数据库课程设计实验报告

一、小组成员与分工

陈序　15331038
陈宇翔　15331042
杨奕嘉　15331366
薛明淇　15331344

二、实验环境

计算机（测试环境）：Intel Core™ i3 7100 CPU / 8GB RAM
操作系统：Ubuntu 16.04
编译器：g++
编程语言：C++
协作工具：github (https://github.com/Mensu/MIPSUCT-Trans)
测试数据集：Netflix

三、实验内容

实现 Ball-Tree 的 C++外存版
任务一：实现 ball-tree 建树过程
任务二：实现将 ball-tree 写进外存的功能
任务三：实现从外存中载入 ball-tree 的功能
任务四：实现查询阶段找到最大内积对象并剪枝的功能
加分项：
任务五：实现添加和删除数据对象的功能（未实现）
任务六：实现四叉 ball-tree 的功能（未实现）

四、实验思路和设计

(一) Ball-Tree 建树、搜索部分

对于算法的基本结构，在论文中已经有相应的伪代码，这里就不重复了，主要讲一些算法实现细节，以及和存储模块对接处的细节。

1、BallTreeNode 表示

由于算法中，BallTree 的叶子和非叶子节点的存储数据略有不同，因此我们使用 BallTreeBranch 和 BallTreeLeaf 作为 BallTreeNode 的子类。其中：在 BallTreeNode 中存储 Branch 和 Leaf 共有的节点下的球心和球半径，以及在 Storage 中的存储位置 rid。在 BallTreeBranch 中额外存储树的左右指针（有两组指针，地址指针在建树时使用，RID 类型作为指针在搜索时使用），在 BallTreeLeaf 中额外存储树的叶子节点中所包含的所有向量的 RID 值。

这里我们选择在 BallTreeLeaf 中保存向量的 RID，而不是向量本身的原因在于，由于向量本身消耗存储空间较大，如果保存全部 20 个向量数据在节点中，会造成节点非常的大，也就是说这样就不能把树节点全部保存在内存中，这样就会造成两个问题：1. 树遍历时会造成树本身结构较为违反直觉，对于实现来说是一个负担。2. 更重要的是，在 Branch and Bound 的时候，需要读取叶子节点的球心和半径，来决定是否对叶子节点进行访问，而如果叶子节点的大小过于臃肿，而从外存中读取进来，如果经过计算（MIP）后发现，这个节点实际上并不需要进行遍历，那么读取大量向量的时间就是浪费了的。基于这两点，我们在叶

子节点中仅保留向量的 RID

2、BallTreeNode 的存储选择

由于所有树节点中所保存的数据相对较小（不包含真正的向量数据），因此我们选择把整棵球树在算法的全过程中保存在内存之中。这样做主要基于以下考虑：1. 实现方便，对于内存中树的遍历在 visitor pattern 的帮助下非常易于实现，遍历到叶子节点时，把所有向量从（逻辑上的）外存中取出（当然，也可能已经被保存在内存缓冲中了）进行计算就可以。2. 由于 BallTree 是二叉树，就算树的高度相对于数据量是对数级别，其高度也会是比较高的，所以在内存中对树的遍历会极大地降低 IO 的开销（或者说保存在外存中会极大增加开销）。而将树结构保存在内存中的一些顾虑包括对内存的占用是否过大，在经过粗略计算后发现，在 Yahoo 的约 60 万 300 维向量的数据下，使用这种表示的方法，对内存的消耗也仅有 100M 左右，尚在可以接受的范围内，因此将树节点完全保存在内存中是完全可行的。

3、BuildTree 函数

对于建树，算法上都是翻译伪代码来完成，没有什么新意。实现细节方面就比较注意地切分任务，使用不同的函数来进行半径、球心以及对节点的分裂，在测试时也会比较方便。另外就是充分利用 Modern C++的特性，充分利用 C++ Algorithm 库的函数式序列操作的接口（std::accumulate, std::transform 等）尽可能地简化代码，以及尽量地发挥移动语义对容器的作用，在分裂时使用 std::move_iterator 来构造新序列使得拷贝的开销降到最低。

考虑到 Leaf 中 Record 的存放问题，将同一个节点的 Record 连续存储到 Storage 中可以有效提高命中率，所以在构建 leaf 时调用了 storeAll 函数将 record 存入 storage 中。

