

# 中国地质大学

## 本科生课程设计报告封面

课程名称\_\_\_\_\_大数据存储与管理\_\_\_\_\_

教师姓名\_\_\_\_\_邓泽\_\_\_\_\_

本科生姓名\_\_\_\_\_钱晓楠\_\_\_\_\_

本科生学号\_\_\_\_\_20221003675\_\_\_\_\_

本科生专业\_\_\_\_\_数据科学与大数据技术\_\_\_\_\_

所在院系\_\_\_\_\_计算机学院\_\_\_\_\_

电话: \_\_\_\_\_18571527085\_\_\_\_\_

日期: \_\_\_\_\_2024.7.6\_\_\_\_\_

## 独创性声明

本人郑重声明：所呈交的课程设计，是本人在老师的指导下，独立进行开发工作所取得的成果。本课程设计不包含任何其他个人或集体已经发表或编写过的作品成果。本人完全意识到本声明的法律结果由本人承担。

课程设计作者签名：

签字日期： 2024 年 7 月 16 日

# 课程设计评语

对课程论文的评语:

平时成绩:	课程论文成绩:
总 成 绩:	评阅人签名:

- 注：1、无评阅人签名成绩无效；  
2、必须用钢笔或圆珠笔批阅，用铅笔阅卷无效；  
3、如有平时成绩，必须在上面对评分表中标出，并计算入总成绩。

# 目录

课程设计评语 .....	3
1. 题目 .....	5
2. 设计思路 .....	6
2.1 题目一 .....	6
2.2 题目二 .....	7
2.3 题目三 .....	7
3. 程序设计 .....	8
3.1 题目一 .....	8
3.1.1 数据清洗 .....	8
3.1.2 连接 MongoDB 数据库 .....	8
3.2 题目二 .....	9
3.2.1 实现 MyBloomFilter 类 .....	9
3.2.2 实现 Deal 类 .....	9
3.2.3 实现 CuckooFilter 类 .....	11
3.3 题目三 .....	13
3.3.1 B $\epsilon$ 树索引的实现 .....	13
3.3.2 ALEX 索引的实现 .....	18
4. 算法思想及流程 .....	26
4.1 题目一 .....	26
4.2 题目二 .....	26
4.3 题目三 .....	28
5. 程序代码 .....	31
1dataclean.py: .....	32
2save_to_mongo.py: .....	33
3bf_query.py: .....	34
4cf_query.py: .....	37
5func3_had_buffer.py: .....	41
6alex_try0.py: .....	46
6. 程序运行结果 .....	54
6.1 题目一运行结果 .....	54
6.2 题目二运行结果 .....	55
6.3 题目三运行结果 .....	56
7. 致谢 .....	59

# 1. 题目

1. 将每条书籍评分记录存储至 Mongdb 中，并清理 NULL 值。从数据集中可以发现出版时间的数据格式多样，有 1999,2012/12,1923-4,2019 年六月，因此需要提取出其年份。最后把文件下载到本地（25 分）

	书名	作者	出版社	出版时间	评分	评论数量
1	潮骚	黄伟文	milk magazine	2005	8.1	181
2	特殊传说1 入学!不存在的学园!	护玄	威向文化	2007/07/05	8.8	180
3	漫长迂迴的路	亦舒	天地圖書	2005	7.1	179
4	学校不教的性爱课	走走	浙版数媒	2013/11/1	5.5	179
5	永远的尹雪艳	白先勇	长江文艺出版社	1993	7.8	178
6	不要放棄春天	亦舒	天地圖書	None	6.7	178
7	颠覆式创新：移动互联网时代的	李善友	李善友	2015/3/1	7.8	176
8	SC4	合集	Special Comix	2011/1/8	8.5	175
9	孤独与沉思	[法] 苏利·普吕多	漓江出版社	1991	8.6	174
10	花園的旋轉木馬	柏木晴子	东立出版社	2002	7.7	174
11	關照你的花蕾	みなみ遥	尊龍	2009	7	172
12	陰陽師：天鼓卷	夢枕獃	繆思	2012/5/30	8.7	172
13	LES YEUX DU CHAT	Moebius	Les Humanoïdes	初版1978 新版1991	9.1	170
14	Kellys私房口金包	Kelly	雅书堂	39612	8.6	168
15	書·設計	Works Corporation	積木	39955	8.9	167
16	天下沒有懷才不遇這回事	包益民	圖神	20041130	7.7	167
17	迈向质朴戏剧	[波兰]耶日·格洛托	中国戏剧出版社	1984/7/1	8.6	166
18	留英學生日誌	亦舒	天地圖書	1982	6.9	164
19	密室推理杰作选一一欧美卷	爱德华·D. 霍克 等	谜斗篷	2011/11	8.6	164
20	校舍的後方埋藏著天使 1	小山鹿梨子	尖端出版	2012/11/8	7.7	164
21	Hokkaido Daido Moriyama	森山大道	None	2008	9.3	163
22	玉历宝钞	None	None	None	8	163
23	不存在未出版的错误条目	None	None	None	7.4	162
24	SC5	合集	獵和出版社	2012/8/29	8.7	161
25	舊歡如夢	亦舒	天地圖書	1984/	7	159

2. 由于相同的书籍可能出于多个出版社，为避免书籍评分重复导入，实现 BF 过滤器，每当有一部书籍被存储后将其加入 BF 过滤器，并能够使用 BF 过滤器查询上述书籍是否已经被存储。如果已经被存储则以列表的形式更新出版社的属性值，同时对于其评分和评论数量也需更新（20 分）

	书名	作者	出版社	出版时间	评分	评论数量
37904	茶花女	小仲马 (Alexandre	凤凰出版传媒集团	2011/11/1	8.4	47
49477	茶花女	[法] 小仲马	上海三联书店	2009/5	8.1	201
52818	茶花女	小仲马	外语教研	2006/12	8.3	46
55684	茶花女	(法) 小仲马	上海译文出版社	1993/10	8.3	284
57217	茶花女	小仲马	广州出版社	2008/1	7.5	13
57349	茶花女	亚历山大·小仲马	中央编译	2010/5	7.7	10
57628	茶花女	[法]小仲马 原 陆	上海人民美术出版社	2001/1	7.9	135
58243	茶花女	小仲马	None	2003/8/1	8.4	260
58336	茶花女	(法) 小仲马 (Dum	华夏出版社	2007/10/1	8.4	64
58654	茶花女	小仲马	外国文学出版社	1997/03/01	8.1	44657
58699	茶花女	(法) 小仲马	少年儿童出版社	2002/06	7.9	205
59201	茶花女	[法] 小仲马	上海译文出版社	1994/7	8.5	726

