空间并初始化。
- `init_restore(Btree *btree, int block)` 从硬盘中加载 B+ 树结点，并为其分配空间。
- `read_from_buffer(char *buf)` 和 `write_to_buffer(char *buf)` 从缓存中读写 B+ 树结点。
- `get_entry_size()` 获得索引节点一个数据对 `<key_, son_>` 的大小，即 SIZEFLOAT + SIZEINT 。
- `find_position_by_key(float key)` 和 `get_key(int index)` 查询函数。
- `get_left_sibling()` 和 `get_right_sibling()` 通过 `init_restore(btree_, left_sibling)` 传入当前 B+ 树和左右兄弟结点的地址，使用 `init_restore` 从硬盘地址中加载一个结点，从而获得左右兄弟结点。
- `get_son(int index)` 通过下标获得当前索引节点的孩子结点的地址 `son_[index]`
- `add_new_child(float key, int son)` 向当前索引结点中加入数据，并且设置脏写标记。

叶子结点：

`BLeafNode` 叶子节点同样继承于 `BNode` 基础结点类，由于叶子节点存储数据，其具有新的属性：

- `num_keys_` 当前叶子节点的键值对数量。
- `capacity_keys_` 最大容量。
- `int *id_` 指向数据位置的数组。

同时实现了父类 `BNode` 中的虚函数，初始化函数 `init()` 和 `init_restore()` 与索引节点大致相同，但需要额外处理 key 和 value 数组的大小，其余如从缓冲区读写与获得左右节点的函数实现都类似。但叶子节点增加了一些对 key 和 value 处理的函数：

- ` get_increment() ` 查看增加的叶子节点数量。
- `get_num_keys()` 查看 keys 数量的函数。
- `get_entry_id(int index) ` 跟据下标 index 的值查看 entry 编号 `id` 的值。
- `add_new_child(float key, int id)` 新增一个到当前叶子节点的尾部。

2）B+树数据结构

在 `b_tree.h`，`b_tree.cc` 中定义了 B+ 树的数据结构（在 QALSH 中用于构建带索引的哈希表）`BTree` 类，具有基本的树形结构属性：

- `root_` 根节点的硬盘地址
- `BNode *root_ptr` 指向根节点的指针
- `BlockFile *file_` 指向当前 B+ 树存储的硬盘文件

对 B+树除了基本的构造和析构函数外，还具有以下函数方法：

- `init(int b_length, const char *fname)` 通过指定文件名以及其大小，初始化一棵 B+ 树，并且将根节点初始化为索引节点，初始化当前的属性 `root_` 和 `root_ptr`
- `init_restore(const char *fname)` 根据硬盘中的树文件加载一棵 B+ 树，初始化过程和 `init()` 类似
- `load_root()` 和 `delete_root()` 分别对根节点加载和删除。
- `bulkload(int n, const Result *table)` 串行地从数据集 `Result* table` 中批量地载入数据，并且构建 B+ 树，将在后面讲解。

3）文件流与磁盘交互

在 `block_file.h`，`block_file.cc` 中定义了数据结构如何与磁盘进行交互，构建的 B+ 树的每一个节点会依次存储到一个文件的不同区域中，每一个节点有一个 block 号，表示节点存储到文件的区域编号。

`BlockFile` 类具有与磁盘交互的属性：

- `FILE *fp_` 文件指针
- `char *fname` 文件名
- `new_flag_` 标记当前文件是否为新文件
- `block_length` 磁盘文件一个 block 的长度，
- `act_block` fp 位置的块数
- `num_blocks` blocks 的数量

在构造函数中，需要制定 block 的长度以及文件名用于初始化当前的 BlockFile 或打开已有的 BlockFile，同时需要具有文件读写的相关函数方法：

- `put_bytes(const char *bytes, int num)` 向当前文件写入长度为 num 的 bytes 字符串
- `get_bytes(const char *bytes, int num)` 从当前文件读取长度为 num 的字符串写入到 bytes
- `seek_block(int bnum)` 使用 `fseek` 将文件流 `fp` 偏移 `bmun` 个 block 位置，即 `(*bnum*-act_block_)*block_length_`
- `file_new()` 检查当前文件是否是新文件

- `get_blocklength()` 获取一个 block 块的长度
- `get_num_of_blocks()` 获取所有 block 块的数量
- `fwrite_number(int num)` 使用上述 `put_bytes()` 函数向文件流中写入一个数
- `fread_number()` 使用上述 `get_bytes()` 函数从文件流中读取一个数
- `read_header(char *buffer)` 和 `set_header(char *buffer)` 从缓冲区中读取或设置剩余字节，即写入或读取当前第一个 block 块的值，但使用了 `fseek(fp_, BFHEAD_LENGTH, SEEK_SET);` 偏移，即并不是操作当前 blockfile 的 header，
- `read_block(Block block, int index)` 和 `write_block(Block block, int index)` 使用下标 index 读取 BlockFile 中第 index 个 block 并写入到参数 block 字符串中，或读取当前参数 block 块中的数据，写入到 BlockFile 中。
- `append_block(Block block)` 将一个 block 块添加到当前文件流 BlockFile 中
- `delete_last_blocks(int num)` 通过偏移删除当前文件流中最后 num 个 block 块

4）辅助文件

在 `pri_queue.h` 和 `pri_queue.cc` 中定义了基本的数据结构 `Result` 具有 `id_` 属性表示值，`key_` 表示键，以及用于比较数据的比较函数 ：`ResultComp` 升序比较函数和 `ResultCompeDesc` 降序比较函数。以及 `Mink_List` 数据结构维护最小 k 值，是近似最近邻检索算法 QALSH 中的数据结构，用于存储和查找 K 近邻，具有获取当前最大值，最小值以及第 i 个值等查询函数和插入函数，其具有 `k` 个数据，当前 `num` 个激活数据，以及 `Result *` 链表数组。

`make_data.cpp` 用于生成 B+ 树中的结点，其定义了 `Result` 结构体记录数据用于存储生成的数据，具有 `id_` 属性表示值，`key_` 表示键。通过定义结点数 `n` 与生成值的范围 `range`，使用 `random() % range` 生成随机数（`id_` 属性递增，`key_` 值随机），通过 `qsort()` 快速排序，按照 `key_` 升序排序，如果 `key_` 相同则按照 `id_` 升序排序。最后使用 `ofstream` 输出文件流，将所有节点数据按照 `key_, id_` 格式存入 `data.csv` 逗号分隔值数据文件。

在 `def.h` 中声明了 Block 字符串即磁盘中的地址。同时使用宏命令声明了一些比较函数如 MIN，MAX，以及不同类型的数值的常量如 MAXREAL，MAXINT 以及自然底数 E 和圆周率 PI 等等，但其中 INT 类型的最小值以及 REAL 实数（FLOAT）类型的最小值定义有误，参照 C/C++ numeric limits 库中的宏命令，`INT_MIN = (-INT_MAX - 1)`，而 `FLT_MIN = 1.175494e-38`。

在 `random.h`、`random.cc` 中声明了各类概率与统计学函数，用于生成随机数，如高斯分布、柯西分布、列维分布等等，以及各种分布的概率密度函数，相关系数等等统计学工具。

在 `util.h`、`util.cc` 中声明了时间变量 `g_start_time` 和 `g_start_time`，用于使用 Linux 中 `gettimeofday()` 的时间函数记录起始时间、结束时间。并且声明了统计

当前运行时间、基准真相、IO/内存占用的比率等全局变量，同时定义和声明了许多实用的函数，比如 `create_dir(char *path)` 创建文件目录，`int read_txt_data(int, int, const char*, float **)` 读取数据等等。提供了各种使用的文件读写、统计时间、统计数据工具。

5）主要入口

在 `main.cc` 中定义了使用 `make_data.cpp` 随机生成数据文件 `./data/dataset.csv` ，数据的个数 `n_pts_` ，生成的 B+ 树文件流 `./result/B_tree`（用于构建 BlockFile 和存储构建 B+ 树的结果），以及 B+ 树一个结点的 block 大小 `B = 512`。

数据集 `Result *table = new Result[n_pts_] ` 用于读取文件流 `dataset.csv` 中的数据，使用 `atof()` 和 `atoi` 将逗号分隔数据转换成 `float` 类型的 `key_` 值以及 `int` 类型的 `id_` 值。

读取完数据后，关闭文件流，统计起始时间 `start_t` 并且构建一棵 B+ 树并对其进行初始化，随后使用 `trees_bulkload(n_pts_, table)` 将数据集中的数据批量地载入到 B+ 树中，当构建完成后，输出串行 BulkLoad 的时间。

2. 局部敏感哈希 LSH

综上，这些数据结构和辅助文件都来自于 HuangQiang 学长在 QALSH 论文中实现的源代码，其用于解决高纬度的欧几里得空间上的最近邻搜索问题，同时，这种高纬度的最近邻搜索问题可以在关系数据库中实现，比如构建索引等等。

由于在高纬中 NN 问题线性搜索非常耗时，所以需要加入索引项，以实现在常数时间内得到查找结果，比如使用 KDtree 或是近似最近邻查找（Approximate Nearest Neighbor, ANN），而 LSH (Locality-Sensitive Hashing, LSH) 就是 ANN 中的一种，在 HuangQiang 学长的源代码中也出现了 KDTree 等数据结构。

LSH 的思想与哈希产生冲突类似，如果原始数据空间中的两个相邻数据点通过哈希映射 hash function 后，产生碰撞冲突的概率较大，而不相邻的数据点经过映射后产生冲突的概率较小。就可以在相应的映射结果中继续进行线性匹配，以降低高维数据最近邻搜索的问题。LSH 用于对大量的高维数据构建索引，并且通过索引近似最近邻查找，主要应用于查找网络上的重复网页、相似网页以及图像检索等等。其中在图像检索领域中，由于其可以对图片的特征向量进行构建 LSH 索引，加快索引速度，所以在图像检索中 LSH 有着重要用途。