4、Search 函数

对于搜索，我们对论文中的算法进行了简单的修改，在论文中，Branch and Bound 的操作是在遍历到一个节点的开头时进行，进入节点后对该节点下可能存在的最大内积，再决定是否进行搜索。这样做的问题在于，在决定遍历左右节点的次序时，已经计算过一遍子节点可能存在的最大内积了，计算可能的最大内积的开销是关于节点维度的 O(n)，不过能省点就省点吧。因此把 Branch and Bound 的时机提前到遍历子节点之前，也就是决定遍历的优先次序时。由于遍历过一个子树过后，当前的最大内积值可能改变，因此当一个子树遍历返回后，需要再次进行判断来避免无意义的遍历。

至于遍历的具体实现，由于 BallTreeNode 具有两种表示形式（Branch 和 Leaf），因此这里使用模式匹配在面向对象中的一个实现方式，也就是 visitor 模式来对树进行遍历。visitor 模式的一个经典应用场景也是对于内部表示形式不同的数据结构的遍历，遍历时的中间状态（当前最大内积等等）也可以保存在 visitor 的具体实现（MIPSearcher）内部。

5、和存储模块对接

我们在设计时，将上层算法，和底层的存储部分完全分离开。算法部分通过 RecordStorage 这个接口与存储进行交互，分别是 Rid RecordStorage::Put(const Record&) 和 std::unique_ptr<Record> RecordStorage::Get(const Rid&)。对于数据的这样在进行测试的时候，可以选择不同的 RecordStorage 的接口来对算法进行测试（SimpleStorage 是用于测试的实现），这样可以在完全不依赖下层存储实现的情况下确认算法部分的正确性。（测试代码在 test-algorithm.cpp 中给出，依赖是 gtest。只在开发第一阶段时测试使用，由于之后的更改现已不能使用）

(二) Storage 存入、读取部分

为了提高 IO 效率，减少系统调用的次数，外存版的实现中所有的数据均是以页为单位进行读写。每次对页内数据进行读写的时候，并不会马上写入文件，而是放在缓冲区中，等待缓冲管理器的调度，再将其从内存交换至外存。实际测试的时候，对一个单一页进行单次的读写，其速度还是比较可观的。

另一点，由于每种存入页内的数据，其长度都是相对固定的，其中节点和记录的数据的长度与数据的维度有关，而叶子节点的大小与数据的维度和 N0 的值有关，所以这里使用了定长记录和使用位图的方法进行数据的存储。

1、页面结构设计

对于一种数据类型的存储页，其存储的数据大小和数据量都是可以再初始化的时候确定的，但是从文件中读取数据的时候，为了提高效率，页的结尾部分除了位图之外，还有槽的长度（也就是数据的最大长度），以及数据的类型。这里的槽长度类型主要是为了读取的时候能够方便在不知道数据维度的情况下的得到数据的类型和长度，方便解析二进制的数据到对应的类型。

这样一来，每个页中的结构大致是这样的

数据段	位图	数据类型	槽长度
*Slot * slot_num*	*std::vector*	*Rid::DataType*	*size_t*

2、槽的设计

对于不同类型的数据，其存在于内存中的形式，页是不关心的。为了将数据从二进制的形式转换成所需要的数据类型，我们这里使用了一个专门的类管理数据的存取的时候对对应数据类型和二进制形式的装换。

槽的类不会管理内存，其读写的内存块是有对应的页类的缓冲区。为了读写页内二进制数据，先由对应类型的页产生对应类型的槽，如果是新建的槽，页会尝试在当前页内进行槽的分配。

Record 的 slot 结构

Int	Size_t	Float[data_size]
Index	Data_size	Vector data

BallTreeBranch 的 slot 结构

Size_t	Float [center_size]	Double	Rid	Rid
center_size	Vector center	Radius	Left	Right

BallTreeLeaf 的 slot 结构

Size_t	Float [center_size]	Double	Size_t	Rid[rid_size]
Center_size	Vector center	Radius	Rid_size	Vector rids

3、页面缓冲管理器的设计

页缓冲管理器将缓存一定数量的页面，存放于帧中。它对上提供 Put 和 Get 接口，从内存中找到合适的页供用户存放数据，找到指定的页获取数据。当所需页面没缓存在内存中时，管理器才从外存取回所需的页放进内存的帧中，同时将一个不常用的页写回外存，即对页面进行换入换出操作。根据局部性原理，程序接下来即将使用的数据，很有可能来自最近使用的页中。将最近使用的页缓存起来，能一定程度上减少 IO 的开销。