加分项：实现 CF 过滤器完成上述功能（15 分）

3. 为了判断近些年哪些书值得一读，根据书籍评分记录中的“出版时间”和“评分数量”构建 B<sup>+</sup>-Tree 索引。根据 B<sup>+</sup>-Tree 索引实现书籍出版时间和评论数量的范围查询，例如将出版时间在 1990 和 2020 之间以及评论数量大于 50000 的书籍提取出来，并按照评分降序排序，输出前十，结果如下：（25 分）

	书名	作者	出版社	出版时间	评分	评论数量
0	小王子	[法] 圣埃克苏佩里	人民文学出版社	2003	9.0	209602
1	白夜行	[日] 东野圭吾	南海出版公司	2008	9.1	170493
2	围城	钱锺书	人民文学出版社	1991	8.9	178288
3	红楼梦	[清] 曹雪芹 著	人民文学出版社	1996	9.5	111576
4	解忧杂货店	(日)东野圭吾	南海出版公司	2014	8.6	160063
5	活着	余华	南海出版公司	1998	9.1	118521

加分项：选择书籍评分记录中的“出版时间”和“评分数量”任意一个属性值来构建基于 ALEX 的学习索引。（15 分）

## 2. 设计思路

### 2.1 题目一

首先读取数据集中的数据，经过观察后发现，原数据存在以下特征：

- 1、数据中存在大量的 None 值以及 0 值。
- 2、“出版时间”有多种格式，如 2015/3/1、民国 77/10 月、民国 56、2004/8/4、1984/2 月第一版、初版 1978 新版 1991、20060608、Jan 01, 2003、39337 等。
- 3、针对“出版时间”的多种格式，有的明显不是年份数据如 39337，所以我们设置正则表达式，将数字的位数控制在 4 位数。针对其他格式的数据，我们在整个字符串中进行匹配，并将前两位数字设置为 19 或者 20，查找满足前两位为 19 或者 20 的四位数。这样不管是年份出现在字符串前端、中间还是后端，都能够与正则表达式进行匹配。

**数据处理：**首先将含有 None 或者空值的数据行直接删除。再定义一个正则表达式提取出正确的“出版时间”。将经过处理后的数据保存到 cleandata.csv 文件中。  
**存入 MongoDB 数据库：**利用 python 中的库进行与 MongoDB 的连接，然后将 cleandata.csv 文件中的数据转为字典类型存入到 MongoDB 中名为 book\_collectio

n 的集合当中。最后将 `book_collection` 中的数据逐一打印，实现输出。

## 2.2 题目二

程序实现了两个功能：

一个是简单查询书籍是否存在，即仅仅将所有书名插入到过滤器中，再输入书名，通过过滤器查询书名是否已经存在，输出查询结果“yes”或者“no”。

另一个是查询精确查询书籍信息，返回书籍的“出版社”、“出版时间”、“评分”以及“评论数量”等等各种信息。这需要在简单查询的基础上，在 MongoDB 数据库中进一步查找书籍对应的信息。

实现过滤器类 `MyBloomFilter` 与 `CuckooFilter`，利用两个 MongoDB 的集合，一个为集合即之前已经存入好清洗之后数据的 `book_collection`，另一个集合 `book_info` 用于存放经过过滤之后的数据。设计一个 `Deal` 类，用于处理 `book_collection` 当中的数据：首先遍历 `book_collection` 当中的数据，每次遍历到一个数据时，将它的“出版社”、“出版时间”、“评分”以及“评论数量”改为列表类型。判断该数据的书名是否已经加入到过滤器中，如果没有加入到过滤器，那么将该数据直接加入到 `book_info` 当中。若该书名已经加入到过滤器中了，获取当前书名在 `book_info` 中存入的数据 `existing_book`，然后将当前数据的“出版社”、“出版时间”、“评分”以及“评论数量”加入到 `existing_book` 对应的列表中，再利用 MongoDB 中的更新语句实现四个列表的更新，完成数据向列表的加入。

## 2.3 题目三

实现 B+ 树索引，进行范围查找时对 `book_collection` 中的数据进行一次按照“出版时间”的范围查找，进行第一次筛选，将第一次筛选后得到的结果存入到 `result1` 数组中。然后再以 `result1` 中的“评论数量”为索引，再次进行一次范围查找，得到最后满足条件的数据 `result`。最后对 `result` 按按照“评分”进行排序，输出评分排名前十的数据。

构建基于 ALEX 的学习索引时，按照其基本结构进行实现。基本结构基于 B+ 树和 RMI 结构改进，每个结点上有一个线性回归模型和一部分连续存储空间（数组）。内部结点的数组存储关键字以及叶节点的指针。叶结点（数据结点）有两个数组，一个用来存储关键字，另一个保存关键字对应的其他信息。其中关键字数据利用 Gapped Array (GA) 保留一些间隔，用来提高数据动态更新时的性能。于是针对这种特定类型的数组新建一个 `GappedArray` 类，设置数组长度个 `flag` 来标记该位置是否为空。

使用线性回归模型来拟合数组中的数据分布，由于数据结构为 `GappedArray`，故预测时需要对 `x` 与 `y` 进行处理，只利用不为空数据进行预测。范围查找是选取 1970-1990 年的出版时间作为查询区间，按照评分降序，并输出前十。

## 3. 程序设计

### 3.1 题目一

#### 3.1.1 数据清洗

(1) 删除包含 None、0 或者为空的行

```
df.replace(['None', 0, ""], np.nan, inplace=True)
```

```
df.dropna(inplace=True)
```

replace 方法将数据框中所有包含 ‘None’、0 或空字符串的单元格替换为 NaN（缺失值）。

dropna 方法删除包含 NaN 值的行，以确保数据的完整性和一致性

(2) extract\_max\_year(date\_str):用正则表达式从字符串中提取所有的四位数年份，并返回最大的年份值。

```
if isinstance(date_str, str):
    year_matches = re.findall(r'\b(?:19|20)\d{2}\b', date_str)
    if year_matches:
        years = [int(year) for year in year_matches]
        return max(years)
return np.nan
```