3. BulkLoad 过程

BulkLoad 的整个过程在 `b_tree` 中 `int BTree:bulkload(int n, const Result *table)` 函数，从数据集 `table` 中构建 `n` 个结点的索引：

1） 构建叶子结点

首先构建叶子节点存储数据集 `table` 中的所有 `key` 和 `value`，即下图中的循环，其中 `first_node` 标记表示当前叶子结点是否是第一个结点，`start_block` 和 `end_block` 表示当前叶子结点的 block 起始位置和结束位置：

```
// ----------------------------------------------------------------------
//  build leaf node from <_hashtable> (level = 0)
// ----------------------------------------------------------------------
bool first_node = true;  // determine relationship of sibling
int start_block = 0;     // position of first node
int end_block = 0;       // position of last node

for (int i = 0; i < n; ++i) {
    id = table[i].id_;
    key = table[i].key_;

    if (!leaf_act_nd) {
        leaf_act_nd = new BLeafNode();      // init current left node
        leaf_act_nd->init(0, this);

        if (first_node) {
            first_node = false;  // init <start_block>
            start_block = leaf_act_nd->get_block();
        } else {
            //   prev <-> cur : doubly linked list
            leaf_act_nd->set_left_sibling(leaf_prev_nd->get_block());
            leaf_prev_nd->set_right_sibling(leaf_act_nd->get_block());

            delete leaf_prev_nd;
            leaf_prev_nd = NULL;
        }
        end_block = leaf_act_nd->get_block();
    }
    leaf_act_nd->add_new_child(id, key);  // add new entry

    if (leaf_act_nd->isFull()) {  // change next node to store entries
        leaf_prev_nd = leaf_act_nd;
        leaf_act_nd = NULL;
    }
}
```

对数据集进行遍历：

- 如果**当前叶子结点不存在**，则**新建**一个叶子结点 `leaf_act_nod = new BLeafNode` 其中 `act_nd` 关键字表示**当前结点**，使用 `init(0, this)` 进行初始化并加入到当前 B+ 树中；
- 如果**当前新建的叶子节点**是 B+ 树当前**第一个叶子节点**，则设定 `start_block` 起始叶子结点的 block 位置为当前叶子结点的 block 号。
- 如果**当前新建的叶子节点**不是 B+ 树当前第一个叶子节点，则需要指定其左兄弟结点，即构建一个双向的链表，使用 `leaf_act_nd->set_left_sibling(leaf_prev_nd->get_block())` 设置当前结点左兄弟结点的 block 号为前一个叶子节点，使用 `leaf_prev_nd->set_right_sibling(leaf_act_nd->get_block())` 设置前一个结点的右兄弟结点的 bloc 号为当前叶子节点，随后释放前一个叶子结点指针。
- 最后，使用 `end_block` 记录当前叶子节点的最后一个 block 号。
- 如果**当前叶子结点存在**，即数据**没有存满**当前叶子结点，则忽略中间的判断，进入下一步添加数据。

使用 `add_new_child(id, key)` 将数据加入到**当前叶子结点**中，如果当前叶子结点**已满**，则将**当前叶子结点指针**赋值给**前一个叶子结点指针** `leaf_prev_nd = leaf_act_nd`，同时将当前叶子节点指针指向空 `leaf_act_nd NULL`，以在一下次遍历中**新建叶子结点**。

2） 构建索引结点

将所有数据存入并构建叶子节点后，释放掉使用过的指针 `leaf_prev_nd` 前一个叶子结点指针和 `leaf_act_nd` 当前叶子结点指针，获得构建完叶子节点后，当前最后一个叶子节点的起始 block 号 `last_start_block`，以及最后一个叶子结点的结尾 block 号 `last_end_block`，从第一层开始**自底向上**构建索引结点（叶子节点在第 0 层）：

```
// ------------------------------------------------------------------------
//  stop condition: lastEndBlock == lastStartBlock (only one node, as root)
// ------------------------------------------------------------------------
int current_level = 1;               // current level (leaf level is 0)
int last_start_block = start_block;  // build b-tree level by level
int last_end_block = end_block;      // build b-tree level by level

while (last_end_block > last_start_block) {
    first_node = true;
    for (int i = last_start_block; i <= last_end_block; ++i) {
        block = i;  // get <block>
        if (current_level == 1) {
            leaf_child = new BLeafNode();
            leaf_child->init_restore(this, block);
            key = leaf_child->get_key_of_node();

            delete leaf_child;
            leaf_child = NULL;
        } else {
            index_child = new BIndexNode();
            index_child->init_restore(this, block);
            key = index_child->get_key_of_node();

            delete index_child;
            index_child = NULL;
        }

        if (!index_act_nd) {
            index_act_nd = new BIndexNode();
            index_act_nd->init(current_level, this);

            if (first_node) {
                first_node = false;
                start_block = index_act_nd->get_block();
            } else {
                index_act_nd->set_left_sibling(index_prev_nd->get_block());
                index_prev_nd->set_right_sibling(index_act_nd->get_block());

                delete index_prev_nd;
                index_prev_nd = NULL;
            }
            end_block = index_act_nd->get_block();
        }
        index_act_nd->add_new_child(key, block);  // add new entry

        if (index_act_nd->isFull()) {
            index_prev_nd = index_act_nd;
            index_act_nd = NULL;
        }
    }
    if (index_prev_nd != NULL) {  // release the space
        delete index_prev_nd;
        index_prev_nd = NULL;
    }
    if (index_act_nd != NULL) {
        delete index_act_nd;
        index_act_nd = NULL;
    }

    last_start_block = start_block;  // update info
    last_end_block = end_block;      // build b-tree of higher level
    ++current_level;
}
root_ = last_start_block;  // update the <root>
```

从**第一个叶子结点**的起始 block 和**最后一个叶子节点**的结束 block 开始遍历，即上一层的起始 block 为 `last_start_block` 和上一层的结束 block 为 `last_end_block`，自底向上构建索引层，直到 `last_end_block == last_start_block` 只剩一个 block 即**根节点**。

自底向上的过程中，又对 `last_start_block` 到 `last_end_block` 上一层结点中每个 block **正序遍历**（保证 key 索引值升序递增）构建索引结点：

- 如果当前是叶子节点上一层（即 `current_level == 1`，**此时 block 号指向叶子节点**），初始化一个叶子节点指针 `leaf_child` 并使用 `init_restore(this, block)` 从当前 B+ 树以及 block 号从文件流中加载当前叶子节点，使用 `leaf_child->get_key_of_node` 获取叶子节点的第一个 key 值用作索引，并且释放 `leaf_child` 叶子结点指针。
- 如果当前不是叶子结点上一层（即**当前 block 号指向索引结点**），初始化一个索引节点指针 `index_child` 并使用 `init_restore(this, block)` 从当前 B+ 树及 block 号从文件流加载当前所以索引节点，使用 `index_child->get_key_of_node()` 获得索引结点的第一个 key 值用作索引，并且释放 `index_child` 叶子结点指针。

获得**索引值**之后，**构建索引结点**：

- 如果**当前索引节点** `index_act_nd` 为空，则使用 `index_act_nd = new BIndexNode()` 并且使用 `init(current_level, this)` 初始化当前层的索引结点：
  - 如果当前索引节点是第一次创建，则记录其当前索引层第一个索引结点的起始 block 值 `start_block =index_act_nd->get_block() `
  - 如果**上一个索引节点已满**，则当前索引结点**不是第一次创建**，需要将其连接****构建双向链表**，使用 `index_act_nd->set_left_sibling(index_prev_nd->get_block())` 设置**当前索引结点的左兄弟结点 block 号**为上一个索引节点，`index_prev_nd->set_right_sibling(index_act_nd->get_block())` 设置上一个索引结点的右兄弟结点的 block 号为当前索引节点，随后释放上一个索引节点指针。
  - 最后，使用 `end_block` 记录当前索引节点的最后一个 block 号。
- 如果**当前索引节点** `index_act_nd` 不为空，即没有装满数据，则忽略上述判断，进入下一步。

使用 `index_act_nd->add_new_child(key, block)` 将索引值 `key` 以及孩子节点的 block 号加入到当前索引结点中，与叶子节点的构建一样，如果当前索引节点已满，则需要将当前索引节点指针赋值给上一索引结点指针 `index_prev_nd = index_act_nd`，并且将**当前索引节点指针赋空**`index_act_nd = NULL`，以便下一次遍历创建新的索引节点。

遍历完**上一层**所有结点的所有 block 号并构建索引节点后，释放 `index_prev_nd` 前一结点指针和 `index_act_nd` 当前结点指针，并且更新 `last_start_block` 上一层起始 block 号为当前层的起始 block，`last_end_block` 上一层的结束 block 号为当前层的结束 block，并且层数加一，表示**自底向上**构建索引节点。

最终当 `start_block == end_block` 时，表示根节点层也构建完成，整棵 B+ 树就构建完成，BulkLoad 过程结束。

3） 总结

Bulkloading 的目标是对有序的数据批量构建 B+ 树。对于本次课程设计，它分为以下两个步骤：

- 构建叶子节点。本质是一层双向链表，链表的每个节点含有两个数组，一个存储键 `key`，另一个存储值 `id`，后者实质上是磁盘块的索引。一个 `key` 对应 16 个 `id`，这里主要是在节省磁盘空间和访问速度之间的权衡。
- 构建索引节点。本质是若干层的双向链表，每一层的一个节点会指向下一层的多个节点，以此达到 B+ 树的搜索功能。每个节点也有两个数组，一个存储 `key`，另一个存储指向叶子节点的磁盘块的指针，它们在数量上则是一比一的关系。