管理器首先需要维护一个索引文件，这里用来存放当前外存中页面的数量。每次初始化时，便从对应的索引文件中获取数量信息，用于新页面生成时钦定页面 id。每次结束使用，便将当前的数量信息写回。

其次管理器还要维护一些帧，分配给页面使用，并负责页面的换入换出操作。

4、具体管理策略

（1）管理数据

管理器存放数据时，首先要从帧缓存的页面中找到未被写满的继续写，不要轻易创建新

页面。考虑到效率问题，这里就不去外存中寻找了。若内存中的页面都被写满，则考虑创建新页面充当未被写满的页面，然后找到一个牺牲的页面换出去，空出一个槽存放新页面。开启该页面的脏位和引用位。
然后，管理器要从未被写满的页面中得到一个空闲的槽，将数据交给它存放，并根据刚才得到的页面和槽，返回一个 Rid。

（2）获取数据

管理器根据 Rid 获取数据时，首先是从帧缓存的页面中找找看有没有所需的页面。若没有，则要找到一个牺牲的页面换出去，空出一个槽，然后从外存中读取所需的页面放入该槽。

然后，管理器从所需的页面中得到指定的槽，从槽中取出数据，交给用户。

（3）替换策略

这里我们所采用的替换策略是最近最少使用策略（LRU）的变体——时钟替换策略。它与 LRU 有相似的行为，但有较少的开销。

它采用 cur 变量以环形顺序选择替换帧中缓存的页面。当某一帧被选中时，若其页面的引用位启动，则将其关闭，然后 cur 变量指向下一帧。否则，可以选择该页进行换出。若转了一圈都没有页面可以换出，则继续转，此时肯定是有一页的引用位是刚才关闭的。这时，便可以将它换出。

五、代码结构与实现

实现时没有直接在原有 BallTree 上实现，而是添加了一个指向 BallTreeImpl 类的 unique_ptr，root_，通过 BallTreeImpl 实现所有的功能。

（一）BallTree 算法部分

（1）BuildTree 函数

```
// BallTree.cpp
bool BallTree::buildTree(int n, int d, float** data) {
    impl_ = std::make_unique<BallTreeImpl>(ArrayToVector(n, d, data));
    return true;
}

//BallTreeImpl.cpp
BallTreeImpl::BallTreeImpl(Records&& records)
    : root_(BuildTree(std::move(records))) {
}

BallTreeNode::Pointer BallTreeImpl::BuildTree(Records&& records) {
    if (records.size() <= N0) {
        return BuildTreeLeaf(records);
    }
    return BuildTreeBranch(std::move(records));
}

BallTreeLeaf::Pointer BallTreeImpl::BuildTreeLeaf(const Records&
records) {
```

```
    std::vector<Rid> rids(StoreAll(records));
    std::vector<float> center(CalculateCenter(records));
    double radius(CalculateRadius(records, center));
    return BallTreeLeaf::Create(std::move(center), radius,
sptd::move(rids));
}


BallTreeBranch::Pointer BallTreeImpl::BuildTreeBranch(Records&& data) {
    std::vector<float> center(CalculateCenter(data));
    double radius(CalculateRadius(data, center));
    std::pair<Records, Records>
split_result(SplitRecord(std::move(data)));
    return BallTreeBranch::Create(
        std::move(center), radius,
BuildTree(std::move(split_result.first)),
        BuildTree(std::move(split_result.second)));
}

std::vector<Rid> BallTreeImpl::StoreAll(const Records& records) {
    std::vector<Rid> ret;
    ret.reserve(records.size());
    std::transform(
        begin(records), end(records), std::back_inserter(ret),
        [this](const Record::Pointer& record) {
            return record_storage_->Put(*record);
        });
    return ret;
}
```

由于使用了访客模式，Node 中有 Accept 函数用来接收 Visitor，后续实现中对于访问节点的部分都使用一个 Visitor 来完成。

```
// BallTreeNode.h
virtual void Accept(BallTreeVisitor& v) override {
    v.Visit(this);
}
// BallTreeVisitor.h
class BallTreeVisitor {
  public:
    virtual void Visit(BallTreeBranch*) = 0;
    virtual void Visit(BallTreeLeaf*) = 0;
};
```