'\b(?:19|20)\d{2}\b'这个正则表达式可以匹配任何以 19 或 20 开头的四位数年份，不论是出现在字符串的开头、中间还是结尾，并确保四位数字紧密相连，也就是作为一个独立的完整单词存在于文本中。

(3) 选取合适的列

```
df = df[['Unnamed: 0', '书名', '作者', '出版社', '出版时间', '评分', '评论数量']]
```

由于第一列为序号，故我们不存储。

(4) 将清洗后的数据存入表格 cleandata.csv 中。

#### 3.1.2 连接 MongoDB 数据库

# 连接 MongoDB 数据库

```
client = MongoClient("mongodb://localhost:27017/")
```

```
db = client['book_database'] # 创建或连接数据库
```

```
collection = db['book_collection'] # 创建或连接集合
```

# 将 DataFrame 转换为字典并插入到 MongoDB

```
data = df.to_dict(orient='records')
```

```
collection.delete_many({})
```



collection.insert\_many(data)

## 3.2 题目二

### 3.2.1 实现 MyBloomFilter 类

变量:

- (1) DEFAULT\_SIZE: 布隆过滤器的默认大小, 这里设置为  $2^{24}$  位。
- (2) SEEDS: 用于哈希函数的种子列表, 这里种子数量为 6 个, 对应了 6 个哈希函数。

内部类 SimpleHash:

- (1) 参数:  
cap: 表示哈希表的容量大小。  
seed: 是用来进行哈希运算的种子值。
- (2) 函数方法:  
hash(self, value): 计算哈希值的哈希函数。

```
def hash(self, value):  
    result = int(hashlib.md5(value.encode('utf-8')).hexdigest(), 16)  
    return (self.seed * result) % self.cap
```

成员函数:

- (1) \_\_init\_\_(self): 初始化方法, 创建布隆过滤器的位数组和多个哈希函数。  
bits: 使用 bitarray 库创建一个位数组, 并将所有位初始化为 0。  
func: 使用 SimpleHash 类生成多个哈希函数。  
代码: self.func = [self.SimpleHash(self.DEFAULT\_SIZE, seed) for seed in self.SEEDS]

- (2) add(self, value): 将值添加到布隆过滤器中。

```
def add(self, value):  
    for f in self.func:  
        self.bits[f.hash(value)] = 1
```

遍历所有的哈希函数并设置对应位为 1。

- (3) contains(self, value): 检查值是否存在于布隆过滤器中。

```
def contains(self, value):  
    return all(self.bits[f.hash(value)] for f in self.func)
```

使用所有的哈希函数检查对应位, 如果所有位都为 1, 则返回 True, 否则返回 False。

### 3.2.2 实现 Deal 类

变量:

- (1) filter: MyBloomFilter 对象。
- (2) mongo\_client: MongoDB 客户端对象，连接本地 MongoDB 服务器。
- (3) database: MongoDB 数据库对象，指向 book\_database 数据库。
- (4) collection: MongoDB 集合对象，指向 book\_collection 集合，存储了未经过滤器过滤之前的数据。
- (5) book\_info\_collection: MongoDB 集合对象，用于存储 book\_collection 经过过滤器过滤后的带有列表类型的数据。

### 成员函数：

- (1) \_\_init\_\_(self, filter): 初始化方法，接收一个 MyBloomFilter 对象作为参数。  
初始化 MongoDB 客户端和相关集合。
- (2) convert\_books\_to\_list(self): 将所有数据过滤后存入到 book\_info 集合中。  
首先将 book\_info 集合中的数据全部清空，然后调用 store\_book 函数将 book\_collection 中的书籍信息转存到 book\_info 集合中。
- (3) store\_book(self, book): 传入参数 book，通过判断书籍是否已经存在于过滤器当中，采用不同的处理方式将数据存入到 book\_info 集合中。  
如果书名已经被添加到过滤器中了，则更新书籍的出版社、出版时间、评分和评论数量；否则，将其作为新书籍添加到集合中。  
详细解释见下方重点代码介绍。
- (4) query\_book(self, title): 查询特定书籍的详细信息。  
使用布隆过滤器确认书籍是否存在，如果存在则在 book\_info\_collection 中使用 find\_one 函数返回该书籍的详细信息，否则返回 None。
- (5) print\_all\_books(self): 打印所有存储在 book\_info\_collection 中的书籍信息。

### 重点代码介绍：

```
def store_book(self, book):
    book_title = book['书名']
    if self.filter.contains(book_title):
        existing_book = self.book_info_collection.find_one({'书名': book_title})
        #在 book_info_collection 中提取已经存储了书名的 book
        if existing_book:
            # 更新出版社
            publishers = existing_book.get('出版社', []) #得到 ‘出版社’
            # 列表，为空的话则返回空列表
            publishers.append(book['出版社'])

            # 更新出版时间
            times = existing_book.get('出版时间', [])
            times.append(book['出版时间'])

            # 更新评分
```

```

        scores = existing_book.get('评分', [])
        scores.append(book['评分'])

        # 更新评论数量
        reviews = existing_book.get('评论数量', [])
        reviews.append(book['评论数量'])

        self.book_info_collection.update_one(
            {'书名': book_title},
            {'$set': {
                '出版社': publishers,
                '出版时间': times,
                '评分': scores,
                '评论数量': reviews
            }}
        )
    else:
        self.filter.add(book_title)
        book['出版社'] = [book['出版社']]      #这里将原本的数据类型更
新为列表
        book['出版时间'] = [book['出版时间']]
        book['评分'] = [book['评分']]
        book['评论数量'] = [book['评论数量']]
        if '_id' in book:
            del book['_id'] # 移除 _id 字段
        self.book_info_collection.insert_one(book)