助教提供的源码采用串行化的加载算法，先构造叶子节点，后构造索引节点。

- 对于叶子节点，按顺序遍历输入的数据，逐个地构建出链表的节点。遍历结束之后构建完成。
- 对于索引节点，首先按顺序遍历叶子节点层里的所有节点，拿到它们的 `key` 和磁盘块指针，以此构建一层链表。之后，又以前面构建的一层链表构建新的一层链表，重复过程直到得到单个节点（也就是根节点）。

二、算法并行的设计思路

1. 构建叶子结点

经过分析发现，串行构建代码中存在以下可进行并行优化的点：

1. 可以将待构建的链表节点分为 N 份，由多线程并发地构建每一份，最后连接起来。
2. 对 `block_file.cc` 中 `BlockFile::append_block()` 函数的调用产生了大量的随机 IO，对于机械硬盘而言是非常严重的性能问题。因此，可以先缓存 `b_node.cc` 的 `BLeafNode::init()` 中对前者函数的调用，将多次随机 IO 转换为单次顺序 IO。

结合**上述两点**，得到这一步的并行设计思路：

设有 N 个待构建的叶子节点，且待构建的叶子节点为 $N_i, i \in [0,1,\dots,N]$。设当前有 M 个 CPU 逻辑核心。我们先不考虑 IO，则节点的构建是 CPU 密集型任务，因此可以使用 $M-1$ 个线程来高效地运行接下来的任务。令 M_j 为第 j 个线程，其中 $j \in [0,1,\dots,M-1]$。

对 $\mathrm{k} = \mathrm{j} + \mathrm{h}(\mathrm{M}-1), \mathrm{h} \in \left[0,1,\dots,\left\lfloor \frac{\mathrm{N}}{\mathrm{M}-1} \right\rfloor - 1\right]$，将 N_k 分配给 M_j，这样每个线程都得到了 $\{N_i\}$ 的一部分。让它们各自按下标顺序开始构建叶子节点（我们已经提前算出每个叶子节点最多能装多少个 id 值）。我们这时使用另一个线程 $\mathrm{T_c}$ 来接收 $\mathrm{M_j}$ 输出的叶子节点 $\mathrm{N_i}$，将它们按顺序连接（即设置左右指针），缓存一定数量的节点之后一

次性地写入磁盘。这里由于我们将原本的多次小规模的随机 IO 转换成了一次大规模的顺序 IO，效率会大大提升。

M可以通过小顶堆来实现向 T_c 交付有序的叶子节点，同时通过自旋锁来防止竞态条件。T_c 通过从小顶堆获取有序地叶子节点，将若干个数量的节点一次性写入磁盘。

当所有节点都被线程 T_c 落盘之后，它可以得到这一层节点的起始节点的指针以及末尾节点的指针，作为下一层的输入。

2. 构建索引结点

索引节点的构建大致流程和构建叶子节点一样，不过有一点不同：索引节点在构建时需要获得儿子节点的 key，这就需要从磁盘读取信息。从整体来看，构建索引节点的过程会是多次的读—写—读循环，即先读取若干个硬盘块得到儿子节点的 `key` 值，然后将新生成的索引块数据写入磁盘文件中的末尾，之后又回头读取新的儿子节点，不断往复。

这样对硬盘的利用并不高效。因此，这里是另一个可以进行并行优化的点：先让某个线程一次性顺序读取大量的儿子节点到内存中，让索引节点可以直接从内存读取儿子节点的数据。

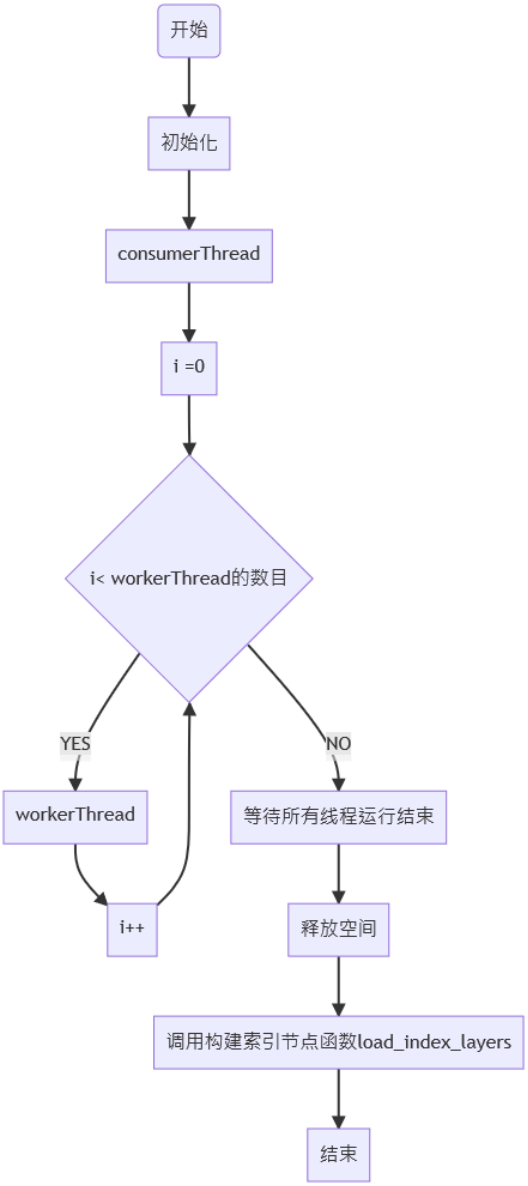
对于每一层索引节点的构建，设有 B 个待扫描的 block，有 N 个待构建的索引节点，且待构建的索引节点为 $N_i, i \in [0,1,\dots,N]$。设当前有 M 个 CPU 逻辑核心。使用 $M-1$ 个线程来高效地运行接下来的任务。令 M_j 为第 j 个线程，其中 $j \in [0,1,\dots,M-1]$。

我们使用 T_c 线程将 B 个块切分为若干个部分 B_i，将 B_i 使用一次顺序磁盘读取加载进内存中。之后，将 B_i 交给 M_j，M_j 会将对应于自身的部分的 B_i 进行处理，例如使用读进内存的数据初始化 `BLeafNode` 或 `BIndexNode`，得到正确的 `key` 等，其它步骤和构建叶子节点时一致。完成后交付给 T_c，完成后续步骤。

