

数据库系统课程设计报告

B+树 BulkLoading 多核并行设计

小组分工

学号	姓名
19309049	黄皓佳
19335008	曾家洋
19335018	陈俊熹
19335054	何杰
19335286	郑有为（组长）

目录

目录

I. B+树 BulkLoading 过程理解

- 1.1 BulkLoading 的基本思想
- 1.2 结合实现理解 BulkLoading
- 1.3 BulkLoading 的利弊

II. 并行设计思路

- 2.1 基本想法
- 2.2 并行思路图解
- 2.3 更多讨论

III. 算法流程图

IV. 创新优化思路

- 4.1 连续分配块
- 4.2 连续写磁盘

V. 关键代码描述

- 5.1 `parallelBulkLoad`
- 5.2 `batchLoadLeaf` 和 `batchLoadIndex`
- 5.3 `append_blocks`、`init_noalloc` 和 `alloc_blocks`
- 5.4 `write_blocks`、`write_leaf_blocks` 和 `write_index_blocks`
- 5.5 `write_B_tree`

VI. 实验结果分析

- 6.0 并行和性能调优的代码说明
 - 6.0.1 改为非连续写磁盘的方式
 - 6.0.2 改为非连续分配块的方式
 - 6.0.3 如何测试代码
- 6.1 并行实验结果代码及分析
 - 6.1.1 影响因素
 - 6.1.2 实验方法
 - 6.1.3 数据项的数目 `int n`
 - 6.1.4 线程数 `int threadNum`
 - 6.1.5 Buffer的大小 `int maxBufferBlock`
- 6.2 性能调优和创新优化实验结果及分析
 - 6.2.1 性能调优的原因
 - 6.2.2 数据量的大小

I . B+树 BulkLoading 过程理解

1.1 BulkLoading 的基本思想

在理解 BulkLoading 过程之前，要先理解 B+ 树的结构特性：首先，B+ 树只在叶子节点中保存数据，非叶子节点只保存索引，叶子节点的数据又根据关键字从小到大顺序排序。

我们单看叶子节点一层，就可以发现起始它和一个有序的数据数组在形状上是一致的。这意味着如果我们的现有数据是有序排列的，我们就可以直接从小到大“铺”到叶子节点上。再加上B+树的子节点的个数有限制，故我们可以直接估算出要多少个叶子节点，多少个相邻的叶子节点有一个共同的父索引节点，把上述过程同样运用到索引节点上，我们就可以自底向上地构建出一棵完整的B+树。

得到一个BulkLoading过程的一个完整描述：**先从底层叶子节点构建，从左往右按顺序构建一个双向链表；从下往上，一层层构建索引节点，每一层也是从左往右构建索引节点。**

1.2 结合实现理解 BulkLoading

讨论 BulkLoading 的实现细节：节点的结构、磁盘文件的数据对应。

- 叶子节点：相邻节点之间有互相指向的指针，叶子层构成了一个双向链表。每个节点有两个数组，一个储存Key，一个储存Value，一个Key对应16个Value，这里的Value就是实际的数据，也就是数据项（Data Entry）。
- 索引节点：每一层的相邻的索引节点也存在双向指针，索引节点都是根据下一层的子节点生成的，每个索引节点有两个数组，一个存储Key，一个存储指向子节点的指针，二者数量相等。一个键值Key和一个指针就构成了一个索引项（Index Entry）。
- 与磁盘文件的关系：磁盘文件被划分为一个个块（Block），每一个节点就对应一个块，块通过块号区分，表示节点存储到文件的区域编号。这样，索引项中所谓的指针，就可以用块号来表示。

这样，BulkLoading 的过程就包含：在内存中组织叶子节点和索引节点，组织的方式就是创建与块一样大的缓冲区，将数据项、索引项拷贝进去，然后再写入到磁盘块里。（区分这些过程有利于并行实现）

因为块号的分配是连续的，块的大小是固定的。在 BulkLoading 过程中，我们只需要记录下一层中第一个块号和最后一个块号，就可以直接算出上一层节点的所有子节点的块号，无需访问子节点。

在B+树中一个节点的索引值可以设置为第一个项的键值，因为 BulkLoading 是自底向上的构造过程，节点的索引值可以直接给出，且过程中不会改变，不需要B+树插入删除节点时的那样进行动态调整。在设计上采用 On-the-fly 的策略，在一层一层循环构建的过程中，线程函数每次都保存一层的键值并返回，键值数组再用于构建上一层节点，无需进行IO访问。

1.3 BulkLoading 的利弊

相比于通过插入节点构造B+树，BulkLoading充分利用了数据有序的特性，让每一个节点构造过程的复杂度都为 $O(1)$ ，构造过程中不存在节点的分叉、合并等调整，生成B+树的过程非常快。

通过BulkLoading构造B+树也可能存在潜在的问题：因为每一个节点从左到右都是尽可能填满的，当需要插入新节点时，B+树结构的变动可能会非常大，会产生较大的时间开销。

或许我们可以加入一个填充因子来控制BulkLoading每次为一个叶子节点填出的数据项的个数，让叶子节点的数据项不是满的，这样可以减弱插入节点对B+树结构造成的影响。

II. 并行设计思路

2.1 基本想法

首先，需要思考的问题是在批量加载的哪些步骤可以并行，通过阅读串行代码我们可知，构建B+树的串行过程大致为：读取数据并在内存中创建一个节点对象存放数据，然后再通过系统函数 `fwrite()` 将该节点中的数据写入到磁盘中。

我们考虑到，节点数据写到文件的过程由于文件指针的限制无法并行，同一时刻只能有一个线程写磁盘，因此可以创建内存缓冲区缓冲节点数据，在获取写文件权限后将内存中节点写出。

由于写磁盘的速度低于写内存，BulkLoading 运行速度的瓶颈可能在于写硬盘，因此要充分提高CPU和磁盘IO的并行度，来减少IO造成的护航效应。

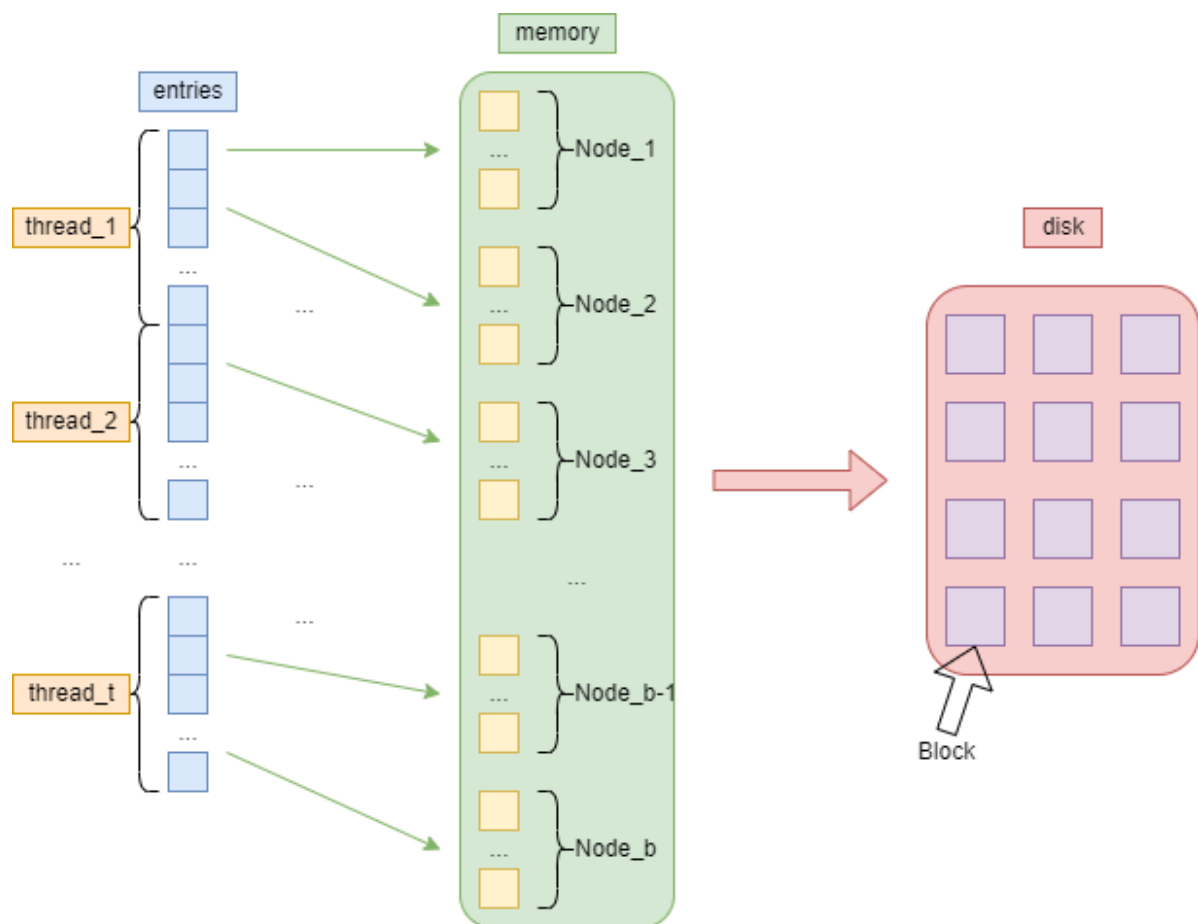
引入互斥锁：由于写硬盘不能并行，因此需要一个互斥锁保证同一时刻只有一个线程能够将已经加载到内存的节点数据写入硬盘。那么对于其余抢不到锁的线程，我们有三种处理思路：

- 一个线程在内存中写完一个节点就等待直到获取锁（堵塞）；
- 一个线程在内存中写完所有待处理的节点再争取锁，等待直到获取锁（堵塞）；
- 一个节点每向内存写一个节点就尝试获取锁，如果获取失败就将块加入一个队列，如果获取成功就磁盘写队列内所有的节点（非堵塞）。

我们采取的是类似于第三种思路的并行思路：一个节点每向内存写完 MIN_BLOCK 个节点就尝试获取锁，如果获取失败就将块加入一个长为 MAX_BLOCK 的队列，如果获取成功就磁盘写队列内所有的节点（非堵塞），如果此时队列已满就进入堵塞状态。其中 MIN_BLOCK 和 MAX_BLOCK 是两个可调的参数。

最后，在我们的并行 BulkLoading 过程中，主线程会为每一层创建若干个线程，并且只有一层的所有线程执行完工作后，返回记录线程产生的一批节点键值的数组，才会开始创建上一层的线程。