（2）StoreTree 函数

```cpp
// BallTree.cpp
bool BallTree::storeTree(const char* index_path) {
    if (not impl_) {
        return false;
    }
    std::string index(index_path);
    return impl_->StoreTree(index);
}
// BallTreeImpl.cpp
bool BallTreeImpl::StoreTree(Path& index_path) {
    if (not record_storage_) {
        record_storage_ = storage_factory::GetRecordStorage(index_path, dim);
        node_storage_ = storage_factory::GetNodeStorage(index_path, dim);
    }

    NodeStorer visitor(node_storage_.get(), record_storage_.get());
    root_->Accept(visitor);

    node_storage_->PutRoot(*root_.get());

    root_ = nullptr;
    record_storage_ = nullptr;
    node_storage_ = nullptr;
    return true;
}
```

（3）RestroreTree 函数

```cpp
// BallTree.cpp
bool BallTree::restoreTree(const char* index_path) {
    std::string index(index_path);
    impl_ = std::make_unique<BallTreeImpl>(index);
    impl_->SetDimension(dim);
    return true;
}
// BallTreeImpl.cpp
BallTreeImpl::BallTreeImpl(Path& index_path):dim(-1) {
    if (not record_storage_) {
        record_storage_ = storage_factory::GetRecordStorage(index_path, dim);
        node_storage_ = storage_factory::GetNodeStorage(index_path, dim);
    }
    root_ = node_storage_->GetRoot();
```

```
}
```

（4）MipSearch 函数

使用了一个 MIPSearcher 的 Visitor 来进行搜索，结果保存在 MIPSearcher 的 cur_max_idx_ 和 cur_mip_中。

```
// BallTree.cpp
int BallTree::mipSearch(int d, float* query) {
    if (not impl_) {
        return -1;
    }
    return impl_->Search(std::vector<float>(query, query + d)).first;
}
//BallTreeImpl.cpp
std::pair<int, double> BallTreeImpl::Search(const std::vector<float>& v)
{
    if (not root_) {
        assert(false && "root is nullptr!");
        return {-1, 0};
    }
    MIPSearcher visitor(v, record_storage_.get(), node_storage_.get());
    root_->Accept(visitor);
    return {visitor.ResultIndex(), visitor.ResultMIP()};
}
//MIPSearcher.cpp
void MIPSearcher::Visit(BallTreeBranch* branch) {
    double left_mip = PossibleMip(node_storage_->Get(branch->r_left));
    double right_mip = PossibleMip(node_storage_->Get(branch->r_right));
    if (left_mip > right_mip and left_mip > cur_mip_) {
        node_storage_->Get(branch->r_left)->Accept(*this);
        if (right_mip > cur_mip_) {
            node_storage_->Get(branch->r_right)->Accept(*this);
        }
    } else if (right_mip >= left_mip and right_mip > cur_mip_) {
        node_storage_->Get(branch->r_right)->Accept(*this);
        if (left_mip > cur_mip_) {
            node_storage_->Get(branch->r_left)->Accept(*this);
        }
    }
}
void MIPSearcher::Visit(BallTreeLeaf* leaf) {
    for (const auto& rid : leaf->data) {
        auto record = record_storage_->Get(rid);
        double innerproduct = InnerProduct(needle, record->data);
        if (innerproduct > cur_mip_) {
```

```
            cur_mip_ = innerproduct;
            cur_max_idx_ = record->index;
        }
    }
}
```

（二）Storage

（1）Slot

统一的 slot 便于管理

```
// Slot.h
class Slot {
  public:
    Slot() = delete;
    Slot(const Slot&) = default;
    Slot(Byte* slot, size_t byte_size, const Rid::DataType& type): slot(slot), byte_size(byte_size), type(type) {}

    bool Get(std::unique_ptr<Record>&);
    bool Get(std::unique_ptr<BallTreeBranch>&);
    bool Get(std::unique_ptr<BallTreeLeaf>&);

    bool Set(const Record&);
    bool Set(const BallTreeBranch&);
    bool Set(const BallTreeLeaf&);

    int Size() {
      return byte_size;
    }
    void SetId(unsigned int slot_id) { this->slot_id = slot_id; }

    static size_t GetSize(Rid::DataType type, int dimension);
  private:
    Byte* slot;
    int byte_size;
    unsigned slot_id;
    const Rid::DataType type;
};
```