三、算法流程图

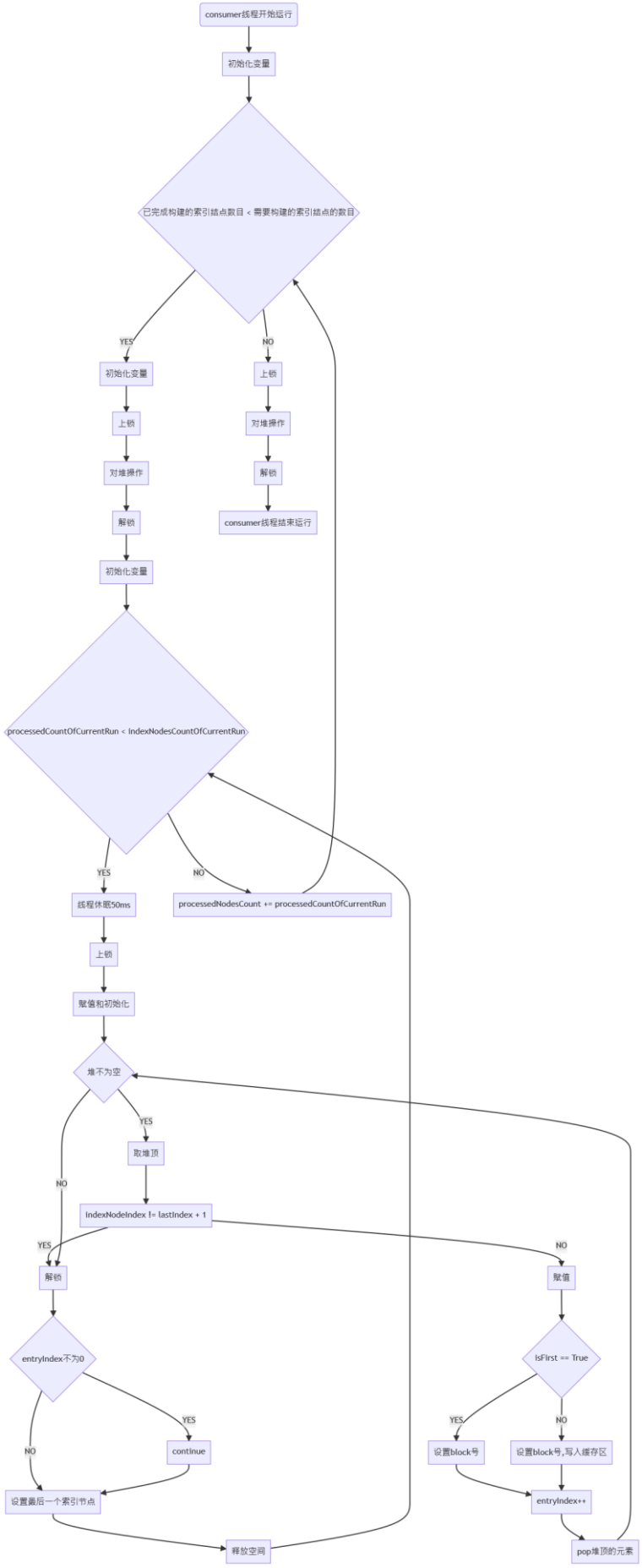
注：需要初始化和赋值的内容太多，流程图里省略，具体见代码。

1. 总算法流程图

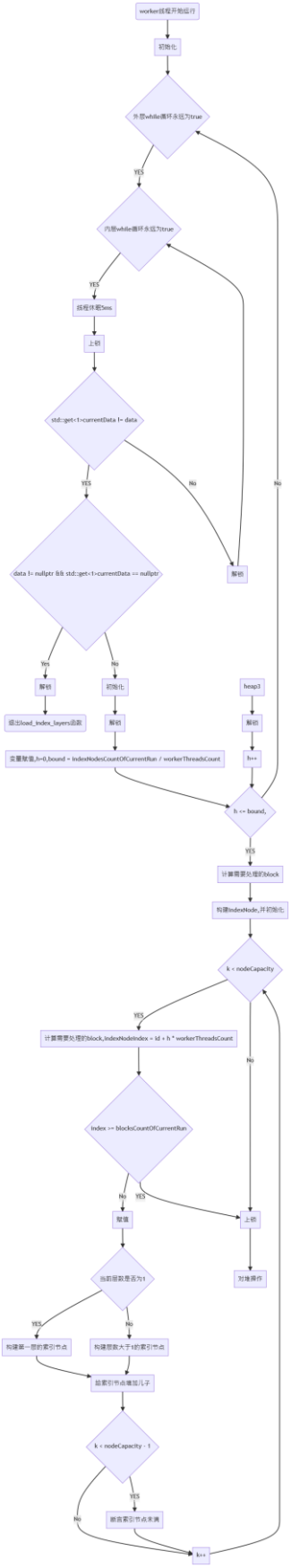


2. 构建叶子结点

1）consumerThread 线程运行

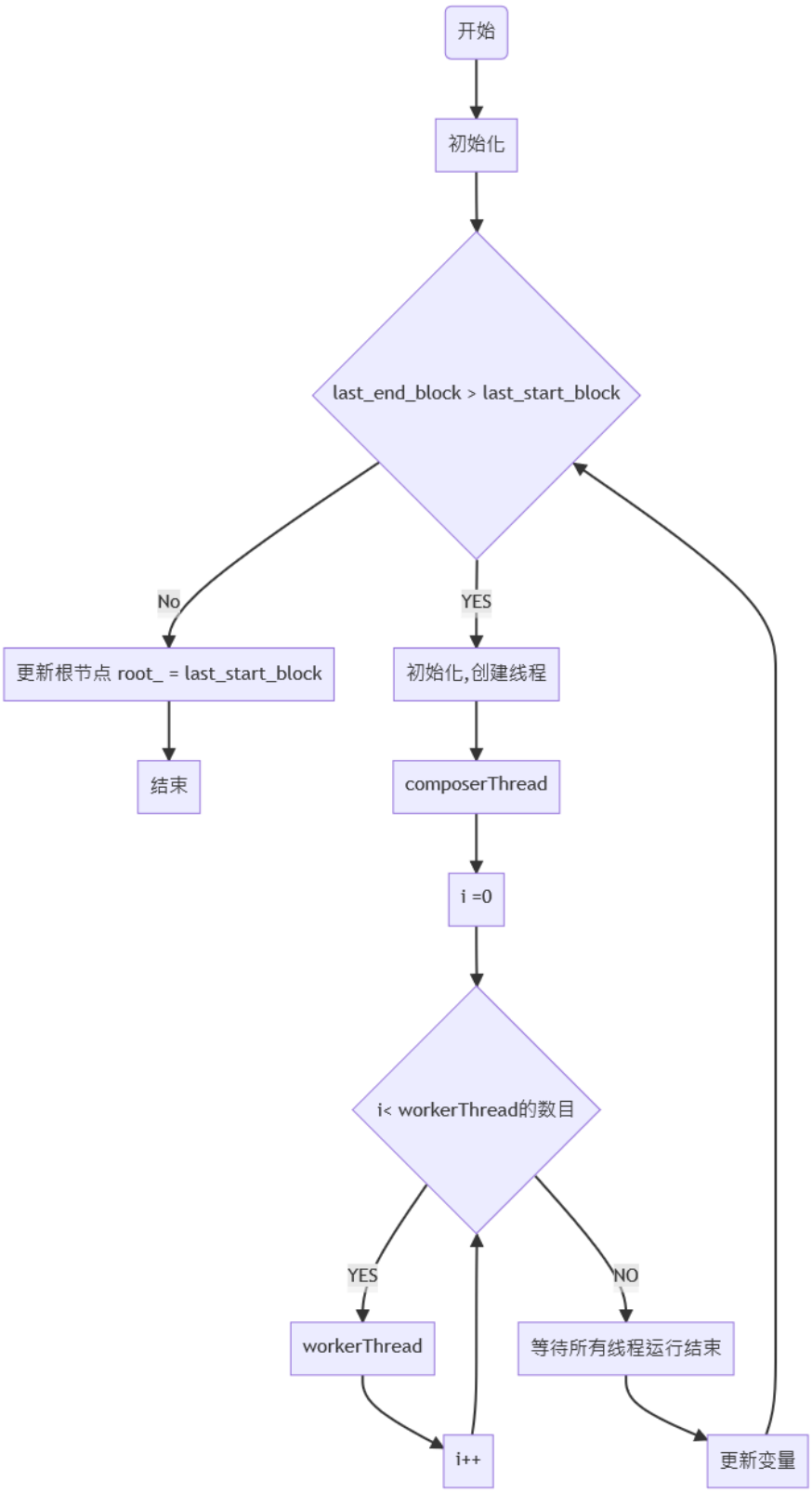

consumer线程开始运行
初始化变量
已完成构建的索引结点数目 < 需要构建的索引结点的数目
YES
NO
初始化变量
上锁
上锁
对堆操作
对堆操作
解锁
解锁
consumer线程结束运行
初始化变量
processedCountOfCurrentRun < indexNodesCountOfCurrentRun
YES
NO
线程休眠50ms
processedNodesCount += processedCountOfCurrentRun
上锁
赋值和初始化
堆不为空
YES
NO
取堆顶
indexNodeIndex != lastIndex + 1
YES
NO
解锁
赋值
entryIndex不为0
isFirst == True
YES
YES
NO
NO
continue
设置block号
设置block号,写入缓存区
设置最后一个索引节点
entryIndex++
释放空间
pop堆顶的元素

2） workerThread 线程运行（for 循环内的算法流程图)