2.2 并行思路图解



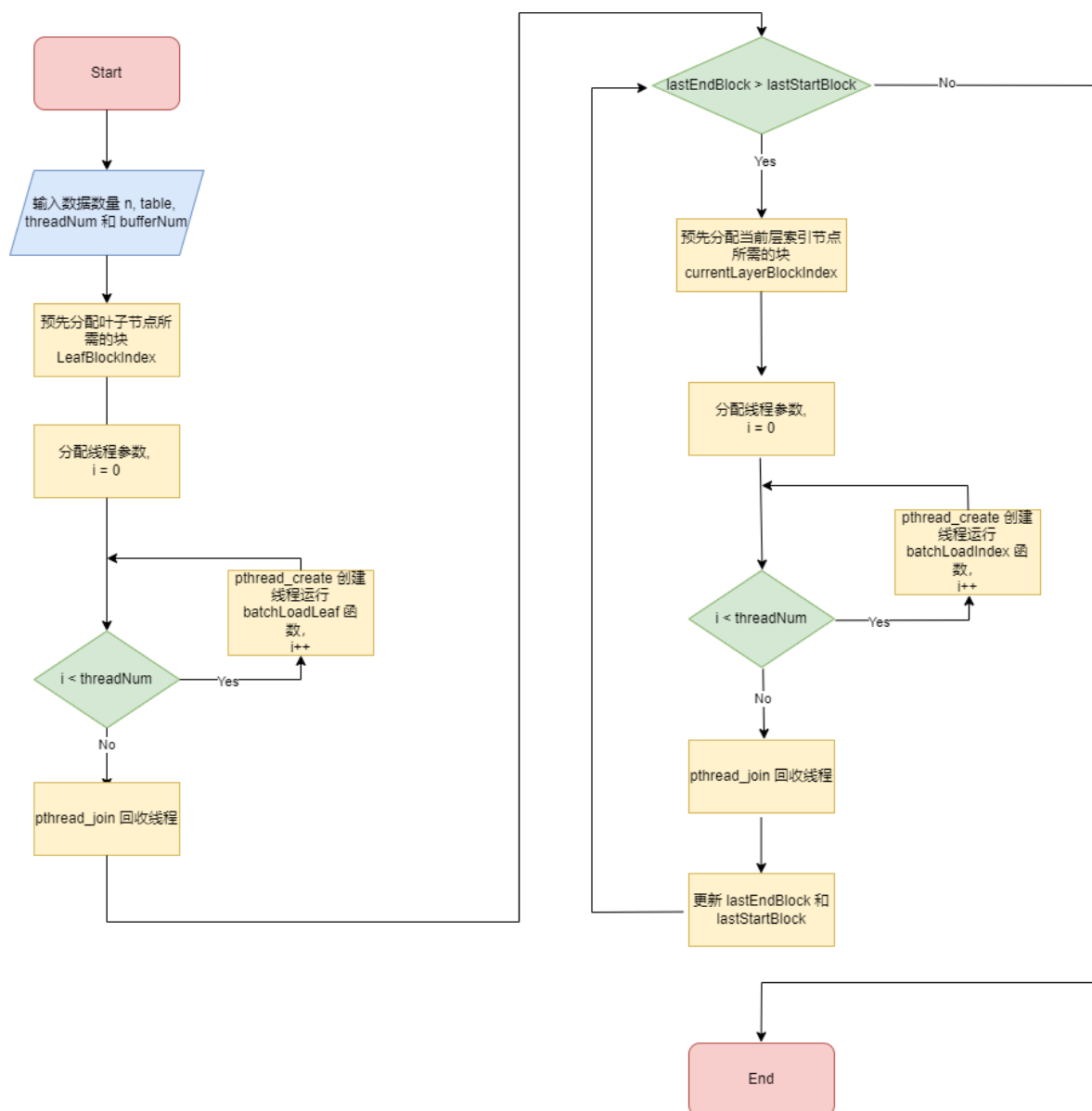
共有 t 个线程， n 个entries，叶子节点有 b 个，
每个线程将自己负责的entries写入一定数量的节点中，
每个节点写满时，就申请写入硬盘，与硬盘通过一个锁来控制，
索引节点同理。

2.3 更多讨论

因为我们只在向内存写节点这个过程进行并行实现，而分配块号的过程是串行的，串行申请块号意味着我们无需面临块号分配不连续的问题。同时，我们每次都尽可能填满一个节点，所以也不会出现中间的叶子节点数据不满的情况。

通过我们的并行算法生成的B+树和串行生成的B+树是完全一样的，因此测试算法正确性相对简单。

III. 算法流程图



IV. 创新优化思路

4.1 连续分配块

我们分离了申请块号和节点初始化两个过程，在串行阶段，我们预先使用 `alloc_blocks` 申请一层所需要的所有块号，再在并行阶段节点创建时调用 `init_noalloc` 而不是原 `init` 函数，此阶段无需访问磁盘申请块号。

这样做有两个好处：一是避免了多个节点并行申请块号，访问磁盘空间造成的错误，二是我们可以通过优化 `alloc_blocks` 过程来提高速度。

我们对申请块号的优化利用了块号连续的条件，通过修改底层，增加一次申请多个块的函数 `append_blocks`，让一次申请多个连续块号只用调用一次 `fwrite`，而不是每申请一个块就调用一次 `fwrite`。通过这样的方法可以有效地减少 `fwrite` 的调用开销，提高运行速度。

相关的函数声明如下，我们在5.3节详细描述了这些函数的实现。

```

1 // @file: block_file.h
2 // @line: 116-118
3 int *append_blocks( // append new blocks at the end of file
4     Block block,    // the new blocks
5     int n);
6 
```

```

7 // @file: b_node.h
8 // @line: 36-40
9 virtual void init_noalloc( // init a node in memory without allocating a
    block
10     int level,           // level in b-tree
11     BTree *btree,        // b-tree of this node
12     int block            // the index of the block that pre-allocated for this
    node
13 );
14
15 // @file: b_tree.h
16 // @line: 92-93
17 int *alloc_blocks( // allocate n empty blocks in the block file
18     int n);         // returns an array of indexes of blocks that were
    allocated

```

4.2 连续写磁盘

写入磁盘的过程和像磁盘申请块号这个过程很类似，而在原代码中，节点的磁盘写入实在析构函数中实现的，即当脏位被设置为真时，析构函数回将节点内容写回磁盘。我们同样将写入磁盘这个该过程分离出来，用一个 `write_blocks` 函数来封装，并在执行该函数后修改节点的脏位，使其无需在析构函数中写回磁盘。

对于并行BulkLoading，每个线程负责将内容写入节点，当保存完成的节点的队列中结点的数目到达一定数目时，我们调用 `write_leaf_blocks` 或 `write_index_blocks` 将这些连续的节点的节点内容一次性写回磁盘。

这个过程同样利用了块号连续的特性，减少了调用 `fwrite` 的次数，显著地提高了运行速度。

相关的函数声明如下，我们在5.4节详细描述了这些函数的实现。

```

1 // @file: block_file.h
2 // @line: 105-109
3 bool write_blocks( // write a block <b> in the <pos>
4     Block blocks,   // a series of blocks
5     int firstIndex, // pos of the first block
6     int n);
7
8 // @file: b_tree.h
9 // @line: 96-99
10 void write_leaf_blocks( // write n blocks to the block file
11     BLeafNode * nodes[],
12     int firstBlock,
13     int n);
14
15 // @file: b_tree.h
16 // @line: 102-105
17 void write_index_blocks( // write n blocks to the block file
18     BIndexNode* nodes[],
19     int firstBlock,
20     int n);

```

V. 关键代码描述

5.1 parallelBulkLoad

`parallelBulkLoad` 是并行BulkLoading的创建函数，负责每次完成对一层的连续块号申请，创建和初始化并行线程来执行每一层写节点的任务，自底向上层层构建B+树。

在此之前引入线程函数的参数列表：`BatchLoadLeafArgs` 和 `BatchLoadIndexArgs`，因为参数太多，他们以结构体的形式进行封装。

```
1 // @file: b_tree.h
2 typedef struct
3 {
4     const Result *hashTable; // hash table that store entries
5     int entryNum;           // number of entries that the function needs
6     // to load
7     int *blockIndex;        // the index of block that allocated for
8     // these entries
9     int blockIndexLen;      // the length of array blockIndex
10    int blockIndexStart;     // the start index of array blockIndex
11    int blockIndexEnd;       // the end index of array blockIndex
12    int maxBlock;            // max number of block the function can
13    // allocate
14    int minBlock;            // min block num that write out
15    BTree *bTree;            // the btree that these nodes should belongs
16    // to
17    pthread_mutex_t *fileLock; // prevent multi-thread access file
18 } BatchLoadLeafArgs;
```

```
1 // @file: b_tree.h
2 typedef struct
3 {
4     int *sonBlockTable;      // hash table that store entries
5     float *sonKeyTable;      // table that store sons' key index
6     int son_num;             // the number of entries that the function
7     // needs to load
8     int currentLevel;        // layers level
9     int *blockIndex;         // the index of block that allocated for
10    // these entries
11    int blockIndexLen;        // the length of array blockIndex
12    int blockIndexStart;      // the start index of array blockIndex
13    int blockIndexEnd;        // the end index of array blockIndex
14    int maxBlock;             // max number of block the function can
15    // allocate
16    int minBlock;             // min block num that write out
17    BTree *bTree;             // the btree that these nodes should belongs
18    // to
19    pthread_mutex_t *fileLock; // prevent multi-thread access file
20 } BatchLoadIndexArgs;
```

`parallelBulkLoad` 代码及注释:

```
1 // @file: b_tree.cc
2 int BTree::parallelBulkLoad(
3     int n, // number of entries
4     const Result *table, // hash table
5     int maxThreadNum, // max thread num
```

```

6     int maxBufferBlock    // max number of blocks in buffer
7 )
8 {
9     pthread_mutex_t fileLock; // lock for write out
10    pthread_mutex_init(&fileLock, NULL);
11
12    int threadNum = maxThreadNum;
13    int id = -1;
14    int block = -1;
15    float key = MINREAL;
16
17    // get Index node and Leaf node's capacity
18    int BlockLength = this->file_->get_blocklength();
19    int BIndexNodeCap = (BlockLength - (SIZECHAR + SIZEINT * 3)) /
(SIZEFLOAT + SIZEINT);
20    int BLeafNodeCap = (BlockLength - (SIZECHAR + SIZEINT * 3) -
(int)ceil((float)BlockLength / LEAF_NODE_SIZE) * SIZEFLOAT - SIZEINT) /
SIZEINT;
21
22    // calculate the blocks number that the leaves need
23    int LeafBlockNum = (int)ceil((double)n / BLeafNodeCap);
24
25    // pre-allocate blocks for the Leaf layer
26    int *LeafBlockIndex = alloc_blocks(LeafBlockNum);
27    int c = 0;
28
29    // -----
-----
30    // build leaf node from <_hashtable> (level = 0)
31    // -----
-----
32    bool firstNode = true; // determine relationship of sibling
33    int startBlock = 0;    // position of first node
34    int endBlock = 0;      // position of last node
35
36    // calculate batch load variable
37    int batchBlockSize = LeafBlockNum / threadNum;
38    int lastBatchBlockSize = LeafBlockNum - (threadNum - 1) *
batchBlockSize;
39    int batchEntrySize = batchBlockSize * BLeafNodeCap;
40    int lastBatchEntrySize = n - batchBlockSize * BLeafNodeCap * (threadNum
- 1);
41    int MAX_BLOCK = maxBufferBlock;
42    int MIN_BLOCK = maxBufferBlock > 30 ? 10 : maxBufferBlock / 2;
43
44    // define thread and arguments for each thread
45    pthread_t tid[threadNum + 5];
46    BatchLoadLeafArgs BLLA[threadNum + 5];
47    BatchLoadIndexArgs BLIA[threadNum + 5];
48
49    // assign thread arguments for leaf node
50    for (int i = 0; i < threadNum; i++)
51    {
52        BLLA[i].hashTable = table + i * batchEntrySize;
53        BLLA[i].entryNum = batchEntrySize;
54        BLLA[i].blockIndex = LeafBlockIndex;
55        BLLA[i].blockIndexLen = LeafBlockNum;
56        BLLA[i].blockIndexStart = 0 + i * batchBlockSize;