```

### 3.2.3 实现 CuckooFilter 类

**变量：**

- (1) size: cf 过滤器的总大小
- (2) bucket\_size: 每个桶的大小，默认为 4，每个桶可以容纳 4 个元素
- (3) max\_kicks: 插入元素时的最大踢出次数，防止无限循环
- (4) buckets: 存储桶的列表，初始化时为包含 size 个空列表的列表  
`self.buckets = [[] for _ in range(size)]` #由 size 个空列表组成
- (5) hash1 和 hash2: 两个哈希函数，用于计算元素的哈希值。

**成员函数：**

- (1) `__init__(self, size, bucket_size=4, max_kicks=500):`  
 初始化方法，接受三个参数：过滤器大小、每个桶的大小和最大踢出次数。  
 初始化哈希表和哈希函数。设置最大踢出次数为 500。

(2) `get_hash_function(self)`: 生成一个哈希函数，使用 MD5 哈希算法计算字符串的哈希值，并对 `size` 取模，保证哈希值不超出范围。

```
def get_hash_function(self):
```

```
    return lambda x: int(hashlib.md5(x.encode('utf-8')).hexdigest(), 16) % self.size
    #用 hashlib 中的 md5 算法计算字符串 x 的哈希值，然后将而且对 self.size 取模
```

(3) `fingerprint(self, item)`: 生成元素的指纹，使用 SHA-1 哈希算法，并取前四位作为指纹。

```
def fingerprint(self, item):
```

```
    return hashlib.shal(item.encode('utf-8')).hexdigest()[:4] #指纹取前四位
```

(3) `add(self, item)`: 将元素添加到过滤器中。

计算元素的指纹和两个哈希值 `i1` 和 `i2`。尝试将指纹插入 `i1` 或 `i2` 对应的桶中。如果两个桶都已满，则进行踢出操作，直到成功插入或达到最大踢出次数。

详细介绍见下方重点代码介绍。

(4) `contains(self, item)`: 检查元素是否在过滤器中。

计算元素的指纹和两个哈希值 `i1` 和 `i2`，如果指纹在任意一个桶中，则返回 `True`，否则返回 `False`。

### 重点代码介绍:

```
def add(self, item):
    # 计算元素的指纹
    fp = self.fingerprint(item)
    # 计算第一个哈希值
    i1 = self.hash1(item)
    # 计算第二个哈希值，使用指纹进行计算
    i2 = i1 ^ self.hash2(fp)

    # 检查第一个桶是否有足够的空间
    if len(self.buckets[i1]) < self.bucket_size:
        # 将指纹添加到第一个桶中
        self.buckets[i1].append(fp)
        return True

    # 检查第二个桶是否有足够的空间
    if len(self.buckets[i2]) < self.bucket_size:
        # 将指纹添加到第二个桶中
        self.buckets[i2].append(fp)
        return True
```

```

# 两个桶都满了，执行桶溢出处理
# 选择桶内元素较少的桶作为起始桶
i = i1 if len(self.buckets[i1]) < len(self.buckets[i2]) else i2
# 最多允许的重新插入次数
for _ in range(self.max_kicks):
    # 弹出起始桶中的第一个元素
    evicted_fp = self.buckets[i].pop(0)
    # 将新的指纹插入到起始桶的末尾
    self.buckets[i].append(fp)
    # 更新指纹为被替换的指纹
    fp = evicted_fp
    # 计算新的第二个哈希值
    i = i1 if i == i2 else i2
    i2 = i ^ self.hash2(fp)

    # 检查更新后的桶是否有足够的空间
    if len(self.buckets[i]) < self.bucket_size:
        # 将指纹添加到更新后的桶中
        self.buckets[i].append(fp)
        return True

# 若所有桶都满且无法插入，则返回 False
return False

```

### 3.3 题目三

全局变量：

# 连接到 MongoDB

mongo\_client = MongoClient('mongodb://localhost:27017/')

database = mongo\_client['book\_database'] #MongoDB 中的数据库名称

collection = database['book\_collection'] #存储书籍信息的 MongoDB 集合

#### 3.3.1 B+树索引的实现

##### 1、BeTreeNode 类

变量：

- (1) t: 最小度数 (Minimum degree)，决定每个节点的最小和最大子节点数。
- (2) leaf: 表示该节点是否为叶子节点。
- (3) keys: 用于存储节点的键值对，是一个二维列表，keys[i][0]为索引，keys[i][1]。为数据值

(4) **children**: 用于存储结点的子节点。

(5) **buffer**: 列表，作为缓冲区，暂时存储键值对，缓冲区的大小为 **t**，由于缓冲区的大小为 **t**，故将缓冲区中的数据加入节点中时不会造成溢出。

### 功能函数：

(1) **\_\_init\_\_(self, t, leaf=False)**: 初始化节点，设置最小度数和是否为叶子节点，并初始化键、子节点和缓冲区。

(2) **insert\_to\_buffer(self, k, data)**: 将键值对 (**k**,**data**) 插入缓冲区。如果缓冲区已满，则触发 **flush()** 操作，**flush()** 将会把已经满了的缓冲区中的数据转移到对应的地方。

(3) **flush(self)**: 处理缓冲区中的内容，将缓冲区中的数据清空。如果是叶子节点，直接将键值对插入到 **keys** 的 **buffer** 中；否则，将键值对传递给子节点，子节点也是优先插入到 **buffer** 当中。

如果当前节点是叶子节点，那么首先对 **buffer** 中的数据按照 **k** 进行排序，确保索引的有序。然后，直接将缓冲区中的键值对插入到当前节点的键列表 (**keys**) 中。对于每个键值对 (**k**, **data**)，调用 **\_insert\_non\_full(self, k, data)** 方法将其插入到当前节点的键列表中。