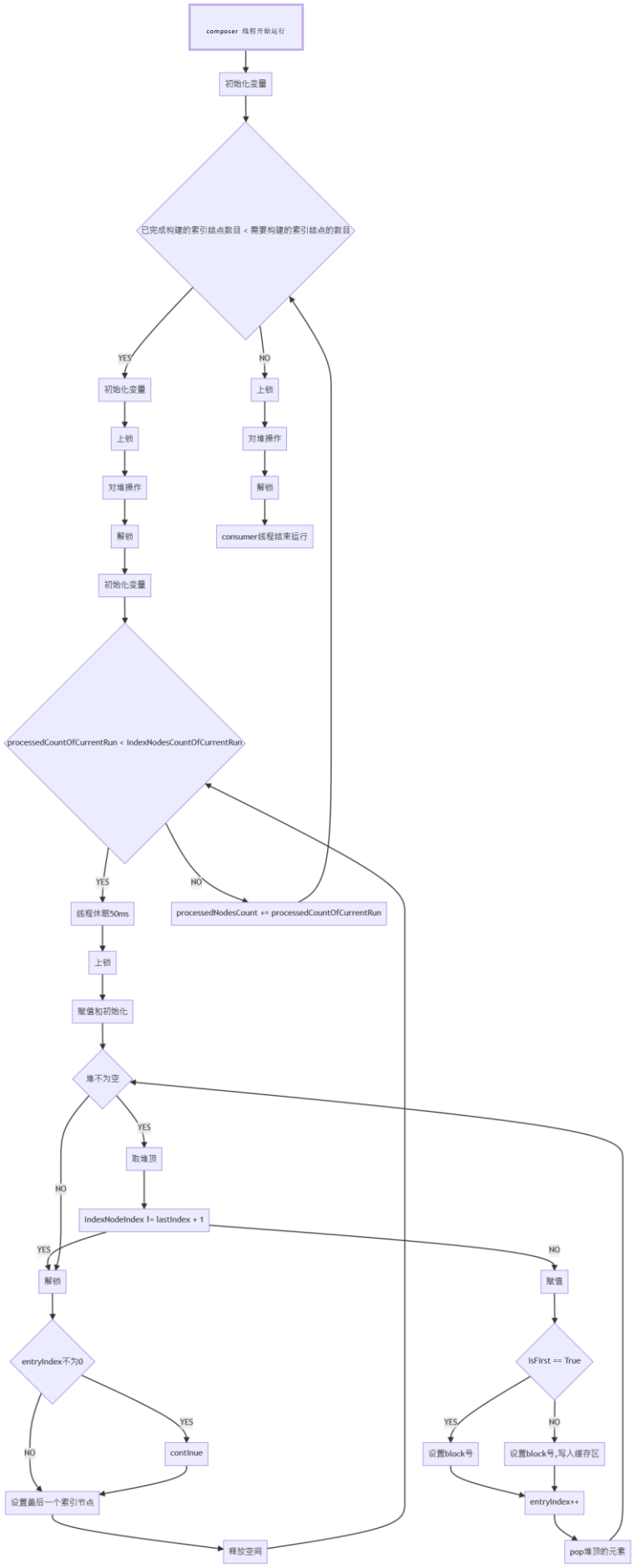


3. 构建索引结点

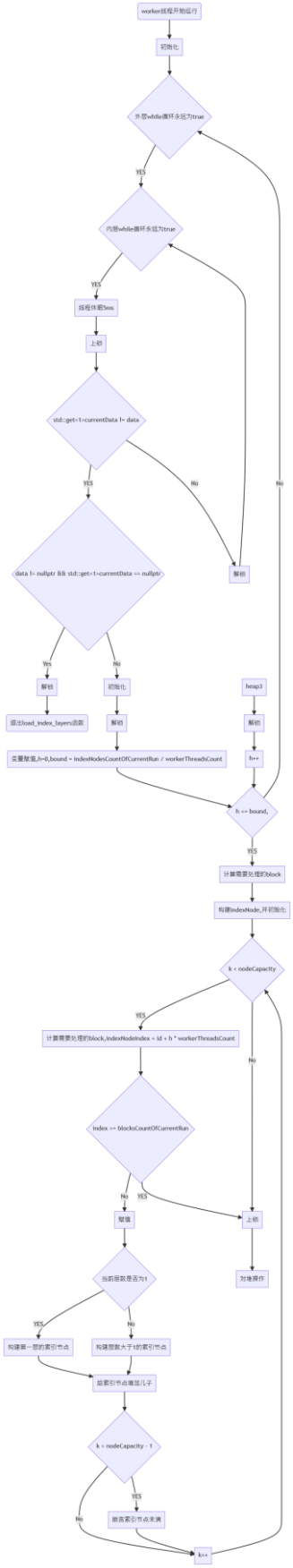
1）load_index_layers 函数算法总流程



2）composerThread 线程运行


composer 线程开始运行
初始化变量
已完成构建的索引结点数目 < 需要构建的索引结点的数目
YES
NO
初始化变量
上锁
对堆操作
解锁
初始化变量
上锁
对堆操作
解锁
consumer线程结束运行
初始化变量
processedCountOfCurrentRun < indexNodesCountOfCurrentRun
YES
NO
线程休眠50ms
processedNodesCount += processedCountOfCurrentRun
上锁
赋值和初始化
堆不为空
YES
NO
取堆顶
indexNodeIndex != lastIndex + 1
YES
NO
解锁
赋值
entryIndex不为0
isFirst == True
YES
YES
NO
NO
continue
设置block号
设置block号,写入缓存区
设置最后一个索引节点
entryIndex++
释放空间
pop堆顶的元素

3）workerThread 线程运行（for 循环内的算法流程图)



四、关键代码描述

1. 并行构建叶子结点

- 由于是并发构建叶子结点，因此`fisrt_node`的位置是位于 block 块号 1 的位置，即：

```
int start_block = 1;      // position of first node, always be 1
int end_block = 0;        // position of last node
```

- 首先，创建并发线程，创建的线程数目通过函数 `std::thread::hardware_concurrency() `产生，而这个函数会返回能并发在一个程序中的线程数量。断言认为线程数量大于等于 1，并且新开一个线程池 `workerThreads`。

```
const auto workerThreadsCount = std::thread::hardware_concurrency() - 1;
assert(workerThreadsCount >= 1);
std::vector<std::thread> workerThreads;
```

- 定义块头部的大小，key 值空间的大小，entry 的大小，树节点的容量，叶子结点的数量。并计算出在我们的设计思路中每个线程需要处理节点个数的理论平均值。

```
const auto headerSize = SIZECHAR + SIZEINT * 3;
const auto keySize =
    ((int)ceil((float)file_->get_blocklength() / LEAF_NODE_SIZE) * SIZEFLOAT + SIZEINT);
const auto entrySize = SIZEINT;
const auto treeNodesCapacity =
    (file_->get_blocklength() - headerSize - keySize) / entrySize;
const auto leafNodesCount = (int)ceil((double)n / treeNodesCapacity);
const auto bound = leafNodesCount / workerThreadsCount;
```

- 创建一个用于并发控制的锁`lock`，便于互斥操作或访问互斥区。同时，让 tree 指针指向 this 指针。

```
auto lock = std::make_unique<SpinLock>();
auto tree = this;
```

- 定义一种元组类型，其内包含两种数据类型，分别为`int`和`BLeafNode*`。创建了以该元组为元素的优先队列，排序方式为自定义的`compare`类型，构建小顶堆。

```
using Tuple = std::tuple<int, BLeafNode *>;
const auto compare = [](const Tuple &a, const Tuple &b) {
    return std::get<0>(a) > std::get<0>(b);
  };
std::priority_queue<Tuple, std::vector<Tuple>, decltype(compare)> heap(compare);
```

- 消费者线程的工作：

- `consumerThread`是用于接收 `workerThread` 输出的叶子节点 N_i，将它们按顺序连接（即设置左右指针），缓存一定数量的节点之后一次性地写入磁盘。
- 初始化：`processedNodes`是已经完成的节点，并把最后一个叶子结点`lastLeafIndex`的索引号设为-1，最后一个块`lastLeafIndex`的索引号设置为 0。
- 进入 while 循环，while 循环的退出条件是所有叶子节点构建完成。即当前完成的节点数量大于等于叶子结点数量`processedNodes >= leafNodesCount`。
- 首先让线程进入休眠，休眠 10 毫秒，然后进行上锁，并获取堆 heap 的大小，定义 data 的大小，并把`isFirst`设置为真，entryIndex 的值设置为 0。
- 当小顶堆不为空时，则`consumerThread`把小顶堆里面的叶子结点提取出来，并把叶子结点连接成双向链表，同时写入缓存区。
- while 循环的过程中，通过自旋锁来防止竞态条件。while 循环结束后，就释放自旋锁。
- 设置最后一个叶子结点的区块号，连接进双向链表，并写入缓存区。
- 最后将若干个数量的节点一次性写入磁盘，即`lastBlockIndex = tree->file_->write_blocks(data, entryIndex, lastBlockIndex);`。
- 释放内存空间。

```
124   auto consumerThread =
125       std::thread([&lock, &heap, tree, &end_block, &leafNodesCount, n] {
126         int processedNodes = 0;
127         int lastLeafIndex = -1;
128         int lastBlockIndex = 0;  // 这里的情况和 start_block = 1 的原因一样，file_ 里第一个 block 一定是树根节点
129
130         while (processedNodes < leafNodesCount) {
131           std::this_thread::sleep_for(std::chrono::milliseconds(10));
132
133           lock->lock();
134           const auto heapSize = heap.size();
135           char *data = new char[heapSize * tree->file_->get_blocklength()];
136           bool isFirst = true;
137           int entryIndex = 0;
138           BLeafNode *prev;
139           while (!heap.empty()) {
140             auto [leafIndex, leafNode] = heap.top();
141             if (leafIndex != lastLeafIndex + 1) {
142               break;
143             }
144             lastLeafIndex = leafIndex;
145             processedNodes++;
```

```
147                 if (isFirst) {
148                   isFirst = false;
149                   if (lastBlockIndex > 0) {
150                     leafNode->set_left_sibling(lastBlockIndex);
151                   }
152                   leafNode->set_block(lastBlockIndex + entryIndex + 1);
153                   prev = leafNode;
154                 } else {
155                   leafNode->set_block(lastBlockIndex + entryIndex + 1);
156                   leafNode->set_left_sibling(prev->get_block());
157                   prev->set_right_sibling(leafNode->get_block());
158                   prev->write_to_buffer(data + (entryIndex - 1) *
159                                                tree->file_->get_blocklength());
160                   delete prev;
161                   prev = leafNode;
162                 }
163
164                 entryIndex++;
165                 heap.pop();
166               }
167               lock->unlock();
168
169               if (!entryIndex) {
170                 continue;
171               }
172
173               prev->set_block(lastBlockIndex + entryIndex);
174               end_block = prev->get_block();
175               if (processedNodes < leafNodesCount) {
176                 prev->set_right_sibling(lastBlockIndex + entryIndex + 1);
177               }
178               prev->write_to_buffer(data + (entryIndex - 1) *
179                                            tree->file_->get_blocklength());
180               delete prev;
181               prev = nullptr;
182
183               // 这里写入的大小不是 heapSize 而是 entryIndex
184               lastBlockIndex =
185                   tree->file_->write_blocks(data, entryIndex, lastBlockIndex);
186               delete[] data;
187               data = nullptr;
188             }
189           });
```

Worker 线程的工作：

- 每个`workerThread`按照其下标顺序构建叶子节点。其中`emplace_back`的构造方式是就地构造，不用构造后再次复制到容器中。
- 进入外层 for 循环
- 首先计算出`workerThread`所需要处理的叶节点起始 ID。
- 新构建一个叶子结点` leafNode = new BLeafNode()`，完成初始化后，叶子结点中添加`data.csv`中的数据（存于 table 中），即当该叶子结点未满（没超过叶子结点

的容量时），往其中加入键值对（key 和 id）。叶子结点容量满时即退出内层 for 循环。

- 由于所有叶子结点都要通过小顶堆传送给`consumerThread`，`consumerThread`中需要取出小顶堆中的叶子结点，因此对小顶堆的操作是互斥操作。因此给其上锁。
- 退出 for 循环时，所有叶子结点构建完成。

```
for (int i = 0; i < workerThreadsCount; i++) {
  workerThreads.emplace_back(std::thread(
      [=, &lock, &heap](int id) {
        for (int h = 0; h <= bound; h++) {
          auto leafIndex = id + h * workerThreadsCount;
          if (leafIndex >= leafNodesCount) {
            continue;
          }
          auto leafNode = new BLeafNode();
          leafNode->init(0, tree);
          for (int k = 0; k < treeNodesCapacity; k++) {
            auto index = leafIndex * treeNodesCapacity + k;
            if (index >= n) {
              break;
            }
            auto id = table[index].id_;
            auto key = table[index].key_;
            leafNode->add_new_child(id, key);
            if (k < treeNodesCapacity - 1) {
              assert(!leafNode->isFull());
            }
          }
          lock->lock();
          heap.emplace(std::make_tuple(leafIndex, leafNode));
          lock->unlock();
        }
      },
      i));
}
```

等待所有的线程运行完成，释放内存空间，进入索引节点的构建流程。

```
consumerThread.join();
for (auto &thread : workerThreads) {
  thread.join();
}
workerThreads.clear();
load_index_layers(start_block, end_block);
```

2. 并行构建索引结点

- 初始化，因为是索引节点，因此此时的层数是从 1 开始，即令`current_level`为 1。

```
int current_level = 1;               // current level (leaf level is 0)
int last_start_block = start_block;  // build b-tree level by level
int last_end_block = end_block;      // build b-tree level by level
```