```



```

57         BLLA[i].blockIndexEnd = BLLA[i].blockIndexStart + batchBlockSize -
1;
58         BLLA[i].maxBlock = MAX_BLOCK;
59         BLLA[i].minBlock = MIN_BLOCK;
60         BLLA[i].bTree = this;
61         BLLA[i].fileLock = &fileLock;
62     }
63     BLLA[threadNum - 1].entryNum = lastBatchEntrySize;
64     BLLA[threadNum - 1].blockIndexEnd = BLLA[threadNum - 1].blockIndexStart
+ lastBatchBlockSize - 1;
65
66     float *leafBlockKey = (float *)malloc(SIZEFLOAT * LeafBlockNum);
67     int keyCount = 0;
68     float *batchKey = NULL;
69
70     // create threads for leaf node
71     for (int i = 0; i < threadNum; ++i)
72     {
73         if (BLLA[i].entryNum > 0)
74         {
75             pthread_create(&tid[i], NULL, batchLoadLeaf, &(BLLA[i]));
76         }
77     }
78     // waiting for all threads finishing
79     for (int i = 0; i < threadNum; i++)
80     {
81         if (BLLA[i].entryNum > 0)
82         {
83             pthread_join(tid[i], (void **)&batchKey);
84             for (int j = 0; j <= (BLLA[i].blockIndexEnd -
BLLA[i].blockIndexStart); j++)
85             {
86                 leafBlockKey[keyCount++] = batchKey[j];
87             }
88             free(batchKey);
89         }
90     }
91     startBlock = LeafBlockIndex[0];
92     endBlock = LeafBlockIndex[LeafBlockNum - 1];
93
94     // -----
95     // build index node from bottom to top
96     // stop condition: lastEndBlock == lastStartBlock (only one node, as
root)
97     // -----
98     int current_level = 1; // current level (leaf level is 0)
99     int lastStartBlock = startBlock; // build b-tree level by level
100    int lastEndBlock = endBlock; // build b-tree level by level
101    float *lastLayerKey = leafBlockKey;
102    float *currentLayerKey = NULL;
103    int *lastLayerBlockIndex = LeafBlockIndex;
104    int lastLayerBlockNum = LeafBlockNum;
105    int *currentLayerBlockIndex = NULL;
106    int currentLayerBlockNum;
107
108    while (lastEndBlock > lastStartBlock)

```

```

109     {
110         currentLayerBlockNum = (int)ceil((double)lastLayerBlockNum /
BIndexNodeCap);
111         currentLayerBlockIndex = alloc_blocks(currentLayerBlockNum);
112         batchBlockSize = currentLayerBlockNum / threadNum;
113         lastBatchBlockSize = currentLayerBlockNum - (threadNum - 1) *
batchBlockSize;
114         batchEntrySize = batchBlockSize * BIndexNodeCap;
115         lastBatchEntrySize = lastLayerBlockNum - batchBlockSize *
BIndexNodeCap * (threadNum - 1);
116
117         // assign thread arguments
118         for (int i = 0; i < threadNum; i++)
119         {
120             BLIA[i].sonBlockTable = lastLayerBlockIndex + i *
batchEntrySize;
121             BLIA[i].sonKeyTable = lastLayerKey + i * batchEntrySize;
122             BLIA[i].son_num = batchEntrySize;
123             BLIA[i].currentLevel = current_level;
124             BLIA[i].blockIndex = currentLayerBlockIndex;
125             BLIA[i].blockIndexLen = currentLayerBlockNum;
126             BLIA[i].blockIndexStart = 0 + i * batchBlockSize;
127             BLIA[i].blockIndexEnd = BLIA[i].blockIndexStart +
batchBlockSize - 1;
128             BLIA[i].maxBlock = MAX_BLOCK;
129             BLIA[i].minBlock = MIN_BLOCK;
130             BLIA[i].bTree = this;
131             BLIA[i].fileLock = &fileLock;
132         }
133
134         BLIA[threadNum - 1].son_num = lastBatchEntrySize;
135         BLIA[threadNum - 1].blockIndexEnd = BLIA[threadNum -
1].blockIndexStart + lastBatchBlockSize - 1;
136
137         // create threads
138         for (int i = 0; i <= threadNum - 1; i++)
139         {
140             if (BLIA[i].son_num > 0)
141             {
142                 pthread_create(&tid[i], NULL, batchLoadIndex, &(BLIA[i]));
143             }
144         }
145         currentLayerKey = (float *)malloc(sizeof(float) *
(currentLayerBlockNum));
146         keyCount = 0;
147
148         // waiting for all threads finsihing
149         for (int i = 0; i <= threadNum - 1; i++)
150         {
151             if (BLIA[i].son_num > 0)
152             {
153                 pthread_join(tid[i], (void **)&batchKey);
154                 for (int j = 0; j <= (BLIA[i].blockIndexEnd -
BLIA[i].blockIndexStart); j++)
155                 {
156                     currentLayerKey[keyCount++] = batchKey[j];
157                 }
158                 free(batchKey);

```

```

159         }
160     }
161
162     free(lastLayerBlockIndex);
163     free(lastLayerKey);
164     lastLayerBlockIndex = currentLayerBlockIndex;
165     lastLayerKey = currentLayerKey;
166     lastLayerBlockNum = currentLayerBlockNum;
167     lastStartBlock = currentLayerBlockIndex[0];
168
169     // update info
170     lastEndBlock = currentLayerBlockIndex[currentLayerBlockNum - 1];
171     // build b-tree of higher level
172     ++current_level;
173     currentLayerBlockIndex = NULL;
174     currentLayerKey = NULL;
175 }
176 assert(lastStartBlock == lastEndBlock);
177 root_ = lastStartBlock; // update the <root>
178 return 0;
179 }

```

5.2 batchLoadLeaf 和 batchLoadIndex

`batchLoadLeaf` 和 `batchLoadIndex` 分别是创建叶子节点的线程和创建索引节点的线程的线程函数。

线程函数的任务就是构建指定数目的节点，设置他们的左右节点，添加索引项或数据项，在获得了写文件权限后写入内存。

线程内维护了一个节点缓冲区用于保存已完成但未写入磁盘的节点，当缓冲区内节点数目大于参数 `minBLOCK` 时，尝试获取锁写回磁盘，当节点数目未为 `maxBLOCK` 时，堵塞知道获取锁然后写回磁盘。

```

1 // @file: b_tree.cc
2 /*
3     start routine of thread, this function wil put n entries into severall
4     blocks, every leaf node will be as full as possible
5 */
6 void *BTree::batchLoadLeaf(
7     void *args)
8 {
9     // load argument structure
10    BatchLoadLeafArgs *funcArgs = (BatchLoadLeafArgs *)args;
11    const Result *hashTable = funcArgs->hashTable;
12    int entryNum = funcArgs->entryNum;
13    int *blockIndex = funcArgs->blockIndex;
14    int blockIndexLen = funcArgs->blockIndexLen;
15    int blockIndexStart = funcArgs->blockIndexStart;
16    int blockIndexEnd = funcArgs->blockIndexEnd;
17    int maxBlock = funcArgs->maxBlock;
18    int minBlock = funcArgs->minBlock;
19    BTree *bTree = funcArgs->bTree;
20    pthread_mutex_t *fileLock = funcArgs->fileLock;
21
22    float *batchKey = (float *)malloc(sizeof(float) * (blockIndexEnd -
    blockIndexStart + 1)); // node key on the fly

```

```

23     int keyCount = 0;
24
25     // queue for unwritten out blocks
26     BLeafNode *unwrittenBackNodeQueue[maxBlock];
27     int queueTail = -1;
28     int queuehead = 0;
29
30     int id;
31     float key;
32     BLeafNode *currentLeafNode = NULL;
33     int currentBlockIndex = blockIndexStart;
34     for (int i = 0; i <= entryNum - 1; i++)
35     {
36         assert(currentBlockIndex <= blockIndexEnd);
37         // get data entry
38         id = hashTable[i].id_;
39         key = hashTable[i].key_;
40
41         if (currentLeafNode == NULL)
42         {
43             currentLeafNode = new BLeafNode();
44             currentLeafNode->init_noalloc(0, bTree,
blockIndex[currentBlockIndex]);
45             // set siblings for current node
46             if (currentBlockIndex - 1 >= 0)
47             { // has previous leaf node
48                 currentLeafNode->
set_left_sibling(blockIndex[currentBlockIndex - 1]);
49             }
50             if (currentBlockIndex + 1 <= blockIndexLen - 1)
51             { // has next leaf node
52                 currentLeafNode->
set_right_sibling(blockIndex[currentBlockIndex + 1]);
53             }
54         }
55
56         // fill node with data entry
57         currentLeafNode->add_new_child(id, key);
58
59         if (currentLeafNode->isFull())
60         {
61             unwrittenBackNodeQueue[++queueTail] = currentLeafNode; // store
the leaf in memory
62             if (queueTail >= maxBlock - 1)
63             { // exceeding the max number
of blocks in memory
64                 queuehead = 0; // points to the bottom of
the stack
65                 pthread_mutex_lock(fileLock); // the thread will be
suspended if the lock is occupied
66
67                 // write out blocks that in queue if buffer in full
68                 bTree->write_leaf_blocks(unwrittenBackNodeQueue,
unwrittenBackNodeQueue[queuehead]->get_block(), queueTail + 1);
69                 while (queuehead <= queueTail)
70                 {
71                     batchKey[keyCount++] =
unwrittenBackNodeQueue[queuehead]->get_key_of_node();

```

```

72         delete unwrittenBackNodeQueue[queuehead];
73         queuehead++;
74     }
75     pthread_mutex_unlock(fileLock);
76     queueTail = -1;
77     queuehead = 0;
78 }
79 else
80 { // not exceeding the max number of blocks in memory
81     if (queueTail + 1 >= minBlock)
82     {
83         int tryLockRes = pthread_mutex_trylock(fileLock);
84         if (tryLockRes == 0)
85         { // the lock is free
86             queuehead = 0;
87
88             // write out blocks that in queue
89             bTree->write_leaf_blocks(unwrittenBackNodeQueue,
unwrittenBackNodeQueue[queuehead]->get_block(), queueTail + 1);
90             while (queuehead <= queueTail)
91             {
92                 batchKey[keyCount++] =
unwrittenBackNodeQueue[queuehead]->get_key_of_node();
93                 delete unwrittenBackNodeQueue[queuehead];
94                 queuehead++;
95             }
96
97             pthread_mutex_unlock(fileLock);
98             queueTail = -1;
99             queuehead = 0;
100         }
101     }
102 }
103 currentLeafNode = NULL;
104 currentBlockIndex++;
105 }
106 }
107 if (currentLeafNode != NULL)
108 {
109     unwrittenBackNodeQueue[++queueTail] = currentLeafNode; // store the
leaf in memory
110 }
111 pthread_mutex_lock(fileLock); // the thread will be suspended if the
lock is occupied
112 queuehead = 0;
113
114 // write out blocks that in queue if queue in not empty
115 bTree->write_leaf_blocks(unwrittenBackNodeQueue,
unwrittenBackNodeQueue[queuehead]->get_block(), queueTail + 1);
116 while (queuehead <= queueTail)
117 {
118     batchKey[keyCount++] = unwrittenBackNodeQueue[queuehead]-
>get_key_of_node();
119     delete unwrittenBackNodeQueue[queuehead];
120     queuehead++;
121 }
122 pthread_mutex_unlock(fileLock);
123 queueTail = -1;