如果当前节点不是叶子节点，那么遍历缓冲区中的每个键值对 (**k**, **data**)。确定要将键值对插入的子节点位置 **i**，从当前节点的键列表 **keys** 中找到合适的位置，使得 **k** 大于或等于 **keys[i][0]** 之后的所有键。如果子节点的缓冲区已满（缓冲区长度大于等于阈值 **self.t**），则递归调用子节点的 **flush()** 方法，将其缓冲区中的数据插入到子节点的键列表中。若未滿，将当前键值对 (**k**, **data**) 添加到子节点的缓冲区中。由于为一个个的遍历缓冲区中的键值对，故每次添加都会检查缓冲区是否已满。

```
def flush(self):
    if self.leaf: # 如果当前节点是叶子节点
        # 首先对 buffer 中的数据按照 k 进行排序，确保索引的有序
        buffer= sorted(self.buffer, key=lambda x: x[0])
        # 将缓冲区中的键值对插入到当前节点的键列表中
        for k, data in self.buffer:
            self._insert_non_full(self, k, data)
    else: # 如果当前节点不是叶子节点
        # 遍历缓冲区中的每个键值对，将其传递给合适的子节点
        for k, data in self.buffer:
            i = len(self.keys) - 1
            # 找到要插入的子节点位置 i
            while i >= 0 and k < self.keys[i][0]:
                i -= 1
            i += 1
            # 检查子节点的缓冲区是否已满，若已满则递归刷新子节点
```

```

        if len(self.children[i].buffer) >= self.t:
            self.children[i].flush()
        # 将键值对 (k, data) 添加到子节点的缓冲区中
        self.children[i].buffer.append((k, data))
# 清空当前节点的缓冲区
self.buffer = []

```

(4) `_insert_non_full(self, x, k, data)`: 节点 `x` 不是满的情况下, 插入键值对(`k, data`)。

分为两种情况:

1、插入到叶子节点 (if `x.leaf`):

如果 `x` 是叶子节点 (`x.leaf` 为 `True`), 则将 `(k, data)` 插入到 `x.keys` 中。确保有足够的空间来存放新的键-值对 `(k, data)`, 必要时移动现有的键以腾出空间。最后, 根据 `k` 的值将 `(k, data)` 插入到正确的位置, 由于 `flush()`操作将缓冲区清空时会调用 `_insert_non_full()`函数, 故保证了关键字的有序。

```

x.keys.append((None, None)) # 在 keys 的末尾添加一个占位符
while i >= 0 and k < x.keys[i][0]:
    x.keys[i + 1] = x.keys[i] # 将 keys 向右移动以腾出空间
    i -= 1
x.keys[i + 1] = (k, data) # 将 (k, data) 插入到正确的位置

```

2、插入到非叶子节点:

找到适合插入 `(k, data)` 的子节点。

通过遍历 `x.keys` 找到正确的子节点索引 `i`, 在这里 `k` 应该被插入。检查索引 `i` 处的子节点是否已满 (`len(x.children[i].keys) == 2 * self.t - 1`)。如果已满, 使用 `self._split_child(x, i)` 分裂子节点。分裂后, 根据 `k` 的值决定是插入原始子节点还是新分裂的子节点。

确定了正确的子节点 (`x.children[i]`) (直接或分裂后) 后, 在该子节点上递归调用 `insert_to_buffer(k, data)`。这样持续插入, 直到到达一个叶子节点, 在叶子节点上进行实际的插入操作。

```

# 检查索引 i 处的子节点是否需要分裂
if len(x.children[i].keys) == 2 * self.t - 1:
    self._split_child(x, i) # 如果子节点已满, 则分裂它
    if k > x.keys[i][0]:
        i += 1

# 递归地将 (k, data) 插入到适当的子节点中
x.children[i].insert_to_buffer(k, data) # insert_to_buffer 处理在子节点中的插入

```

(5) `_split_child(self, x, i)`: 将节点 `x` 的第 `i` 个子节点分裂成两个节点。  
详细解释见下方注释代码:

```
def _split_child(self, x, i):
    # 获取 B 树的最小度数，用于新根节点的生成
    t = self.t
    # 节点 y 为父节点 x 的第 i 个子节点
    y = x.children[i]
    # 创建一个新节点 z，与 y 具有相同的叶子属性
    z = BeTreeNode(t, y.leaf)
    # 将节点 z 插入到父节点 x 的子节点列表中，位置在 x 之后
    x.children.insert(i + 1, z)
    # 将 y 的中间键（索引 t-1）插入到父节点 x 的键列表中的位置 i
    x.keys.insert(i, y.keys[t - 1])
    # 将 y 的键分割，将后半部分赋给 z
    z.keys = y.keys[t:(2 * t - 1)]
    # 将 y 的键列表缩减为前半部分
    y.keys = y.keys[0:(t - 1)]
    # 如果 y 不是叶子节点，则调整 z 的子节点列表为 y 的后半部分子节点
    if not y.leaf:
        z.children = y.children[t:(2 * t)]
        # 缩减 y 的子节点列表为前半部分子节点
        y.children = y.children[0:t]
```

## 2、BeTree 类

变量:

- (1) `root`: 树的根节点。
- (2) `t`: 最小度数，也就是结点的关键字的最少个数。

功能函数:

- (1) `__init__(self, t)`: 初始化树，设置根节点和最小度数。
- (2) `insert(self, k, data)`: 在树的根节点中插入新的键值对(`k`, `data`)。如果根节点已满，则分裂根节点并增加树的高度。对根进行分裂是增加树高度的唯一途径。根节点已满的情况下，需要构造新的根节点，并将原来的根节点设为新的根节点的 `child` 结点，并对原根结点也就是新的根节点的孩子节点进行分裂（调用 `_split_child` 函数），最后将新的数据插入到根节点中。

以下代码为根节点已满的情况

```
s = BeTreeNode(self.t, False)
s.children.insert(0, root)
# 分裂根节点，对根进行分裂是增加树高度的唯一途径
```



```
s._split_child(s, 0)
s._insert_non_full(s, k, data)
self.root = s
```

(3) `range_query(self, start_key, end_key)`: 在树中执行范围查询，返回键在 `start_key` 和 `end_key` 之间的值。

(5) `_range_query(self, x, start_key, end_key)`: 从节点 `x` 开始递归执行范围查  
首先调用 `flush()` 函数，确保缓冲区中的数据已经被处理。

`result = []`: 初始化一个空列表，用于存储查询结果。

第一个 `while` 循环 (`while i < len(x.keys) and x.keys[i][0] < start_key`) 遍历 `x.keys` 直到找到第一个大于等于 `start_key` 的键，将 `i` 设置为范围查询的起始位置。