- 同叶子结点。

```
const auto workerThreadsCount = std::thread::hardware_concurrency() - 1;
// const auto workerThreadsCount = 1;
assert(workerThreadsCount >= 1);
const auto headerSize = SIZECHAR + SIZEINT * 3;
const auto entrySize = SIZEFLOAT + SIZEINT;
// 一个 index node 的容量
const auto nodeCapacity = (file_->get_blocklength() - headerSize) / entrySize;
```

- 进入 while 循环，while 循环内进行索引节点的分层构建，其中 while 循环的结束条件为：最后一个结束的块与最后一个开始块的块号相同。
  - 在 while 循环内，首先计算出需要扫描的 block 总数和需要构建的 index node 总数。

    ```
    // 这一层需要扫描的 block 总数
    const auto totalBlocksCount = last_end_block - last_start_block + 1;
    // 这一层要构建的 index node 总数
    const auto todoNodesCount =
            (int)ceil((double)totalBlocksCount / nodeCapacity)
    ```

  - 创建锁、元组 tuple、线程等，同叶子结点。

    ```
    auto lock = std::make_unique<SpinLock>();
        auto tree = this;
        auto currentData = std::make_tuple<int, char *>(-1, nullptr);

        using Tuple = std::tuple<int, BIndexNode *>;
        const auto compare = [](const Tuple &a, const Tuple &b) {
          return std::get<0>(a) > std::get<0>(b);
        };
        std::priority_queue<Tuple, std::vector<Tuple>, decltype(compare)> heap(
            compare);

        std::vector<std::thread> workerThreads;
    ```

  - `composerThread`线程进行工作（详细见下）。
  - `workerThread`线程进行工作（详细见下）。
  - 等待所有的线程运行完成，释放内存空间，更新信息，更新当前层数。

```
composerThread.join();
for (auto &thread : workerThreads) {
  thread.join();
}

workerThreads.clear();

last_start_block = start_block;  // update info
last_end_block = end_block;      // build b-tree of higher level
++current_level;
```

- 更新根节点。

```
root_ = last_start_block;  // update the <root>
```

消费者线程`composerThread`

- `composerThread`是用于接收 `workerThread` 输出的索引节点 N_i，将它们按顺序连接（即设置左右指针），缓存一定数量的节点之后一次性地写入磁盘。
- 初始化：已加载的 block 的数量设为 0，以构建好的节点数量设为 0，最后一个索引号为-1，令最后一个 block 的块号为 end_block 的块号（从并行构建叶子结点的函数传入并行构建索引节点函数）。
- 进入 while 循环，while 循环的退出条件是索引节点构建完成。即当前完成的节点数量大于等于需要完成的索引结点数量`processedNodes >= todoNodesCount`.
  - 首先让线程进入休眠，休眠 50 毫秒，然后进行上锁，并获取堆 heap 的大小，定义 data 的大小，并把`isFirst`设置为真，entryIndex 的值设置为 0。
  - 当小顶堆不为空时，则`consumerThread`把小顶堆里面的索引结点提取出来，并把索引结点连接成双向链表，同时写入缓存区。
  - while 循环的过程中，通过自旋锁来防止竞态条件。while 循环结束后，就释放自旋锁。
  - 设置最后一个叶子结点的区块号，连接进双向链表，并写入缓存区。
  - 最后将若干个数量的节点一次性写入磁盘，即`lastBlockIndex = tree->file_->write_blocks(data, entryIndex, lastBlockIndex);`。
  - 释放内存空间，重新计算已完成的索引节点数目。
- 在这个工程中，对小顶堆的操作需上锁，使用完需释放锁。

```
266     auto composerThread = std::thread([=, &lock, &heap, &todoNodesCount,
267                                        &start_block, &end_block, &currentData] {
268       int loadedBlocksCount = 0;
269       int processedNodesCount = 0;
270       int lastIndex = -1;
271       int lastBlockIndex = last_end_block;
```

```
273       while (processedNodesCount < todoNodesCount) {
274         const auto loadedCount =
275             std::min(nodeCapacity * 10000, totalBlocksCount - loadedBlocksCount);
276         loadedBlocksCount += loadedCount;
277         char *data = new char[tree->file_->get_blocklength() * loadedCount];
278         assert(tree->file_->read_blocks(data,
279                                         last_start_block + processedNodesCount,
280                                         loadedCount) == true);
281         lock->lock();
282         currentData = {loadedCount, data};
283         lock->unlock();
284
285         auto processedCountOfCurrentRun = 0;
286         const auto indexNodesCountOfCurrentRun =
287             (int)ceil((double)loadedCount / nodeCapacity);
288         while (processedCountOfCurrentRun < indexNodesCountOfCurrentRun) {
289           std::this_thread::sleep_for(std::chrono::milliseconds(50));
290
291           lock->lock();
292           const auto heapSize = heap.size();
293           char *data = new char[heapSize * tree->file_->get_blocklength()];
294           bool isFirst = true;
295           int entryIndex = 0;
296           BIndexNode *prev;
297           while (!heap.empty()) {
298             auto [indexNodeIndex, indexNode] = heap.top();
299             if (indexNodeIndex != lastIndex + 1) {
300               break;
301             }
302             lastIndex = indexNodeIndex;
303             processedCountOfCurrentRun++;
304
```

```
305             if (isFirst) {
306               isFirst = false;
307               indexNode->set_block(lastBlockIndex + entryIndex + 1);
308               if (lastBlockIndex > last_end_block) {
309                 indexNode->set_left_sibling(
310                     lastBlockIndex);  // 不是第一个 block，直接和上一个
311                                       // block 相连
312               } else {
313                 start_block =
314                     indexNode->get_block();  // 这是这个索引层的第一个 block
315               }
316               prev = indexNode;
317             } else {
318               indexNode->set_block(lastBlockIndex + entryIndex + 1);
319               indexNode->set_left_sibling(prev->get_block());
320               prev->set_right_sibling(indexNode->get_block());
321               prev->write_to_buffer(data + (entryIndex - 1) *
322                                            tree->file_->get_blocklength());
323               delete prev;
324               prev = indexNode;
325             }
326
327             entryIndex++;
328             heap.pop();
329           }
```

```
330             lock->unlock();
331
332             if (!entryIndex) {
333               continue;
334             }
335
336             prev->set_block(lastBlockIndex + entryIndex);
337             end_block = prev->get_block();  // 更新 end_block 指针
338             if (processedNodesCount + processedCountOfCurrentRun <
339                 todoNodesCount) {
340               prev->set_right_sibling(lastBlockIndex + entryIndex + 1);
341             }
342             prev->write_to_buffer(data + (entryIndex - 1) *
343                                   tree->file_->get_blocklength());
344             delete prev;
345             prev = nullptr;
346
347             // 这里写入的大小不是 heapSize 而是 entryIndex
348             lastBlockIndex =
349                 tree->file_->write_blocks(data, entryIndex, lastBlockIndex);
350             delete[] data;
351             data = nullptr;
352           }
353
354           // 这一轮读取结束
355           processedNodesCount += processedCountOfCurrentRun;
356         }
357
358         lock->lock();
359         currentData = std::make_tuple<int, char *>(-1, nullptr);
360         lock->unlock();
361       });
```

Worker 线程的工作：

- 每个`workerThread`按照其下标顺序构建索引节点。其中`emplace_back`的构造方式是就地构造，不用构造后再次复制到容器中。
- 初始化，对`blocksCountOfCurrentRun`和`data`进行初始化。
- 进入 while 外层循环。只能通过 break 退出循环
  - 进入 while 内层循环
    - 首先让 workerThread 休眠 5 毫秒。
    - 上锁，因为要对堆`currentData`进行操作，读出堆中的内容。
    - 释放锁。
  - 计算当前轮次的索引节点的数量，以及当前轮次索引节点的数量与`workThreads`的数量，存于`bound`中。
  - 进入 for 循环。
    - for 循环内，首先计算出`workerThread`自身需要处理的`block`的部分，并据此构建出索引结点。

- 新构建一个索引结点`indexNode = new BIndexNode()`，初始化，但是先不写入磁盘中。原因：索引节点在构建时需要获得儿子节点的 key，这就需要从磁盘读取信息。从整体来看，构建索引节点的过程会是多次的读—写—读循环，对硬盘的利用并不高效，因此我们修改了`b_node.h `和`b_node.c`文件（对`BIndexNode`的类进行了修改）。如下所示，新增了函数。

```
virtual void init_no_write(
  int   level,
  BTree *btree);

virtual void init_restore_in_place(
  BTree *btree,
  int block,
  Block data);

void add_new_child_no_dirty(
  float key,
  int son);
```

```
void BIndexNode::init_no_write(int level, BTree *btree) {
  btree_ = btree;
  level_ = (char)level;
  num_entries_ = 0;
  left_sibling_ = -1;
  right_sibling_ = -1;
  dirty_ = false;

  // page size B
  int b_length = btree_->file_->get_blocklength();
  capacity_ = (b_length - get_header_size()) / get_entry_size();
  if (capacity_ < 50) {  // ensure at least 50 entries
    printf("capacity = %d, which is too small.\n", capacity_);
    exit(1);
  }

  key_ = new float[capacity_];
  son_ = new int[capacity_];
  //分配内存
  memset(key_, MINREAL, capacity_ * SIZEFLOAT);
  memset(son_, -1, capacity_ * SIZEINT);
}
```

- 计算 block 号，其中 block 号为`start_block`的值与`index`的值的和。
- 根据当前层数不同（因为如果层数为 1，需要连接叶子结点；其余层数不需要），调用自己编写的`init_restore_in_place`函数，一次性顺序读取大量的儿子节点到内存中。

```
void BIndexNode::init_restore_in_place(BTree *btree, int block, Block data) {
  btree_ = btree;
  block_ = block;
  dirty_ = false;

  int b_len = btree_->file_->get_blocklength();
  capacity_ = (b_len - get_header_size()) / get_entry_size();
  if (capacity_ < 50) {
    printf("capacity = %d, which is too small.\n", capacity_);
    exit(1);
  }

  key_ = new float[capacity_];
  son_ = new int[capacity_];
  memset(key_, MINREAL, capacity_ * SIZEFLOAT);
  memset(son_, -1, capacity_ * SIZEINT);

  read_from_buffer(data);
}
```