```

```

124     queuehead = 0;
125     assert(keyCount == blockIndexEnd - blockIndexStart + 1);
126     return batchKey;
127 }

```

```

1  // @file: b_tree.cc
2  void *BTree::batchLoadIndex(void *args)
3  {
4      // load argument structure
5      BatchLoadIndexArgs *funcArgs = (BatchLoadIndexArgs *)args;
6      int *sonBlockTable = funcArgs->sonBlockTable;
7      float *sonKeyTable = funcArgs->sonKeyTable;
8      int son_num = funcArgs->son_num;
9      int *blockIndex = funcArgs->blockIndex;
10     int currentLevel = funcArgs->currentLevel;
11     int blockIndexLen = funcArgs->blockIndexLen;
12     int blockIndexStart = funcArgs->blockIndexStart;
13     int blockIndexEnd = funcArgs->blockIndexEnd;
14     int maxBlock = funcArgs->maxBlock;
15     int minBlock = funcArgs->minBlock;
16     BTree *bTree = funcArgs->bTree;
17     pthread_mutex_t *fileLock = funcArgs->fileLock;
18
19     int block;
20     float key;
21     BIndexNode *currentIndexNode = NULL;
22     int currentBlockIndex = blockIndexStart;
23
24     // queue for unwritten out blocks
25     BIndexNode *unwrittenBackNodeQueue[maxBlock];
26     int queueTail = -1;
27     int lockRet;
28     int queuehead = 0;
29
30     float *batchKey = (float *)malloc(sizeof(float) * (blockIndexEnd -
blockIndexStart + 1)); // node key on the fly
31     int keyCount = 0;
32
33     for (int i = 0; i <= son_num - 1; i++)
34     {
35         block = sonBlockTable[i]; // get <block>
36         key = sonKeyTable[i];
37
38         if (currentIndexNode == NULL)
39         {
40             currentIndexNode = new BIndexNode();
41             currentIndexNode->init_noalloc(currentLevel, bTree,
blockIndex[currentBlockIndex]);
42
43             // set siblings for current node
44             if (currentBlockIndex - 1 >= 0)
45                 currentIndexNode->
set_left_sibling(blockIndex[currentBlockIndex - 1]);
46             if (currentBlockIndex + 1 <= blockIndexLen - 1)
47                 currentIndexNode->
set_right_sibling(blockIndex[currentBlockIndex + 1]);
48         }

```

```

49
50     currentIndexNode->add_new_child(key, block); // add new entry
51
52     if (currentIndexNode->isFull())
53     {
54         unwrittenBackNodeQueue[++queueTail] = currentIndexNode;
55         if (queueTail >= maxBlock - 1)
56         { // the buffer is full. Have to write these nodes out
57             pthread_mutex_lock(fileLock);
58             queuehead = 0;
59
60             // write out blocks that in queue if buffer is full
61             bTree->write_index_blocks(unwrittenBackNodeQueue,
unwrittenBackNodeQueue[queuehead]->get_block(), \
62                                     queueTail + 1);
63
64             while (queuehead <= queueTail)
65             {
66                 batchKey[keyCount++] =
unwrittenBackNodeQueue[queuehead]->get_key_of_node();
67                 delete unwrittenBackNodeQueue[queuehead];
68                 queuehead++;
69             }
70             pthread_mutex_unlock(fileLock);
71             queueTail = -1;
72             queuehead = 0;
73         }
74     else
75     {
76         if (queueTail + 1 >= minBlock)
77         {
78             lockRet = pthread_mutex_trylock(fileLock);
79             if (lockRet == 0)
80             { // the lock is free
81                 queuehead = 0;
82
83                 // write out blocks that in queue
84                 bTree->write_index_blocks(unwrittenBackNodeQueue,
unwrittenBackNodeQueue[queuehead]->get_block(), \
85                                         queueTail + 1);
86                 while (queuehead <= queueTail)
87                 {
88                     batchKey[keyCount++] =
unwrittenBackNodeQueue[queuehead]->get_key_of_node();
89                     delete unwrittenBackNodeQueue[queuehead];
90                     queuehead++;
91                 }
92                 pthread_mutex_unlock(fileLock);
93                 queueTail = -1;
94                 queuehead = 0;
95             }
96         }
97     }
98     currentIndexNode = NULL;
99     currentBlockIndex++;
100 }
101 }
102 if (currentIndexNode != NULL)

```

```

103     {
104         unwrittenBackNodeQueue[++queueTail] = currentIndexNode;
105     }
106
107     // write out blocks that in queue if queue in not empty
108     pthread_mutex_lock(fileLock);
109     queuehead = 0;
110     bTree->write_index_blocks(unwrittenBackNodeQueue,
unwrittenBackNodeQueue[queuehead]->get_block(), queueTail + 1);
111     while (queuehead <= queueTail)
112     {
113         batchKey[keyCount++] = unwrittenBackNodeQueue[queuehead]-
>get_key_of_node();
114         delete unwrittenBackNodeQueue[queuehead];
115         queuehead++;
116     }
117     pthread_mutex_unlock(fileLock);
118     queueTail = -1;
119     queuehead = 0;
120
121     assert(keyCount == blockIndexEnd - blockIndexStart + 1);
122     return batchKey;
123 }

```

5.3 append_blocks、init_noalloc 和 alloc_blocks

优化代码一：连续分配块过程对 blockfile、b_node 和 b_tree 的修改，优化思路已在4.1节中描述。

BlockFile中连续块号申请的底层实现：

```

1  // @file: blockfile.cc
2  // append a series of blocks(cont), avoid calling fwrite too often
3  int* BlockFile::append_blocks(    // append new block at the end of file
4      Block block,                  // the new blocks
5      int n)                        // the number of blocks
6  {
7      fseek(fp_, 0, SEEK_END);      // <fp_> point to the end of file
8      fwrite(block, n, block_length_, fp_);
9      int start_block = num_blocks_;
10     num_blocks_ += n;              // add 1 to <num_blocks_>
11
12     fseek(fp_, SIZEINT, SEEK_SET); // <fp_> point to pos of header
13     fwrite_number(num_blocks_);    // update <num_blocks_>
14
15     // -----
16     // <fp_> point to the pos of new added block.
17     // the equation <act_block_> = <num_blocks_> indicates the file pointer
18     // point to new added block.
19     // return index of new added block
20     // -----
21
22     fseek(fp_, -block_length_, SEEK_END);
23     act_block_ = num_blocks_;
24     int* blockIndex = (int*)malloc(sizeof(int) * n);
25     for(int i = 0; i <= n - 1; i++) {
26         blockIndex[i] = start_block + i;
27     }
28 }

```



```

26     }
27     return blockIndex;
28 }

```

`init_noalloc` 函数的实现包括: BNode的实现、BIndexNode的重载和BLeafNode的重载。

```

1 // @file: b_node.cc
2 void BNode::init_noalloc( // init a new node in memory without allocating a
   block
3     int level,           // level in b-tree
4     BTree *btree,        // b-tree of this node
5     int block            // the index of the block that pre-allocated for
   this node
6 )
7 {
8     btree_ = btree;
9     level_ = (char)level;
10    dirty_ = true;
11    left_sibling_ = -1;
12    right_sibling_ = -1;
13    key_ = NULL;
14    num_entries_ = 0;
15    block_ = block; // different with BNode::init
16    capacity_ = -1;
17 }
18
19 // @file: b_node.cc
20 void BIndexNode::init_noalloc( // init a new node in memory without
   allocating a block
21     int level,           // level in b-tree
22     BTree *btree,        // b-tree of this node
23     int block            // the index of the block that pre-allocated
   for this node
24 )
25 {
26     btree_ = btree;
27     level_ = (char)level;
28     block_ = block;
29     num_entries_ = 0;
30     left_sibling_ = -1;
31     right_sibling_ = -1;
32     dirty_ = true;
33
34     int b_length = btree->file->get_blocklength();
35     capacity_ = (b_length - get_header_size()) / get_entry_size();
36     if (capacity_ < 50)
37     { // ensure at least 50 entries
38         printf("capacity = %d, which is too small.\n", capacity_);
39         exit(1);
40     }
41
42     key_ = new float[capacity_];
43     son_ = new int[capacity_];
44     // alloc memory
45     memset(key_, MINREAL, capacity_ * SIZEFLOAT);
46     memset(son_, -1, capacity_ * SIZEINT);
47     // without allocating block in the file

```

```

48 }
49
50 // @file: b_node.cc
51 void BLeafNode::init_noalloc( //init a new node in memory without
    allocating a block
52     int level,                // level in b-tree
53     BTree *btree,            // b-tree of this node
54     int block                 // the index of the block that pre-allocated
    for this node
55 )
56 {
57     btree_ = btree;
58     level_ = (char)level;
59     block_ = block;
60     num_entries_ = 0;
61     num_keys_ = 0;
62     left_sibling_ = -1;
63     right_sibling_ = -1;
64     dirty_ = true;
65
66     int b_length = btree_>file_>get_blocklength();
67     int key_size = get_key_size(b_length);
68
69     key_ = new float[capacity_keys_];
70     memset(key_, MINREAL, capacity_keys_ * SIZEFLOAT);
71
72     int header_size = get_header_size();
73     int entry_size = get_entry_size();
74
75     capacity_ = (b_length - header_size - key_size) / entry_size;
76     if (capacity_ < 100)
77     { // at least 100 entries
78         printf("capacity = %d, which is too small.\n", capacity_);
79         exit(1);
80     }
81     id_ = new int[capacity_];
82     memset(id_, -1, capacity_ * SIZEINT);
83     // without allocating block in the file
84 }

```

BTree中的连续块分配方法:

```

1 // @file: b_tree.cc
2 int *BTree::alloc_blocks( // allocate n empty blocks in the block file
3     int n)
4 {
5     char *randomBlk = (char *)malloc(sizeof(char) * (this->file_-
    >get_blocklength()) * n);
6     // an useless block, just to fill the file
7     int *blockIndexs = this->file_>append_blocks(randomBlk, n);
8     return blockIndexs;
9 } // returns an array of indexs of blocks that were allocated