（2）Page

```
// Page.h
class Page {

  public:
    const int page_size;
    using Pool = Byte*;
```

```
7     using IntType = std::size_t;
8     using BitMap = std::vector<bool>;
9
10  public:
11    /**
12     * @Description Create a new page
13     */
14    Page(int page_id, IntType slot_size, Rid::DataType type, Byte*
pool_base, IntType page_size_in_k);
15
16    /**
17     * @Description Make a page from binary page file
18     */
19    Page(int page_id, std::istream& in, Byte* pool_base, int
page_size_in_k);
20    /**
21     * @Description Write data to file;
22     */
23    void sync(std::ostream& out);
24
25    /**
26     * @Description Select a slot according to solt id.
27     */
28    Slot select(const int& slot_id);
29
30    /**
31     * @Description Insert new slot in page
32     * @Return tuple of Rid and Slot.
33     */
34    std::tuple<Rid, Slot> insert();
35
36    /**
37     * @Description Drop a slot from page, just reset its bit in bitmap
38     */
39    bool drop(const int& slot_id);
40
41
42    inline bool isFull() { return m_slot_num >= m_total_slot; }
43
44    inline int PageId() const {
45      return this->m_page_id;
46    }
47
48  private:
```

```
49      /**
50       * @Description Build Bitmap from buffer memory.
51       */
52      void initBitMap();
53
54      /**
55       * @ Restore bitmap to buffer memory if necessary.
56       */
57      void restoreBitMap();
58
59      /**
60       * @Description the delegate construction
61       * @Param delegate_constructor unused.
62       */
63      Page(int page_id, IntType page_size_in_k, Byte* pool_base);
64      void init();
65
66      inline Slot makeSlot(int slot_id) {
67          return Slot(m_slot_pool + m_slot_size * slot_id, m_slot_size,
    type);
68      }
69
70   private:
71      int m_page_id;
72      bool m_dirty;
73
74      Pool m_slot_pool;
75      BitMap m_slot_map;
76      Byte* bitmap_pos;
77
78      int m_total_slot;
79      int m_slot_num;
80      IntType m_slot_size;
81      IntType m_bitmap_size;
82      Rid::DataType type;
83  };
```

（3）FixLengthStorage

```
1   // Storage.h
2   template <
3       int64_t BytesPerPage,
4       Rid::DataType DataType,
5       int64_t MaxPageInMemory = -1,
6       bool OmitZeroInNameOfFirstPage = true>
```

```cpp
class FixedLengthStorage {
    using BitSet = std::vector<bool>;
    using PagePtr = std::shared_ptr<Page>;
  public:
    FixedLengthStorage(int slot_size);

    FixedLengthStorage(
        int slot_size, const std::string& name, const Path& dest_dir = "./");

    template <typename T>
    Rid Put(const T &data);

    template <typename T>
    std::unique_ptr<T> Get(const Rid &rid);

    ~FixedLengthStorage();

    int SlotSize() const {
        return this->slot_size;
    }
  private:

    bool pageInMemory(int page_id) const {
        return this->page_to_frame_map.find(page_id) != this->page_to_frame_map.end();
    }

    bool framesFull() const {
        return this->page_to_frame_map.size() == MaxPageInMemory;
    }

    void initNewPage(PagePtr new_page_ptr, std::size_t frame_id);

    /**
     * @description 尝试找到一个牺牲页 并把它换出去 得到空槽
     * @return 返回空槽 id
     */
    std::size_t swapPageOut();

    void writePageOut(std::size_t frame_id, std::ostream &out);

    void swapPageIn(int page_in_id, std::size_t frame_id);