- 往当前索引节点增加儿子节点，使用自己编写的函数 `add_new_child_no_dirty`。

```
void BIndexNode::add_new_child_no_dirty(
  float key,
  int   son)
{
  key_[num_entries_] = key;
  son_[num_entries_] = son;
  ++num_entries_;
}
```

- 退出内层 for 循环。由于所有素音结点都要通过小顶堆传送给 `composerThread`，`composerThread`中需要取出小顶堆中的索引结点，因此对小顶堆的操作是互斥操作。因此给其上锁。

- 退出 for 循环时，所有索引结点构建完成。

```
363     for (int i = 0; i < workerThreadsCount; i++) {
364       workerThreads.emplace_back(std::thread(
365           [=, &currentData, &lock, &heap, &current_level, &end_block](int id) {
366             int blocksCountOfCurrentRun = -1;
367             char *data = nullptr;
368             while (true) {
369               while (true) {
370                 std::this_thread::sleep_for(std::chrono::milliseconds(5));
371                 lock->lock();
372                 if (std::get<1>(currentData) != data) {
373                   if (data != nullptr && std::get<1>(currentData) == nullptr) {
374                     lock->unlock();
375                     return;
376                   }
377                   blocksCountOfCurrentRun = std::get<0>(currentData);
378                   data = std::get<1>(currentData);
379                   lock->unlock();
380                   break;
381                 }
382                 lock->unlock();
383               }
```

```
385                 const auto indexNodesCountOfCurrentRun =
386                     (int)ceil((double)blocksCountOfCurrentRun / nodeCapacity);
387                 const auto bound =
388                     indexNodesCountOfCurrentRun / workerThreadsCount;
389                 for (int h = 0; h <= bound; h++) {
390                   // 这个 index 是 composerThread 给的这一波 totalCount 个 block
391                   // 数据里的相对顺序 接下来这个 workerThread
392                   // 会算出自己应该处理哪一部分的block，构建出对应的 index node
393                   auto indexNodeIndex = id + h * workerThreadsCount;
394                   if (indexNodeIndex >= indexNodesCountOfCurrentRun) {
395                     continue;
396                   }
397                   auto indexNode = new BIndexNode();
398                   indexNode->init_no_write(current_level, tree);
399                   for (int k = 0; k < nodeCapacity; k++) {
400                     auto index = indexNodeIndex * nodeCapacity + k;
401                     if (index >= blocksCountOfCurrentRun) {
402                       break;
403                     }
404                     float key;
405                     auto block =
406                         start_block + index;  // 此时就已经可以知道 block 号了
407                     if (current_level == 1) {
408                       BLeafNode node;
409                       node.init_restore_in_place(
410                           tree, block,
411                           data + tree->file_->get_blocklength() * index);
412                       key = node.get_key_of_node();
413                     } else {
414                       BIndexNode node;
415                       node.init_restore_in_place(
416                           tree, block,
417                           data + tree->file_->get_blocklength() * index);
418                       key = node.get_key_of_node();
419                     }
420                     indexNode->add_new_child_no_dirty(key, block);
421                     if (k < nodeCapacity - 1) {
422                       assert(!indexNode->isFull());
423                     }
424                   }
425                   lock->lock();
426                   heap.emplace(std::make_tuple(indexNodeIndex, indexNode));
427                   lock->unlock();
428                 }
429               }
430             },
431             i));
432     }
```

五、实验结果

1. 测试环境

产生下列运行结果的测试环境是：

i5-8500B CPU 上的 Ubuntu 21.10 虚拟机，拥有 4 个 CPU 核心以及 4 GB 内存。硬盘使用 NVMe 固态硬盘。

2. 并行构建叶子结点

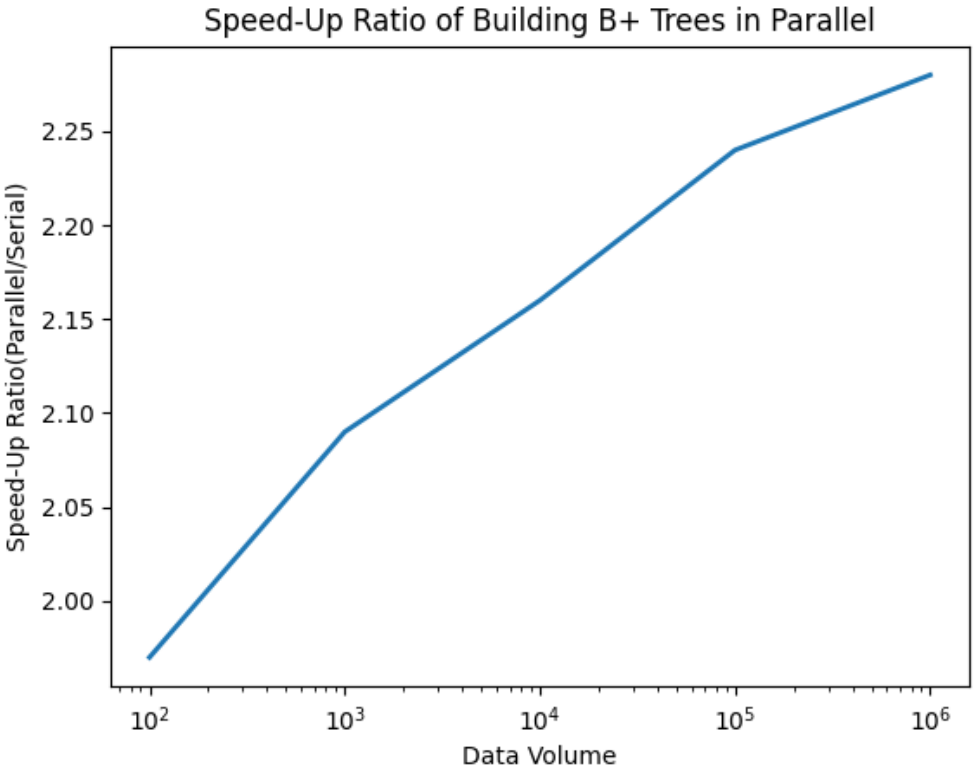
在完成第一步之后，即并行构建叶子节点，但仍然串行构建索引节点，多次对 `data/dataset.csv` 运行原本的程序以及修改后的程序，结果如下。数据集规模为 1M 行。

```
$ ./make_hex.sh
data_file   = ./data/dataset.csv
tree_file   = ./result/B_tree
运行时间: 0.085063  s
data_file   = ./data/dataset.csv
tree_file   = ./result/B_tree
运行时间: 0.146649  s
$ ./make_hex.sh
data_file   = ./data/dataset.csv
tree_file   = ./result/B_tree
运行时间: 0.078678  s
data_file   = ./data/dataset.csv
tree_file   = ./result/B_tree
运行时间: 0.132158  s
$ ./make_hex.sh
data_file   = ./data/dataset.csv
tree_file   = ./result/B_tree
运行时间: 0.059390  s
data_file   = ./data/dataset.csv
tree_file   = ./result/B_tree
运行时间: 0.144739  s
$ ./make_hex.sh
data_file   = ./data/dataset.csv
tree_file   = ./result/B_tree
运行时间: 0.068712  s
data_file   = ./data/dataset.csv
tree_file   = ./result/B_tree
运行时间: 0.161334  s
```

可以看到，此时程序的加速比在 2～3 之间。我们认为如果在机械硬盘环境下测试会有更高的加速比。

3. 并行构建索引结点

在将索引节点也进行并行化之后，多次对 `data/dataset.csv` 针对不同规模的测试数据集运行原本的程序以及修改后的程序，得到的加速比情况如下：



六、实验分析

对比串行和并行代码得到的 B 树二进制文件，`diff` 程序仅输出了上面的不同。可以看到，这几行都是位于文件的末尾，且根据 `b_node.cc` 中相关数据结构的定义，我们发现它们都没有代表实际数据（也就是 block 中的闲置位），两个代码中的这些位在使用 `new char[]` 也即 `malloc()` 之后并没有进行清零，所以导致了输出的不一致。排除掉此因素之后，我们认为二者输出的文件是一致的。