第二个 `while` 循环 (`while i < len(x.keys) and x.keys[i][0] <= end_key`) 从起始位置 `i` 开始遍历 `x.keys`，直到找到大于 `end_key` 的键为止。在此循环中，`result.append(x.keys[i][1])`: 将与键关联的数据 (`x.keys[i][1]`) 添加到 `result` 中。

如果 `x` 不是叶子节点 (`not x.leaf`)，则递归调用 `_range_query` 在相应的子节点 `x.children[i]` 上执行范围查询，将结果合并到 `result` 中。

退出第二个 `while` 循环后，如果 `x` 还有子节点 (`i < len(x.children)`)，那么在最后一个子节点上递归调用 `_range_query`，将其结果合并到 `result` 中。

```
def _range_query(self, x, start_key, end_key):
    x.flush() # 确保查询之前将缓冲区数据处理完毕
    result = []
    i = 0
    # 定位范围查询的起始位置
    while i < len(x.keys) and x.keys[i][0] < start_key:
        i += 1
    # 处理位于范围 [start_key, end_key] 内的键
    while i < len(x.keys) and x.keys[i][0] <= end_key:
        result.append(x.keys[i][1]) # 将与键关联的数据添加到结果中
        if not x.leaf:
            # 递归地在子节点上执行范围查询
            result += self._range_query(x.children[i], start_key, end_key)
        i += 1

    # 如果 x 不是叶子节点，在最后一个子节点上执行范围查询
    if not x.leaf and i < len(x.children):
        result += self._range_query(x.children[i], start_key, end_key)

    return result
```

### 3、BookBeTreeIndex 类

**变量：**

- (1) `index_by_year`: 按出版年份索引的 B+ 树。
- (2) `index_by_reviews`: 按评论数量索引的 B+ 树。

**功能函数：**

- (1) `__init__(self, t)`: 初始化两个 B+ 树索引，分别按出版年份和评论数量索引。
- (2) `insert_books(self)`: 将 `collection` 集合中的书籍信息按照（“书名”，MongoDb 中书籍信息）的键值对插入到 B+ 树索引中。
- (3) `query_books(self, start_year, end_year, min_reviews)`: 先按出版年份范围查询出满足条件的书籍，存入 `result` 中。然后在第一次范围查询的结果 `result` 中再按评论数量进行 B+ 树索引，并进行第二次查询。

**重点代码介绍：**

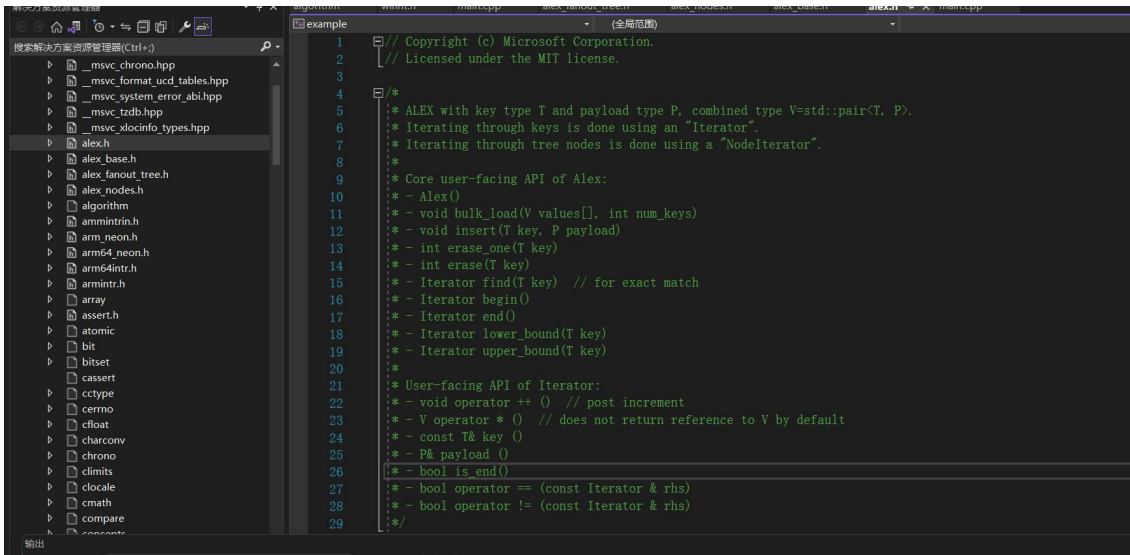
```
def query_books(self, start_year, end_year, min_reviews):
    # 首先根据出版时间进行范围查询
    result_by_year = self.index_by_year.range_query(start_year, end_year)
    # 然后在结果中根据评论数量进行二次范围查询
    for book in result_by_year:
        review_count = book['评论数量']
        self.index_by_reviews.insert(review_count, book)
    result = self.index_by_reviews.range_query(min_reviews, float('inf'))
    return result
```

### 3.3.2 ALEX 索引的实现

本文不仅利用 python 实现了 ALEX 学习索引，而且调用了 ALEX 索引原论文的代码 API 实现范围查询书籍的功能，通过阅读原论文源代码，对 ALEX 学习索引的结构有了很深刻的了解与掌握。