```

5.4 write_blocks、write_leaf_blocks 和 write_index_blocks

优化代码二：连续分配块过程对 blockfile 和 b_tree 的修改，优化思路已在4.2节中描述。

BlockFile中连续块写入的底层实现：

```
1 // @file: blockfile.cc
2 bool BlockFile::write_blocks( // write a block <b> in the <pos>
3     Block blocks, // a series of blocks
4     int firstBlock, // pos of the first block
5     int n) // block num
6 {
7     firstBlock++;
8     fseek(fp_, block_length_ * firstBlock, SEEK_SET);
9     fwrite(blocks, block_length_, n, fp_);
10
11     if (firstBlock + n > num_blocks_) { // update <act_block_>
12         fseek(fp_, 0, SEEK_SET);
13         act_block_ = 0;
14     }
15     else {
16         act_block_ = firstBlock + n;
17     }
18     return true;
19 }
```

BTree 中 write_leaf_blocks 和 write_index_blocks 的实现：

```
1 // @file: b_tree.cc
2 void BTree::write_leaf_blocks( // write n blocks to the block file
3     BLeafNode* nodes[], // array of leaf node's info
4     int firstBlock, // first block num
5     int n) // total block num
6 {
7
8     int block_length = file_>get_blocklength();
9     char* blocksContent = new char[block_length * n];
10    int ptr = 0;
11
12    // merge all the blocks content into a buffer
13    for(int i = 0; i < n; i++){
14        char *buf = new char[block_length];
15        nodes[i]>write_to_buffer(buf);
16        nodes[i]>set_dirty(false);
17        for(int j = 0; j < block_length; j++){
18            blocksContent[ptr++] = buf[j];
19        }
20        delete[] buf;
21        buf = NULL;
22    }
23    this->file_>write_blocks(blocksContent, firstBlock, n);
24    delete[] blocksContent;
25 }
26
27 // @file: b_tree.cc
```

```

28 void BTree::write_index_blocks( // write n blocks to the block file
29     BIndexNode* nodes[],      // array of index node's info
30     int firstBlock,           // first block num
31     int n)                    // total block num
32 {
33     int block_length = file_>get_blocklength();
34     char* blocksContent = new char[block_length * n];
35     int ptr = 0;
36
37     // merge all the blocks content into a buffer
38     for(int i = 0; i < n; i++){
39         char *buf = new char[block_length];
40         nodes[i]>write_to_buffer(buf);
41         nodes[i]>set_dirty(false);
42         for(int j = 0; j < block_length; j++){
43             blocksContent[ptr++] = buf[j];
44         }
45         delete[] buf;
46         buf = NULL;
47     }
48     this->file_>write_blocks(blocksContent, firstBlock, n);
49     delete[] blocksContent;
50 }

```

5.5 write_B_tree

`write_B_tree`: 用于判断传并行产生的B+树是否一致，以此来对算法进行正确性测试。

```

1  // @file: main.cc
2  // print B_Tree and store it in file
3  void print_B_Tree(BTree *trees_, char *filename)
4  {
5      FILE *fp;
6      fp = fopen(filename, "w");
7      if (fp == NULL)
8      {
9          printf("File can not open!");
10         exit(0);
11     }
12
13     int start_block = trees_>root_;
14     int end_block = trees_>root_;
15     int newly_startblock;
16     int newly_endblock;
17
18     BIndexNode *index_child = NULL;
19
20     // read root node
21     char indexnode_level;
22     int indexnode_num_entries;
23     BIndexNode *indexnode_left_sibling;
24     BIndexNode *indexnode_right_sibling;
25
26     fprintf(fp, "root: block %d\n", start_block);
27     index_child = new BIndexNode();
28     index_child->init_restore(trees_, start_block);
29     indexnode_level = index_child->get_level();

```

```

30     indexnode_num_entries = index_child->get_num_entries();
31     indexnode_left_sibling = index_child->get_left_sibling();
32     indexnode_right_sibling = index_child->get_right_sibling();
33     fprintf(fp, "\tlevel: %d \tnum_entries: %d\n", indexnode_level,
indexnode_num_entries);
34     for (int j = 0; j < indexnode_num_entries; ++j)
35     {
36         fprintf(fp, "\t\tkey: %f\tson: %d\n", index_child->get_key(j),
index_child->get_son(j));
37     }
38
39     start_block = index_child->get_son(0);
40     end_block = index_child->get_son(indexnode_num_entries - 1);
41     delete index_child;
42     index_child = NULL;
43
44     // index node
45     // from root to the leaf layer to layer
46     while (start_block > 1)
47     {
48         for (int k = start_block; k <= end_block; k++)
49         {
50             fprintf(fp, "index: block %d\n", k);
51             index_child = new BIndexNode();
52             index_child->init_restore(trees_, k);
53             indexnode_level = index_child->get_level();
54             indexnode_num_entries = index_child->get_num_entries();
55             indexnode_left_sibling = index_child->get_left_sibling();
56             indexnode_right_sibling = index_child->get_right_sibling();
57             fprintf(fp, "\tlevel: %d \tnum_entries: %d\n", indexnode_level,
indexnode_num_entries);
58             for (int j = 0; j < indexnode_num_entries; ++j)
59             {
60                 fprintf(fp, "\t\tkey: %f\tson: %d\n", index_child-
>get_key(j), index_child->get_son(j));
61             }
62             if (k == start_block)
63             {
64                 newly_startblock = index_child->get_son(0);
65             }
66             if (k == end_block)
67             {
68                 newly_endblock = index_child->get_son(indexnode_num_entries
- 1);
69             }
70             delete index_child;
71             index_child = NULL;
72         } // end for loop
73         start_block = newly_startblock;
74         end_block = newly_endblock;
75     } // end while
76
77     // leaf node variable
78     BLeafNode *leaf_child = NULL;
79
80     // read root node
81     char leafnode_level;
82     int leafnode_num_entries;

```

```

83     int leafnode_num_keys;
84     BLeafNode *leafnode_left_sibling;
85     BLeafNode *leafnode_right_sibling;
86
87     // print leaf node
88     for (int k = start_block; k <= end_block; k++)
89     {
90         fprintf(fp, "leaf: block %d\n", k);
91         leaf_child = new BLeafNode();
92         leaf_child->init_restore(trees_, k);
93         leafnode_level = leaf_child->get_level();
94         leafnode_num_entries = leaf_child->get_num_entries();
95         leafnode_left_sibling = leaf_child->get_left_sibling();
96         leafnode_right_sibling = leaf_child->get_right_sibling();
97         leafnode_num_keys = leaf_child->get_num_keys();
98         fprintf(fp, "\tlevel: %d \tnum_entries: %d\tnum_keys: %d\n",
leafnode_level, leafnode_num_entries, leafnode_num_keys);
99         for (int j = 0; j < leafnode_num_keys; ++j)
100         {
101             int count_entries = 0;
102             fprintf(fp, "\t\tkey: %f\n", leaf_child->get_key(j));
103             for (int w = count_entries; w < std::min(count_entries + 16,
leafnode_num_entries); w++)
104             {
105                 fprintf(fp, "\t\t\tid: %d\n", leaf_child->get_entry_id(w));
106             }
107             count_entries += 16;
108         }
109         delete leaf_child;
110         leaf_child = NULL;
111     }
112
113     fclose(fp);
114 }

```

VI. 实验结果分析

6.0 并行和性能调优的代码说明

由于我们提交的代码是将并行和性能调优一起使用的代码，因此在这里有必要说明改到只有并行处理的方式。同时说明一下如何测试代码。

6.0.1 改为非连续写磁盘的方式

- 将 batchLoadLeaf 函数的 //DO 处标记的函数 write_leaf_blocks 注释掉。
- 将 batchLoadIndex 函数的 //DO 处标记的函数 write_index_blocks 注释掉。

6.0.2 改为非连续分配块的方式

1. 将 int *BTree::alloc_blocks(int n) 函数的前两行注释掉
2. 并添加如下几行：

```

1  int *blockIndexs = (int*)malloc(sizeof(int) * n);
2      for(int i = 0; i < n; i++) {
3          char *randomBlk = (char *)malloc(sizeof(char) * (this->file_-
>get_blocklength()));
4          blockIndexs[i] = this->file_->append_block(randomBlk);
5      }

```

6.0.3 如何测试代码

1. 将 main.cc 文件的 evaluate() 函数的注释去掉
2. 使用命令 `sh run.sh` 运行脚本文件

6.1 并行实验结果代码及分析

注：为保证实验的准确性和有效性，排除CPU此时的运行情况和内存情况以及其他进程的影响，我们测试的实验结果都是采用10次实验结果取平均得到的结果，并且使用Python绘图来让结果可视化。

6.1.1 影响因素

并行处理函数的参数就包含了我们的所要分析的影响因素。

```

1  int BTree::parallelBulkLoad(
2      int n,                // number of entries
3      const Result *table, // hash table
4      int maxThreadNum,    // max thread num
5      int maxBufferBlock   // max number of blocks in buffer
6  )

```

这里面除了 `table` 变量是数据集外，其他3个变量即数据项的数目，最大的线程数，Buffer的大小，都是我们需要考虑的因素。

这里我们并没有改变Block的大小，默认仍是512个字节。

6.1.2 实验方法

我们采用**控制变量法**的方式，先确定好其他两个变量的默认值，再去改变其他变量。

3个变量的默认值如下：

- 线程数目：2
- 数据项的数目：1000000
- Buffer的大小：500个Block

6.1.3 数据项的数目 `int n`

测试代码

```

1  int entryNum[] = {100,1000,10000,100000,1000000,10000000};
2  outFile.open("./result/result_entryNum.csv", ios::out);
3  outFile<<"entryNum  servertime  partime"<<endl;
4  printf("结果存放于./result/result_entryNum.csv\n");
5  for(int i = 0;i < 6;++i){
6      float entry_run1 = 0;
7      float entry_run2 = 0;
8      make_data(entryNum[i]);
9
10     strncpy(data_file, "./data/dataset.csv", sizeof(data_file));

```

```

11     strncpy(tree_file_ser, "./result/B_tree_ser",
sizeof(tree_file_ser));
12     strncpy(tree_file_par, "./result/B_tree_par",
sizeof(tree_file_par));
13
14     Result *table = new Result[entryNum[i]];
15
16     ifstream fp(data_file);
17     string line;
18     int k = 0;
19     while (getline(fp,line) && k <= entryNum[i] - 1){
20         string number;
21         istringstream readstr(line);
22
23         getline(readstr,number,',');
24         table[k].key_ = atof(number.c_str());
25
26         getline(readstr,number,',');
27         table[k].id_ = atoi(number.c_str());
28         k++;
29     }
30
31     fp.close();
32     for(int w = 0; w < AVERAGE_NUM; w++) {
33         timeval start_t;
34         timeval end_t;
35
36         BTree* trees_ = new BTree();
37         trees_>init(B_, tree_file_ser);
38         gettimeofday(&start_t,NULL);
39
40         if (trees_>bulkload(entryNum[i], table)) return ;
41
42         gettimeofday(&end_t, NULL);
43         float run_t1 = end_t.tv_sec - start_t.tv_sec +
44             (end_t.tv_usec - start_t.tv_usec) / 1000000.0f;
45         //printf("串行运行时间: %f s\n", run_t1);
46         delete trees_;
47         //-----
48         -----
49         trees_ = new BTree();
50
51         trees_>init(B_, tree_file_par);
52         gettimeofday(&start_t,NULL);
53
54         if (trees_>parallelBulkLoad(entryNum[i], table,
DEFAULT_THREAD_NUM, DEFAULT_BUFFER_SIZE)) return ;
55         gettimeofday(&end_t, NULL);
56
57         float run_t2 = end_t.tv_sec - start_t.tv_sec +
58             (end_t.tv_usec - start_t.tv_usec) /
1000000.0f;
59         //printf("并行运行时间: %f s\n", run_t1);
60         entry_run1 += run_t1;
61         entry_run2 += run_t2;
62     }
63     if(table != NULL){