如下所示：

```
892    < 004512b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
893    < 004512c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
894    < 004512d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
895    < 004512e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
896    < 004512f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
897    < 00451300: 0000 0000 0000 0000 0000 0000 0000 0000  ................
898    < 00451310: 0000 0000 0000 0000 0000 0000 0000 0000  ................
899    < 00451320: 0000 0000 0000 0000 0000 0000 0000 0000  ................
900    ---
```

```
> 004512b0: 8070 0000 0000 0000 6400 0000 0000 0000  .p......d.......
> 004512c0: 223b d79e ea7f 0000 0000 0000 7300 0000  ";..........s...
> 004512d0: 5410 0000 5610 0000 0000 0000 0000 0000  T...V...........
> 004512e0: 0100 0000 5510 0000 7300 0000 0000 0000  ....U...s.......
> 004512f0: 0000 0000 0000 0000 0800 0000 0000 0000  ................
> 00451300: 0000 0000 0000 0000 0800 0000 0000 0000  ................
> 00451310: 0000 0000 0000 0000 b110 0000 0000 0000  ................
> 00451320: 4013 0060 ed7f 0000 8000 0060 ed7f 0000  @..`.......`....
282933,282934c282933,282934
< 00451340: 0000 0000 0000 0000 0000 0000 0000 0000  ................
< 00451350: 0000 0000 0000 0000 0000 0000 0000 0000  ................
---
> 00451340: 0000 0000 0000 0000 e101 0000 0000 0000  ................
> 00451350: a013 0060 ed7f 0000 1043 0160 ed7f 0000  ...`.....C.`....
282947,282954c282947,282954
< 00451420: 0088 2200 0000 0000 0078 2200 0000 0000  .."......x".....
< 00451430: 0079 2200 0000 0000 007a 2200 0000 0000  .y"......z".....
< 00451440: 007b 2200 0000 0000 007c 2200 0000 0000  .{"......|".....
< 00451450: 007d 2200 0000 0000 007e 2200 0000 0000  .}"......~".....
< 00451460: 007f 2200 0000 0000 0080 2200 0000 0000  .."........".....
< 00451470: 0081 2200 0000 0000 0082 2200 0000 0000  .."........".....
< 00451480: 0083 2200 0000 0000 0084 2200 0000 0000  .."........".....
< 00451490: 0085 2200 0000 0000 0000 0000 0000 0000  .."............
---
> 00451420: 0088 2200 0000 0000 0000 0000 0000 0000  .."............
> 00451430: 0000 0000 0000 0000 0000 0000 0000 0000  ................
> 00451440: 0000 0000 0000 0000 0000 0000 0000 0000  ................
> 00451450: 0000 0000 0000 0000 0000 0000 0000 0000  ................
> 00451460: 0000 0000 0000 0000 0000 0000 0000 0000  ................
> 00451470: 0000 0000 0000 0000 0000 0000 0000 0000  ................
> 00451480: 0000 0000 0000 0000 0000 0000 0000 0000  ................
> 00451490: 0000 0000 0000 0000 0000 0000 0000 0000  ................
282964,282971c282964,282971
< 00451530: 0000 0000 0000 0000 0000 0000 0000 0000  ................
< 00451540: 0000 0000 0000 0000 0000 0000 0000 0000  ................
< 00451550: 0000 0000 0000 0000 0000 0000 0000 0000  ................
< 00451560: 0000 0000 0000 0000 0000 0000 0000 0000  ................
< 00451570: 0000 0000 0000 0000 0000 0000 0000 0000  ................
< 00451580: 0000 0000 0000 0000 0000 0000 0000 0000  ................
< 00451590: 0000 0000 0000 0000 0000 0000 0000 0000  ................
< 004515a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
---
```

```
943    > 00451530: e001 0000 0000 0000 6400 0000 0000 0000  ........d.......
944    > 00451540: f119 d5ae ea7f 0000 0000 0000 7300 0000  ............s...
945    > 00451550: 3b1c 0000 3d1c 0000 0000 0000 0000 0000  ;...=...........
946    > 00451560: 0100 0000 3c1c 0000 7300 0000 0000 0000  ....<...s.......
947    > 00451570: 0000 0000 0000 0000 0800 0000 0000 0000  ................
948    > 00451580: 0000 0000 0000 0000 0800 0000 0000 0000  ................
949    > 00451590: 0000 0000 0000 0000 f106 0000 0000 0000  ................
950    > 004515a0: 702d 0650 ed7f 0000 8000 0050 ed7f 0000  p-.P.......P....
951    282973,282974c282973,282974
952    < 004515c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
953    < 004515d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
954    ---
955    > 004515c0: 0000 0000 0000 0000 e101 0000 0000 0000  ................
956    > 004515d0: f076 0350 ed7f 0000 801a 0050 ed7f 0000  .v.P.......P....
```

七、性能调优与创新优化

在设计思路中提到，每当一个叶子节点使用 `BLeafNode::init()` 进行初始化操作时，都会调用到 `BlockFile::append_block()` 方法。而从它的函数体中也可以看见每调用一次 `BlockFile::append_block()` 方法就需要进行两次 `fseek()` 以及一次 `fwrite_number()` 的操作，这些操作将会在 B+ 树的构建过程中产生大量的随机 IO。

```
/*
  block_file.cc
*/
int BlockFile::append_block(    // append new block at the end of file
  Block block)            // the new block
{
  fseek(fp_, 0, SEEK_END);    // <fp_> point to the end of file
  put_bytes(block, block_length_);// write a <block>
  ++num_blocks_;          // add 1 to <num_blocks_>

  fseek(fp_, SIZEINT, SEEK_SET);  // <fp_> point to pos of header
  fwrite_number(num_blocks_);   // update <num_blocks_>

  // ------------------------------------------------------------------------
  //  <fp_> point to the pos of new added block.
  //  the equation <act_block_> = <num_blocks_> indicates the file pointer
  //  point to new added block.
  //  return index of new added block
  // ------------------------------------------------------------------------
  fseek(fp_, -block_length_, SEEK_END);
  return (act_block_ = num_blocks_) - 1;
}
```

对于固态硬盘来说，随机 IO 所带来的影响并不会特别的明显，但对于机械硬盘来说，大量的随机 IO 也就意味着大量的时间将会被浪费用于定位目标页所在的位置，从而造成严重的性能损失。像数据库这种需要大容量数据存储的服务，相对便宜的机械硬

盘才是主力存储设备。考虑机械硬盘顺序 IO 比随机 IO 速度快的特性，我们决定将上述的操作进行修改与优化。

首先修改 `BLeafNode::init()` 函数。直接将末尾处的 `BlockFile::append_block()`，即意味着当进行叶子节点的初始化操作时，将不再立即把当前叶子节点写入到硬盘中。在叶子节点的构建过程中，我们令 `consumerThread` 不停获取初始化完成的叶子节点，当叶子节点的数量占满一个块的时候才进行写出的操作。经过以上操作，即可成功地将大量的随机 IO 操作变为一次顺序 IO 操作。

```
/*
  b_node.cc
  改写 BLeafNode::init() 方法，使得初始化叶子节点不会进行写出操作
*/
void BLeafNode::init(       // init a new node, which not exist
  int   level,            // level (depth) in b-tree
  BTree *btree)           // b-tree of this node
{
  btree_         = btree;
  level_         = (char) level;

  num_entries_   = 0;
  num_keys_      = 0;
  left_sibling_  = -1;
  right_sibling_ = -1;
  dirty_         = true;

  // ------------------------------------------------------------------------
  //  init <capacity_keys_> and calc key size
  // ------------------------------------------------------------------------
  //page size B
  int b_length = btree_->file_->get_blocklength();
  int key_size = get_key_size(b_length);

  key_ = new float[capacity_keys_];
  memset(key_, MINREAL, capacity_keys_ * SIZEFLOAT);

  int header_size = get_header_size();
  int entry_size = get_entry_size();

  capacity_ = (b_length - header_size - key_size) / entry_size;
  if (capacity_ < 100) {      // at least 100 entries
    printf("capacity = %d, which is too small.\n", capacity_);
    exit(1);
  }
  id_ = new int[capacity_];
  memset(id_, -1, capacity_ * SIZEINT);
}
```

```
/*
  b_node.cc
  为 BIndexNode 添加不进行写出操作的初始化方法
*/
void BIndexNode::init_no_write(int level, BTree *btree) {
  btree_ = btree;
  level_ = (char)level;
  num_entries_ = 0;
  left_sibling_ = -1;
  right_sibling_ = -1;
  dirty_ = false;

  // page size B
  int b_length = btree_->file_->get_blocklength();
  capacity_ = (b_length - get_header_size()) / get_entry_size();
  if (capacity_ < 50) {  // ensure at least 50 entries
    printf("capacity = %d, which is too small.\n", capacity_);
    exit(1);
  }

  key_ = new float[capacity_];
  son_ = new int[capacity_];
  //分配内存
  memset(key_, MINREAL, capacity_ * SIZEFLOAT);
  memset(son_, -1, capacity_ * SIZEINT);
}
```

在进行索引节点的构建过程中，我们需要使用到当前层数所对应的前一层节点，于是会使用到`init_restore()` 方法将前一层节点从硬盘中读取到内存进行构建的操作。从代码中不难发现 `BIndexNode::init_restore` 和 `BLeafNode::init_restore` 也存在着 IO 操作。由于我们的并行实现中将数据写入到了缓冲区中，当需要前一层节点的信息时，可以直接从缓冲区中读取数据。因此为 `BIndexNode` 和 `BLeafNode` 新增 `init_restore_in_place()` 方法，从缓冲区中读取数据，无需再进行 IO 操作。

```
/*
  b_node.cc
  一次读取整个索引节点块
*/
void BIndexNode::init_restore_in_place(BTree *btree, int block, Block data) {
  btree_ = btree;
  block_ = block;
  dirty_ = false;

  int b_len = btree_->file_->get_blocklength();
  capacity_ = (b_len - get_header_size()) / get_entry_size();
  if (capacity_ < 50) {
    printf("capacity = %d, which is too small.\n", capacity_);
    exit(1);
  }

  key_ = new float[capacity_];
  son_ = new int[capacity_];
  memset(key_, MINREAL, capacity_ * SIZEFLOAT);
  memset(son_, -1, capacity_ * SIZEINT);

  read_from_buffer(data);
}

/*
  b_node.cc
  一次读取整个叶子节点块
*/
void BLeafNode::init_restore_in_place(BTree *btree, int block, Block data) {
  btree_ = btree;
  block_ = block;
  dirty_ = false;

  int b_length = btree_->file_->get_blocklength();
  int key_size = get_key_size(b_length);

  key_ = new float[capacity_keys_];
  memset(key_, MINREAL, capacity_keys_ * SIZEFLOAT);

  int header_size = get_header_size();
  int entry_size = get_entry_size();

  capacity_ = (b_length - header_size - key_size) / entry_size;
  if (capacity_ < 100) {
    printf("capacity = %d, which is too small.\n", capacity_);
    exit(1);
  }
  id_ = new int[capacity_];
  memset(id_, -1, capacity_ * SIZEINT);

  read_from_buffer(data);
}
```