```

```
49      void readPageIn(int page_in_id, std::size_t frame_id, std::istream
    &in);
50
51      /**
52       * @description 如果换出后不马上换入，就会 gg，作者太菜了
53       * @return 牺牲页的 page_id
54       */
55      int findVictimPage();
56      template <std::ios::openmode openmode>
57      std::fstream getFs(int page_id);
58
59      template <std::ios::openmode openmode>
60      std::fstream getIndexFs();
61
62      void readPageNum();
63      void writePageNum();
64
65      inline Byte *getFrameAddr(std::size_t frame_id) const {
66          auto buffer_ptr = reinterpret_cast<Byte
    (*)[page_size]>(this->buffer_ptr.get());
67          return reinterpret_cast<Byte *>(buffer_ptr + frame_id);
68      }
69
70    private:
71      int slot_size;
72      std::string name;
73      Path dest_dir;
74      int page_num = 0;
75
76      BitSet is_referenced;
77      BitSet is_dirty;
78      std::vector<PagePtr> frames;
79      std::unordered_map<int, std::size_t> page_to_frame_map;
80      std::unique_ptr<Byte> buffer_ptr;
81      std::fstream fs;
82
83      static constexpr std::ios::openmode in_mode = std::ios::binary |
    std::ios::in;
84      static constexpr std::ios::openmode out_mode = std::ios::binary |
    std::ios::out;
85      static constexpr std::ios::openmode openmode = std::ios::binary |
    std::ios::in | std::ios::out;
86      static constexpr std::size_t page_size_in_k = (BytesPerPage + 1023)
    / 1024;
```

```
    static constexpr std::size_t page_size = page_size_in_k * 1024;
    static constexpr std::size_t buffer_size = page_size *
MaxPageInMemory;
    static constexpr std::size_t begin_pos = 0;
};
```

（4）NodeStorage 和 RecordStorage

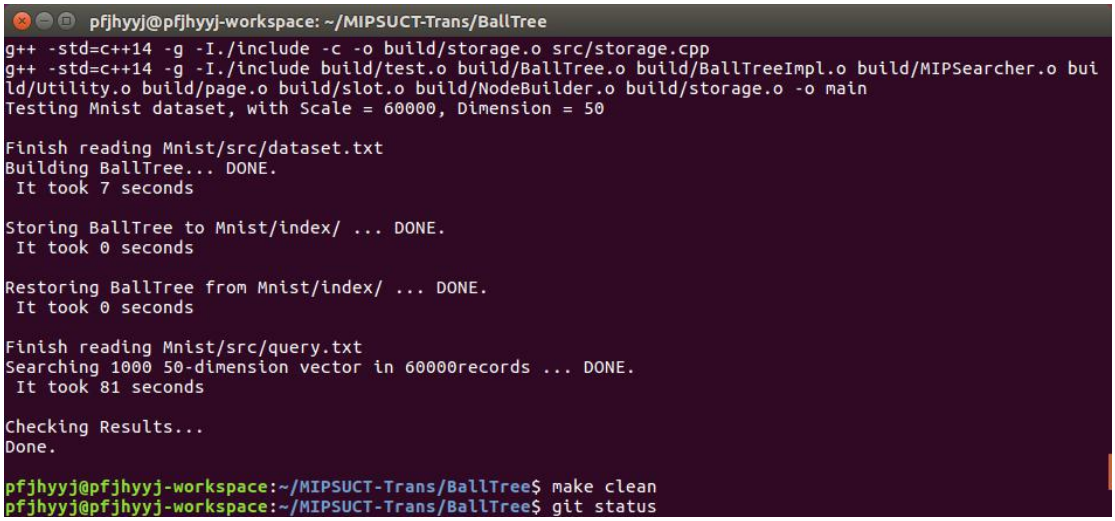
```
// Storage.h
class NormalStorage: public RecordStorage {
    public:
    NormalStorage(const Path& dest_dir, int dimension);
    virtual Rid Put(const Record& record) override;
    virtual std::unique_ptr<Record> Get(const Rid& rid) override;
    virtual void DumpTo(const Path& path) override {
        // no op
    }
    private:
    using RStorage = FixedLengthStorage<64, Rid::record, 4>;
    std::unique_ptr<RStorage> storage;
};
/**
 * storage store node
 */
class NodeStorage {
    using BranchStorage = FixedLengthStorage<64, Rid::branch, 2>;
    using LeafStorage = FixedLengthStorage<64, Rid::leaf, 2>;
  public:
    NodeStorage(const Path& dest_dir, int dimension);
    std::unique_ptr<BallTreeNode> Get(Rid rid);
    Rid Put(const BallTreeNode& node);

    std::unique_ptr<BallTreeNode> GetRoot();
    Rid PutRoot(const BallTreeNode& node);

    inline int GetDimension() {
        return m_dimension;
    }
  private:
    int m_dimension;
    Rid root;
    std::unique_ptr<BranchStorage> branch_storage;
    std::unique_ptr<LeafStorage> leaf_storage;
    Path dest_dir;
};
```