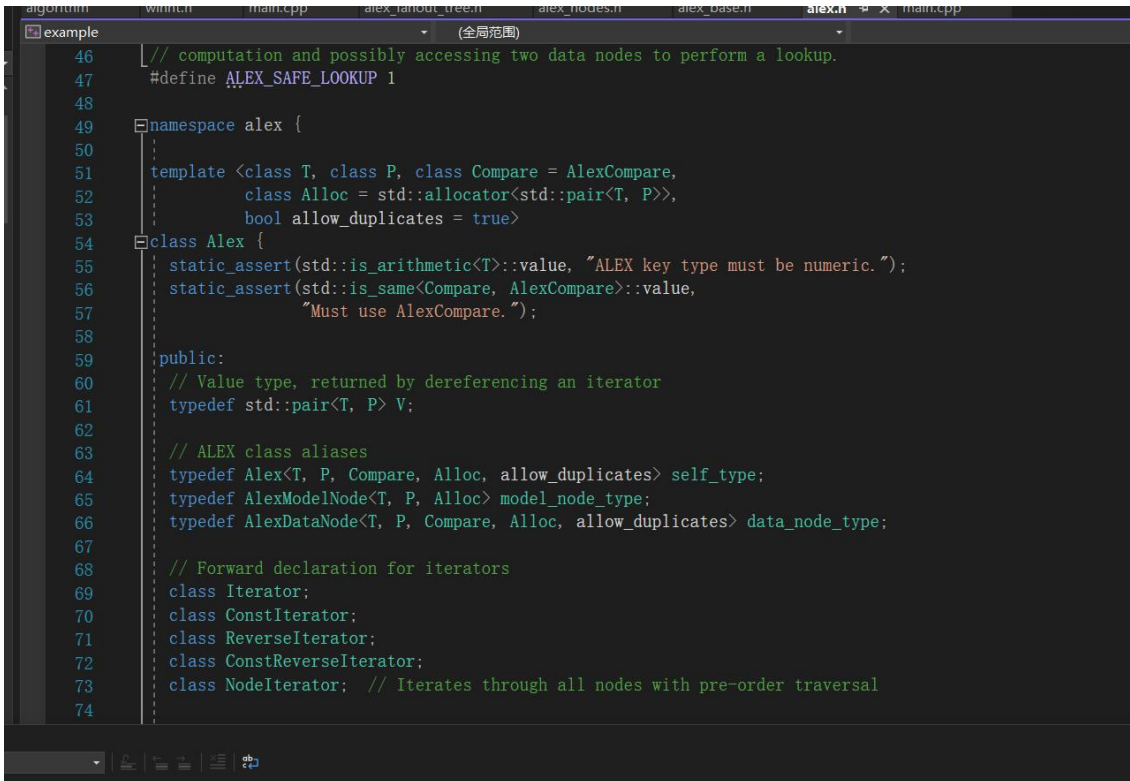
#### (1) 使用 C++ 的 api 实现

由于一开始不了解 ALEX 索引的结构，于是我上网查阅了大量资料，了解到 ALEX 学习索引的基本结构以及原理。在阅读 ALEX 索引的开山之作（论文）：ALEX: An Updatable Adaptive Learned Index 后，我复现了索引的 C++ 代码，并利用 API 实现了与以上查询一直的范围查询。以下为论文源代码部分 API 函数概述截图。实现效果可以见下方具体模块。



```
1 // Copyright (c) Microsoft Corporation.
2 // Licensed under the MIT license.
3
4 /*
5  * ALEX with key type T and payload type P, combined type V=std::pair<T, P>.
6  * Iterating through keys is done using an "Iterator".
7  * Iterating through tree nodes is done using a "NodeIterator".
8  *
9  * Core user-facing API of Alex:
10  * - Alex()
11  * - void bulk_load(V values[], int num_keys)
12  * - void insert(T key, P payload)
13  * - int erase_one(T key)
14  * - int erase(T key)
15  * - Iterator find(T key) // for exact match
16  * - Iterator begin()
17  * - Iterator end()
18  * - Iterator lower_bound(T key)
19  * - Iterator upper_bound(T key)
20  *
21  * User-facing API of Iterator:
22  * - void operator ++ () // post increment
23  * - V operator * () // does not return reference to V by default
24  * - const T& key ()
25  * - P& payload ()
26  * - bool is_end()
27  * - bool operator == (const Iterator & rhs)
28  * - bool operator != (const Iterator & rhs)
29  */
```

图 1 ALEX 论文源代码



```
46 // computation and possibly accessing two data nodes to perform a lookup.
47 #define ALEX_SAFE_LOOKUP 1
48
49 namespace alex {
50
51 template <class T, class P, class Compare = AlexCompare,
52          class Alloc = std::allocator<std::pair<T, P>>,
53          bool allow_duplicates = true>
54 class Alex {
55     static_assert(std::is_arithmetic<T>::value, "ALEX key type must be numeric.");
56     static_assert(std::is_same<Compare, AlexCompare>::value,
57                   "Must use AlexCompare.");
58
59 public:
60     // Value type, returned by dereferencing an iterator
61     typedef std::pair<T, P> V;
62
63     // ALEX class aliases
64     typedef Alex<T, P, Compare, Alloc, allow_duplicates> self_type;
65     typedef AlexModelNode<T, P, Alloc> model_node_type;
66     typedef AlexDataNode<T, P, Compare, Alloc, allow_duplicates> data_node_type;
67
68     // Forward declaration for iterators
69     class Iterator;
70     class ConstIterator;
71     class ReverseIterator;
72     class ConstReverseIterator;
73     class NodeIterator; // Iterates through all nodes with pre-order traversal
74 }
```

图 2 ALEX 论文源代码

```
algorithm  winnt.h  main.cpp  alex_fanout_tree.h  alex_nodes.h  alex_base.h  alex.h  main.cpp
example  (全局范围)
2989  }
2990
2991  AlexNode<T, P>* current() const { return cur_node_; }
2992
2993  AlexNode<T, P>* next() {
2994  if (node_stack_.empty()) {
2995      cur_node_ = nullptr;
2996      return nullptr;
2997  }
2998
2999      cur_node_ = node_stack_.top();
3000      node_stack_.pop();
3001
3002      if (!cur_node->is_leaf_) {
3003          auto node = static_cast<model_node_type*>(cur_node_);
3004          node_stack_.push(node->children_[node->num_children_ - 1]);
3005          for (int i = node->num_children_ - 2; i >= 0; i--) {
3006              if (node->children_[i] != node->children_[i + 1]) {
3007                  node_stack_.push(node->children_[i]);
3008              }
3009          }
3010      }
3011
3012      return cur_node_;
3013  }
3014
3015  bool is_end() const { return cur_node_ == nullptr; }
3016  };
3017  };
```

图 3 ALEX 论文源代码

## (2) python 实现

### 1. GappedArray 类

**类变量:**

**size:** 数组的大小。

**array:** 实际存储数据的数组，初始化为 None。

**gaps:** 表示数组中每个位置是否为间隙的布尔数组，True 表示为该位置为空,False 为非空。

**count:** 记录当前数组中已插入的元素数量。

**类函数:**

**\_\_init\_\_(self, size):**

初始化数组大小、数组、间隙和计数。

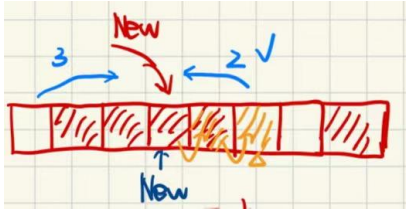
**insert(self, index, value):**

在 index 处插入值，若 index 处为空，则直接插入；如果该位置已被占用，则调用 create\_gap 方法，通过整体移动元素在 Index 处创造出空位置，然后将 value 值插入到 index 处。