```



```

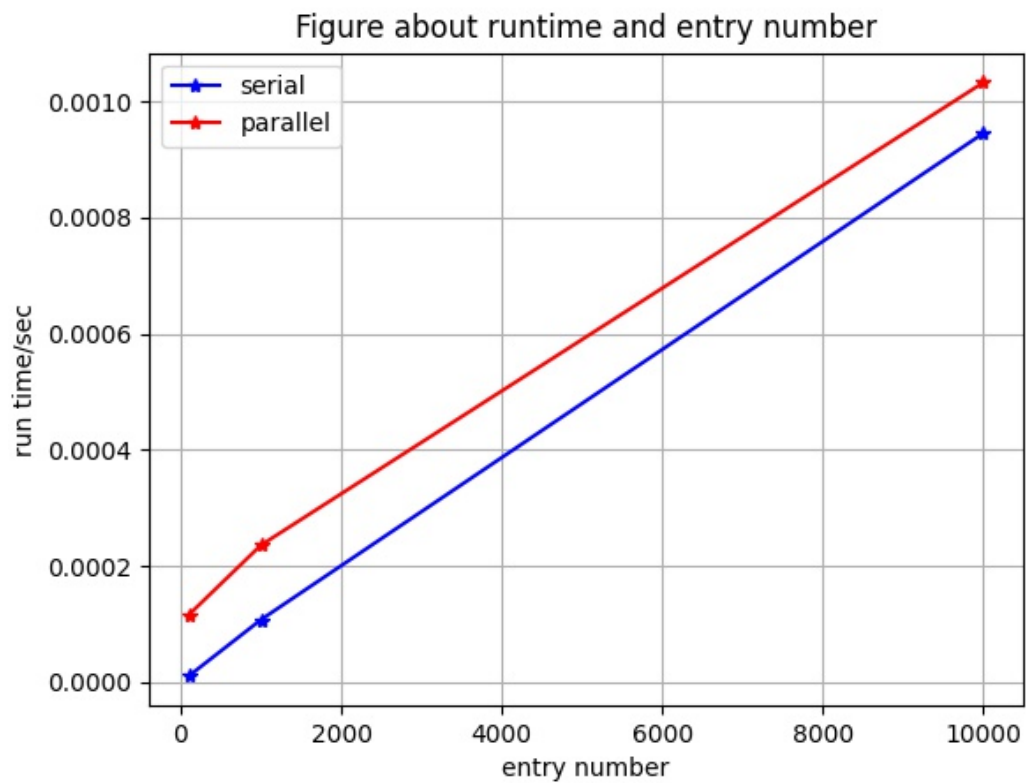
64         delete[] table;
65         table = NULL;
66     }
67     outFile<<entryNum[i]<<" "<<entry_run1/AVERAGE_NUM<<" "
<<entry_run2/AVERAGE_NUM<<endl;
68 }
69 outFile.close();

```

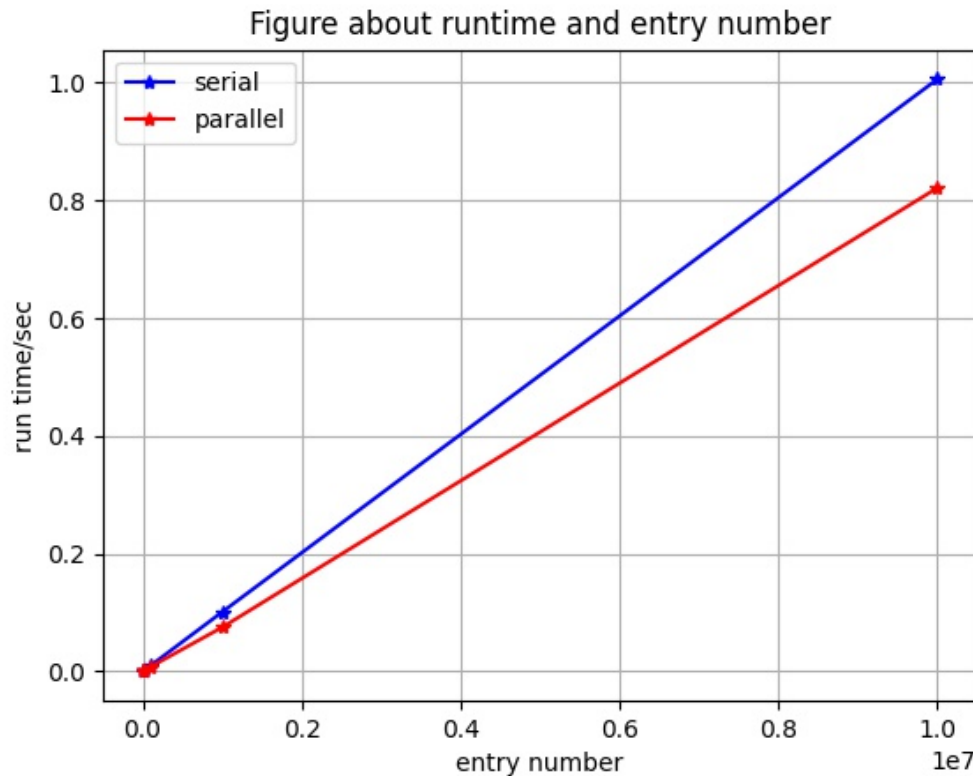
实验结果

实验结果分为两部分：

- 第一部分为数据量较小的部分，数据量小于 10^4 ：



- 第二部分为数据量较大的一部分，数据量直到 10^7 ：



结果分析

- 当数据量较小的时候，并行的优势并不能很好地体现出来，可以看到串行的运行时间要比并行的运行时间略短。可能的原因是线程的创建和删除的影响大于并行处理数据缩短的影响。
- 当数据量较大的时候，并行的优势就很好地体现出来。随着数据量的增加，并行处理的优势变得越来越大。此时并行处理数据缩短的影响大于线程的创建和删除所带来的消耗。
- 从数据图上分析，线程数为2应该提升40%~50%左右的性能，但是事实上只提高了10%~20%，因此后续我们可以进行优化。

6.1.4 线程数 `int threadNum`

测试代码

```

1  int threadNum[] = {1,2,3,4,5,6,7,8,9,10,15,20,30,40,50};
2  printf("结果存放于./result/result_threadNum.csv\n");
3  for(int i = 0;i < 15;++i){
4      float thread_run1 = 0;
5      float thread_run2 = 0;
6      Result *table = new Result[DEFAULT_ENTRY_NUM];
7      ifstream fp(data_file);
8      string line;
9      int k = 0;
10     while (getline(fp,line) && k <= DEFAULT_ENTRY_NUM - 1){
11         string number;
12         istringstream readstr(line);
13
14         getline(readstr,number,',');
15         table[k].key_ = atof(number.c_str());
16
17         getline(readstr,number,',');
18         table[k].id_ = atoi(number.c_str());
19         k++;
20     }

```

```

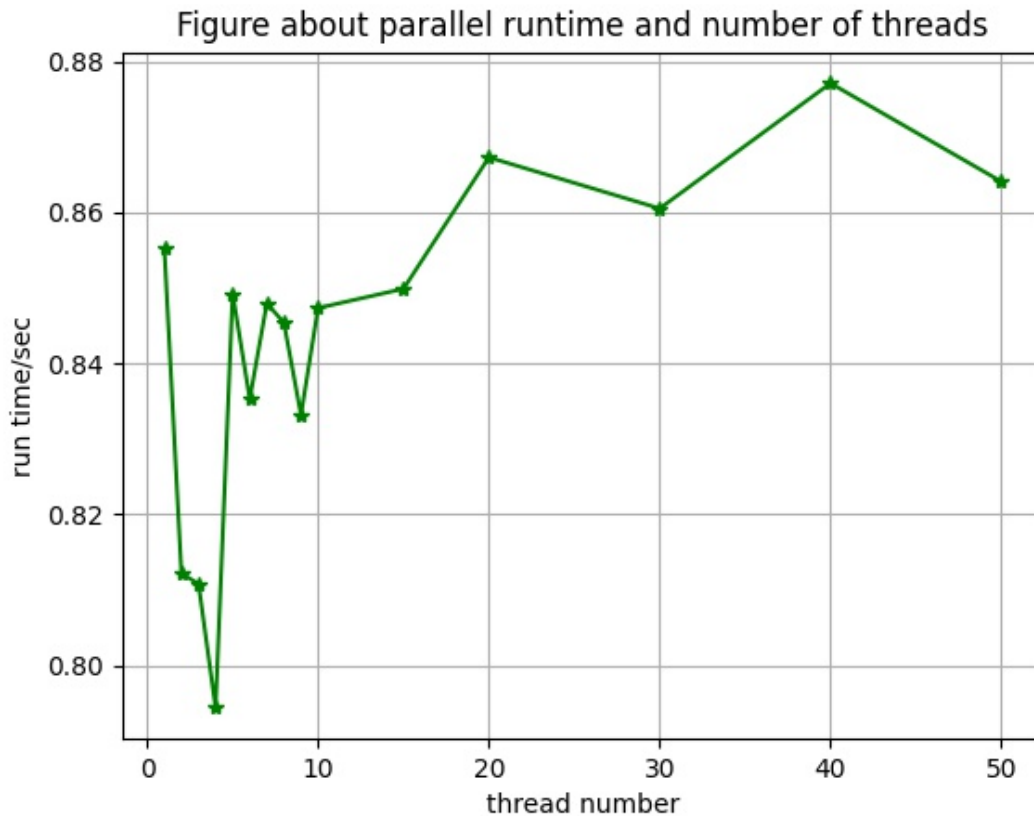
21
22         fp.close();
23
24         for(int w = 0; w < AVERAGE_NUM; w++) {
25             timeval start_t;
26             timeval end_t;
27
28             BTree* trees_ = new BTree();
29             trees_>init(B_, tree_file_ser);
30             gettimeofday(&start_t, NULL);
31
32             if (trees_>bulkload(DEFAULT_ENTRY_NUM, table)) return ;
33
34             gettimeofday(&end_t, NULL);
35             float run_t1 = end_t.tv_sec - start_t.tv_sec +
36                 (end_t.tv_usec - start_t.tv_usec) / 1000000.0f;
37             //printf("串行运行时间: %f s\n", run_t1);
38             delete trees_;
39             //-----
40             trees_ = new BTree();
41
42             trees_>init(B_, tree_file_par);
43             gettimeofday(&start_t, NULL);
44
45             if (trees_>parallelBulkLoad(DEFAULT_ENTRY_NUM, table,
threadNum[i], DEFAULT_BUFFER_SIZE)) return ;
46             gettimeofday(&end_t, NULL);
47
48             float run_t2 = end_t.tv_sec - start_t.tv_sec +
49                 (end_t.tv_usec - start_t.tv_usec) / 1000000.0f;
50             //printf("并行运行时间: %f s\n", run_t1);
51             thread_run1 += run_t1;
52             thread_run2 += run_t2;
53         }
54
55
56         if(table != NULL){
57             delete[] table;
58             table = NULL;
59         }
60         outFile<<threadNum[i]<<" "<<thread_run1/AVERAGE_NUM<<" "
<<thread_run2/AVERAGE_NUM<<endl;
61     }
62     outFile.close();