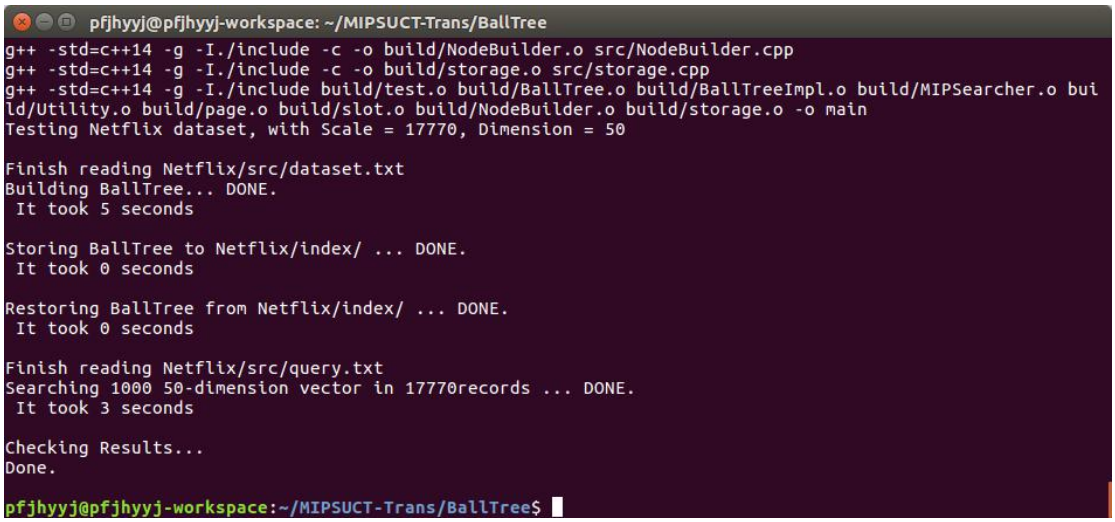
六、实验结果与性能分析

1、Minst 数据集



速度如图所示，正确率 100%

2、NetFlix 数据集



速度如图所示，正确率 100%

3、Yahoo 数据集

由于使用递归建树，数据过大无法完成建树过程。

七、心得与体会

[自我批评]

我们小组成员一开始严重低估了作业的难度，在刚拿到题目的时候粗略地看了一下，认为这个作业总体来说并不算特别困难，于是作业前几周我们都在做别的事情，最后一周才开始赶工。

事实证明这是一个很大的错误，虽然论文中球树算法的实现并不复杂，不过底层的内存

缓冲页和文件的管理才是真正困难的部分：一个是设计文件、页、帧等等结构的抽象的复杂性：如何管理页，如何管理文件，如何接触模块之间的依赖关系，如何在每个模块之间传递消息，都是需要认真仔细考虑的问题；另一个是实现的复杂性：因为操作底层数据不像实现抽象逻辑，如果有一点疏忽，错误不会像大的逻辑错误那样立马反映出来，而是会把错误累积到一定程度才使程序崩溃，或者返回一个错误的结果，这样就会非常难以解决。而这两个部分一定程度上是相辅相成的，如果两个部分任意部分没有做好，总体的难度就会上升非常多。

而我们在这两个方面都没有完全做好，因为在低估了作业难度的基础上，迫于时间的催促，很多事情不能周详地考虑，在 C++实现上很多地方也没有太考虑代码的清晰程度以及选择一些更为优雅的实现，当然也有对 C++语言本身掌握不熟练所造成的一些实现上的原因，使得后来在 Debug 的时候遇到了非常多的困难

另外，虽然算法本身被实现的速度非常的快，以及很快做好了单元测试，但是由于配合上的一些失误，在算法部分实现完毕后，负责存储的同学发现算法部分给下面的接口不能完全适配他的需要，所以根据整体的需要进行了一些重构，然而重构过后就没有再重写单元测试对算法部分进行覆盖，所有的复杂性都堆积到最后的 debug 阶段，也给我们增加了一些麻烦。

由于关于模块之间的依赖关系，如何在每个模块之间传递消息，我们只是在前期做了简要的构想和规范，对在后期实现时产生的新的困难和需求没有及时讨论，导致个人实现时为了能让自己的部分能够顺利工作，自己添加了传入参数或者自己定义了新的传出变量，不能很好地和别人的代码对接上，产生了许多 bug。在实现过程中，对于新的需求和新的接口改动，小组内的沟通还是有些不及时，导致每一次一个模块的接口的更改都要进行大规模的改动，严重影响的效率，当然其中也有我们经验不足的原因。