```

实验结果

实验结果也可以分为两部分，一部分是线程数小于10，由于线程数此时影响较大，我们会逐个线程数进行测试；一部分是线程数大于10后，此时线程数影响较小，我们分隔的粒度会增大，会提高到每隔5个或者10个线程数进行比较。

但是总的结果我们还是只放在一张图片上：



结果分析

- 可以看到，当线程数小于等于4时，运行时间不断地下降，而再之后却不断回升。原因在于我们的CPU处理器的核数是4，因而在线程数小于等于4时，CPU是每个线程分配一个核去运行，这时候并行处理的优势能够随着线程数的增加显示出来。
- 当线程数大于4后，运行时间整体上是不断增加的趋势，虽然中间有些许的波动。原因在于：由于CPU处理器只有4核，而线程数大于4，CPU不得不进行一定的并发处理，会轮流切换给其他线程使用，CPU的开销反而增大。

6.1.5 Buffer的大小 `int maxBufferBlock`

测试代码

```

1  int bufferSize[] = {10,50,100,200,300,500,1000,2000};
2  outFile.open("./result/result_bufferSize.csv", ios::out);
3  outFile<<"bufferSize  sertime  partime"<<endl;
4  printf("结果存放于./result/result_bufferSize.csv\n");
5  for(int i = 0;i < 8;++i){
6      float buffer_run1 = 0;
7      float buffer_run2 = 0;
8      Result *table = new Result[DEFAULT_ENTRY_NUM];
9      ifstream fp(data_file);
10     string line;
11     int k = 0;
12     while (getline(fp,line) && k <= DEFAULT_ENTRY_NUM - 1){
13         string number;
14         istringstream readstr(line);
15
16         getline(readstr,number,',');
17         table[k].key_ = atof(number.c_str());
18
19         getline(readstr,number,',');

```

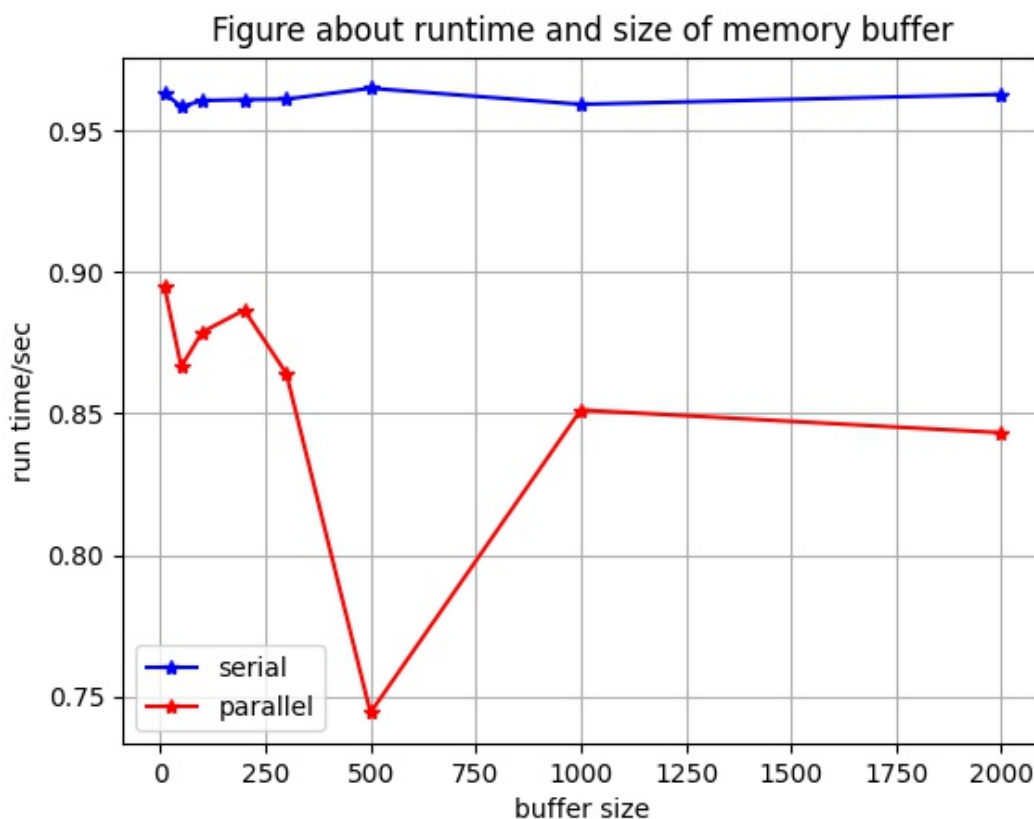
```

20         table[k].id_ = atoi(number.c_str());
21         k++;
22     }
23
24     fp.close();
25     for(int w = 0; w < AVERAGE_NUM; w++) {
26         timeval start_t;
27         timeval end_t;
28
29         BTree* trees_ = new BTree();
30         trees_>init(B_, tree_file_ser);
31         gettimeofday(&start_t, NULL);
32
33         if (trees_>bulkload(DEFAULT_ENTRY_NUM, table)) return ;
34
35         gettimeofday(&end_t, NULL);
36         float run_t1 = end_t.tv_sec - start_t.tv_sec +
37             (end_t.tv_usec - start_t.tv_usec) /
1000000.0f;
38         //printf("串行运行时间: %f s\n", run_t1);
39         delete trees_;
40         //-----
41         trees_ = new BTree();
42
43         trees_>init(B_, tree_file_par);
44         gettimeofday(&start_t, NULL);
45
46         if (trees_>parallelBulkload(DEFAULT_ENTRY_NUM,
table, DEFAULT_THREAD_NUM, bufferSize[i])) return ;
47         gettimeofday(&end_t, NULL);
48
49         float run_t2 = end_t.tv_sec - start_t.tv_sec +
50             (end_t.tv_usec - start_t.tv_usec) /
1000000.0f;
51         //printf("并行运行时间: %f s\n", run_t1);
52         buffer_run1 += run_t1;
53         buffer_run2 += run_t2;
54     }
55
56     if(table != NULL){
57         delete[] table;
58         table = NULL;
59     }
60     outFile<<bufferSize[i]<<" "<<buffer_run1/AVERAGE_NUM<<" "<<buffer_run2/AVERAGE_NUM<<endl;
61 }
62 outFile.close();

```

实验结果

实验的运行结果如下（同样是前面粒度较小而后面粒度较大）：



结果分析

- 可以看到Buffer的大小对串行的运行时间没有明显的影响。
- 而相对于并行而言：虽然前面有所起伏，但是整体上在500块Block以前运行时间是逐渐减少的。

原因可能在于：在一定限度内，随着缓冲区变大，IO的最小粒度增大，不同线程的磁盘IO和内存中节点的构建过程并行度提升，进而减少了总运行时间。

- 当Buffer的大小大于500后，运行时间却不断增加。

分析可知：由于缓冲区过大，导致所有的线程都是先将Block写入内存，到最后才轮流进行I/O，而不是I/O和写内存并行处理，或者说此时I/O和CPU的并行度不高，因而运行时间变长。

6.2 性能调优和创新优化实验结果及分析

6.2.1 性能调优的原因

1. 在上面的对entry number，也就是数据量的大小的分析中，我们可以知道，随着数据量的增加，并行的优势不断明显，但是也只是提高了10%~20%，性能以及运行时间还有很大的提升空间。
2. 在对Buffer大小的分析中，我们了解到采用的I分配内存和I/O方式是一个一个进行，为了提高性能，我们有必要进行整体（多块）地进行I/O和分配内存来提高运行效率。

我们依旧从3个影响因素进行分析。

6.2.2 数据量的大小

实验结果

同样分为数据量较大和数据量较小两部分：

Figure about runtime and entry number

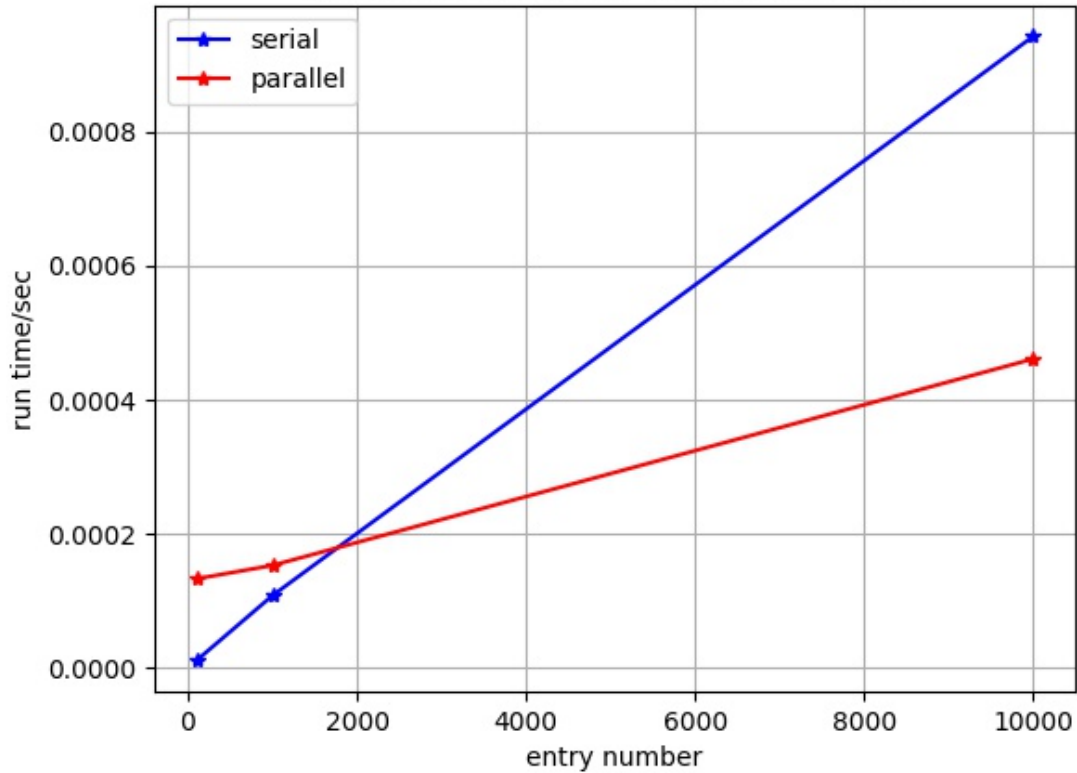
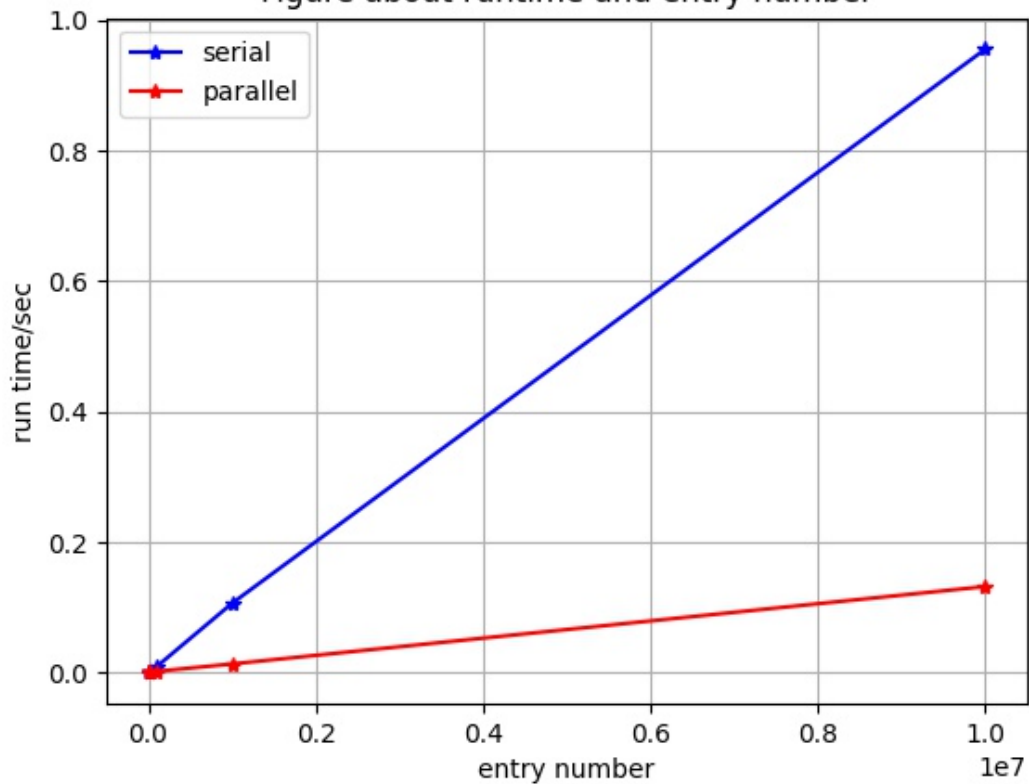