除了依赖问题，还有的是技术上的问题，由于有些部分水平高的组员用了更新的实现方法（c++11、c++14 标准），别的组员对新标准的理解不足或不够熟练，以至于在对接时产生了很多奇妙的 bug，例如 unique_ptr 的使用，没有解决好 get 后可能被释放的问题，

[心得体会]

每一次小组实验都能体会到 git 工作流的便利性，特别是这次的实验是一个比较大的 project，利用 github 进行小组协作，git 工作流的特性为我们提供了很多便利，例如利用 branch 分开模块开发、实现错误时及时回滚等等，对于合作开发有着极大的优势。

同时要善用单元测试，由于整个实验的复杂性，如果要从底层一个个来进行 gdb 调试是要花费海量的时间的，通过完善的单元测试能剩下很多的时间。点名表扬某同学的 test-algorithm 确保了算法的准确性，我们不需要在 BallTree 的算法上花费更多时间，而是专心对 Storage 部分 debug。

同时这次实验给我们很多的教训：

其一就是永远不要低估任意一项任务，不论是什么任务都要以最认真的态度和精神去对待，因为只有着手去做了之后才能发现任务中真正含有的坑，很多一眼看上去容易的事情内部其实也会含有非常多的细节。

另外就是写一份代码永远要考虑设计问题，如果团队成员之间配合更加紧密一些，集合大家的智慧一开始就把类的分工以及接口设计清楚，这样在实现的过程中，就算发生了实现上的困难，也可以通过单元测试的方式分别测试每一个模块，这样就可以把最后 debug 的困难降低非常多：如果保证每一个实现细节都是正确的话，那么最后无意义的检查就会减少非常多。

其三就是对于实现的重视一定要加强，因为设计不仅仅存在于大结构之中，还存在于每个模块的实现之中，如何让自己容易写出单元测试，写出来的单元测试很大程度上能反映自

己代码的正确性，也是非常非常重要的，这也是我们这一次产生大失误的一个很大方面。

不过说来说去，全部推锅给时间紧迫是不公平的，其实还是我们平时写代码的时候习惯不够好，没有模块化，以抽象思维思考，以及严谨处理代码的习惯，才导致在做的过程中遇到这样那样的问题。所以也许这并不是我们没有时间注意，而是没有这个习惯去注意，真正需要注意的时候就会感觉非常不自然，要特地的分精力，分时间去特地注意这些问题。

因此我想这次作业之后，我们对软件设计和开发的认识会更加深入：知道一个软件里不仅仅只是时间和代码的堆积和消耗，俗话也说磨刀不误砍柴工，其实融入在代码中的思想和思维也许比代码本身还会更加重要一些。这样的思想也许以后也能推广到生活的不同方面，而不光只是在工作当中——大概这就是抽象思维吧哈哈哈。

其实客观来说，以及我们团队队员事后反思来说，这次作业本身的绝对困难程度也许也并不是特别的大，一定没有我们一些团队成员参加的项目的难度来的高，考虑的事情来得多。不过作业反映出来我们自己的问题是非常非常大的，不过出了问题其实对我们的未来也都是好事，我们的团队协作的精神和素养也在这次作业中极大地提升了，这次过程中所带来的教训和经验对我们今后都是一个宝贵的财富。

不过虽然说了这么久问题，不过我们团队队员本身也不是非常有软件工程的经验，所以出了一些问题也是在所难免。从正面角度上说，在出了这么多问题，时间这么紧迫的情况下，经过队员们的顽强拼搏和奋斗精神，也完成了任务。在一些地方起了摩擦和冲突之后也能迅速进入状态完成开发，其实是非常非常不容易的。虽然代码风格和思想上出现了问题，而且大多队员平时使用的主力语言也都不是 C++，但是最终这个任务还是完整地实现了出来，说明队员们的编码的硬实力还是非常强的。虽然在整个过程中出现了这样那样的问题，我们在这之中反映出来的精神和水平仍然也是可圈可点的！

感谢这次作业给我们每个队员的大学生活带来的有趣而难忘的体验。

【鞠躬】