Figure about runtime and entry number

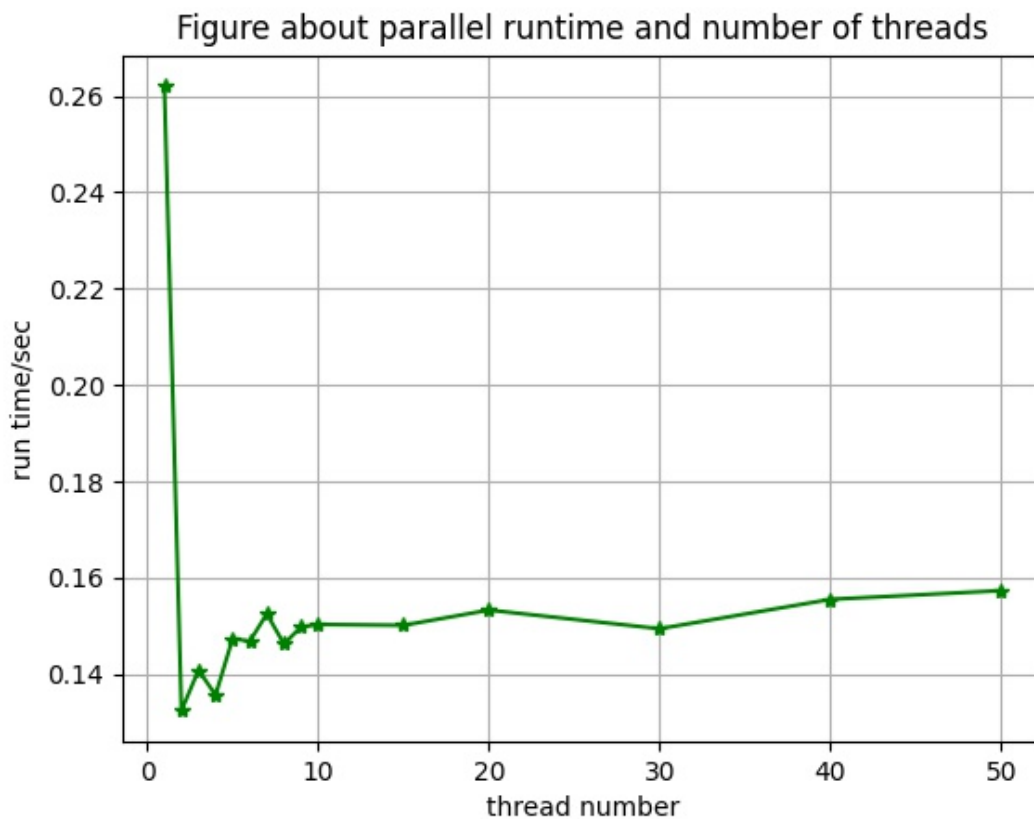


实验分析

- 在数据量较小的时候，这次在1000 ~ 2000左右并行的运行时间就优于串行的运行时间。
- 并且随着数据量的增大，并行的优势相比串行更是有了大幅度的增加。优化后的运行时长缩短了80%到90%。
- 由此可见整体分配内存和多个Block的I/O对性能有着极为重大的影响。

6.2.3 线程数

实验结果



实验分析

- 这次我们可以看到1个线程到两个线程有了明显的提高。

原因在于：1个线程的运行相当于是串行，也就是不断地取一个一个Block写内存然后I/O，运行效率很低，而2个线程不仅将数据项并行处理，还将内存的分配和I/O进行优化，运行时长大幅减小。

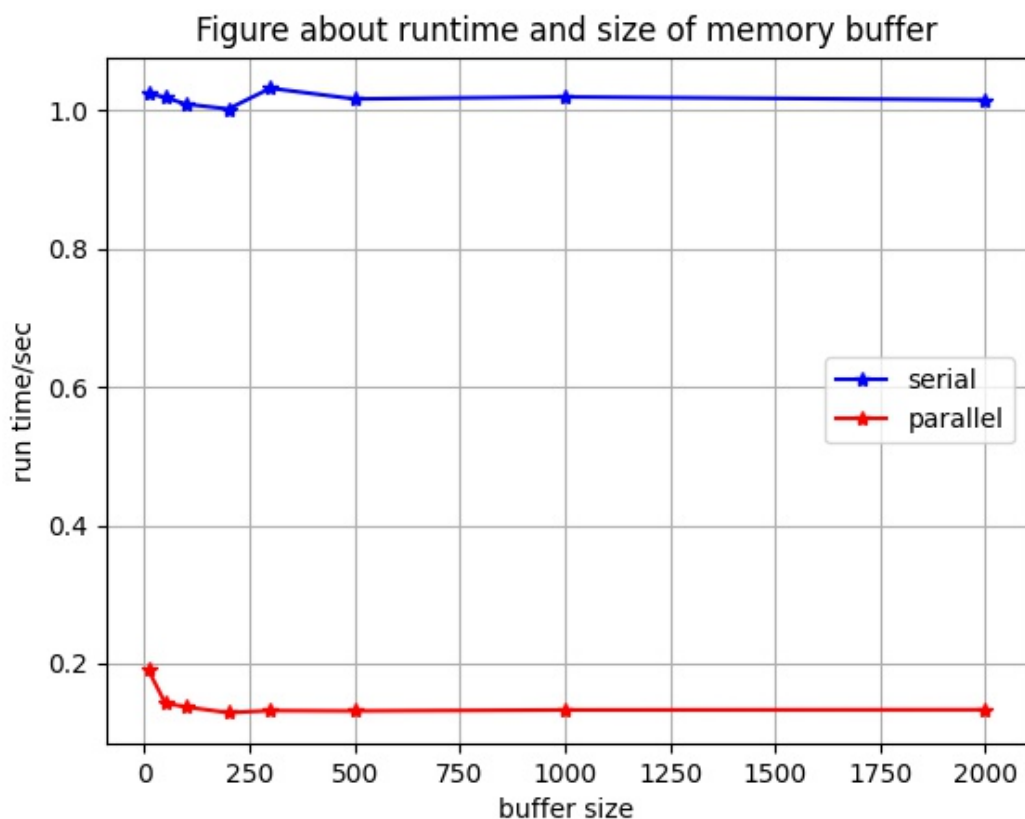
- 而2个线程之后运行时间虽然中间有所波动，但是整体上是逐渐增加的。

原因在于：此时的运行时间已经很短，为进程分配和销毁线程的开销相比处理数据的时长的影响更大。

- 线程数增加后，后面也不断趋于平稳。

6.2.4 Buffer大小

实验结果



实验分析

- 此时BufferSize的影响对于串行和运行都不是很大。
- 但是并行的情况下，随着BufferSize的增加，运行时间也是有小幅度的减小。

原因在于：随着BufferSize的增加，内存分配和I/O的粒度也不断增加，`fwrite` 的调用次数也不断减小。

- 并且在BufferSize较小的时候运行的时长有明显下降，增大之后，运行时长降幅较小。

原因：当BufferSize较小时，对I/O次数的影响较大，当BufferSize增大后，I/O次数的变化也没有那么明显，这可能是由于对IO进行优化后，IO速度带来的对整个程序运行时间的负面影响减少，所以差异不大。

VII. 总结

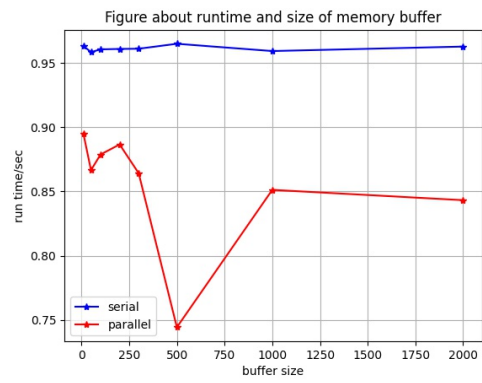
我们在串行B+树 BulkLoading 项目的基础上，设计并实现了 BulkLoading 的多核并行设计。在创新优化方面，我们设计并实现了连续分配块和连续写入磁盘的优化，让算法的性能有了较大的提升，BulkLoading 过程提速 80% 到 90%。

我们遵循控制变量和多次求平均的原则，对代码进行了多维度的测试，得到以下结论，并在报告第六节中进行了分析和解释。

1. 随着数据量的增加，并行处理的优势变得越来越大。
2. 当线程数与CPU处理器核数相近时，运行耗时最短。
3. 未经优化时，缓冲区大小为500个块时，运行耗时最短，经过优化后，缓冲区大小对运行耗时的影响并不显著。

最后结合实验分析中两张测试结果折线图，其中数据项总数为10000000，串行代码耗时在1秒左右，优化前并行代码耗时在0.75-0.9秒之间，优化后的代码耗时在0.1-0.2秒之间，可以看到我们的优化效果显著。

串行代码和优化前并行代码耗时比较



串行代码和优化后并行代码耗时比较