**create\_gap(self, index):**

首先通过遍历数组，找到 index 两个方向中距离 index 最近的空值，记录下 n

ear\_gap\_index 的位置。然后判断这个位置位于 index 的左边还是右边，再统一将数据往空值的方向移动一个位置，从而在 index 处创建出了一个间隙（空值），此时将 index 处的 gaps 标记为 True。



**search(self, value):**

遍历数组中的值，返回其 index 没找到则返回-1。

**get\_items(self):**

返回数组中所有非 None 的元素。

**is\_full(self):**

检查数组是否已满。

**\_\_len\_\_(self):**

返回数组中已插入的元素数量。

## 2. InternalNode 类

**类变量:**

**order:** 内部节点的阶数。

**keys[]:** 存储键的列表，为普通列表。

**Children[]:** 存储子节点的列表，为普通列表。

**model:** 用于预测位置的线性回归模型，用 LinearRegression() 进行初始化。

**类函数:**

**\_\_init\_\_(self, order):**

初始化节点的阶数、键、子节点和模型。

**def insert(self, key, child):** #内部结点的插入操作

self.keys.append(key)

self.children.append(child)

self.keys.sort() #插入时对关键字进行排序

self.update\_model()

**update\_model(self):**

只有当关键字的个数大于 1 时才进行模型训练，否则没有意义。

使用当前结点的关键字数组中的 Key 值 x 以及对应位置 y 更新线性回归模型

self.model.fit(X, y)。

**predict\_position(self, key):**

预测键的插入位置，返回 `int(self.model.predict(np.array([[key]]))[0])`。

**split(self):**

分裂内部节点，首先选出结点中间位置的键值设为中间键 `mid_index`。然后创建两个内部节点 `left` 与 `right`，将原来的内部结点的第一个键值到 `mid_index` 个键值和孩子节点 `children` 传递给 `left` 结点；`mid_index` 到最后一个键值和孩子节点传递给 `right` 节点。最后分别对 `left` 与 `right` 结点进行模型训练，并返回中间键和两个新节点。

### 3. LeafNode 类

**类变量:**

**order:** 叶子节点的阶数。

**keys[]:** 为 `GappedArray` 类型，用于存储叶子节点的关键字。

**data[]:** 为 `GappedArray` 类型，用于存储叶子节点的数据。

**next:** 指向下一个叶子节点的指针。

**model:** 用于预测位置的线性回归模型。

**类函数:**

**\_\_init\_\_(self, order):**

初始化节点的阶数、键、数据、`next` 指针和线性回归模型。

**insert(self, key, value):**

向叶子结点中插入关键字和数据，并更新模型。

优先通过线性回归模型预测 `key` 值所在的 `index`，然后判断 `index` 是否合法，若 `index` 小于 0 或者超出数组范围，则对 `index` 置 0 或者缩小为一半。

当 `index` 的范围合法时，则再次判断 `index` 处是否为空，若 `index` 处为空，则直接插入，否则进行指数搜索找到新的 `index` 位置进行插入。

指数搜索过程的代码如下：

```
else:                                #进行指数搜索
```

```
exp = 1
```

```
while index + exp < self.order and self.keys.array[index + exp] is not None:
```

```
    exp *= 2
```

```
i = index + exp
```

```
if i >= self.order:
```

```
    i = self.order - 1
```

```
while i > index and self.keys.array[i] is not None:
```

```
    i -= 1
```

```
self.keys.insert(i, key)
```

```
self.data.insert(i, value)
```

### **update\_model(self):**

与内部结点的模型训练不同，这里的数组类型为 `GappedArray`，要首先筛选出不为空的元素，得到他们的 `key` 值与位置 `index`，然后使用非空元素的 `key` 值组成 `x`，他们的位置 `index` 组成 `y`，利用 `x` 与 `y` 对当前节点更新线性回归模型。

注意只有的当不为空的 `key` 值个数大于 1 时才进行模型训练。

### **predict\_position(self, key):**

与内部节点 `predict_position` 方法一样，预测关键值 `key` 的插入位置。

### **split(self):**

由 ALEX 树索引出进行数据插入时，会对结点的 `key` 值个数做检查，故这里叶子结点的 `insert` 函数没有考虑节点关键字数组满的情况，`split` 函数在 ALEX 类中调用。

叶子结点的分裂与内部结点基本一致，有以下几点不同：

(1) 创建新结点时为 `LeafNode` 类型

(2) 创建出 `left` 与 `right` 结点后，对 `next` 指针进行赋值：`left.next = right`, `right.next = self.next`。

最后返回中间键和两个新节点并对新节点训练模型。

## **4. Alex 类**

### **类变量:**

**order:** 节点的阶数。

**root:** 树的根节点，为内部节点类型。

### **类函数:**

**`__init__(self, order):`** 初始化树的阶数和根节点。

**`insert(self, key, value):`** 插入关键字和数据，处理节点的分裂。

`node = self.root:` 初始化当前节点为根节点。

`parent_stack = []:` 初始化一个栈来存储路径上的父节点及插入位置。

遍历树找到叶节点：

`while isinstance(node, InternalNode):` 如果当前节点是内部节点，则进入循环。

`pos = node.predict_position(key):` 预测要插入的子节点位置。

根据键值的大小调整插入位置，确保键值在正确的范围内。

将当前节点及其插入位置存入栈中：`parent_stack.append((node, pos))`。

更新当前节点为其子节点：`node = node.children[pos]`。

在叶节点插入键值对：

`node.insert(key, value)`：在叶节点插入键值对。

处理叶节点满的情况：

`if node.keys.array.count(None) == 0`：如果叶节点已满，则需要分裂。

`mid_key, left, right = node.split()`：分裂叶节点。

如果存在父节点，处理父节点：

`parent_node, pos = parent_stack.pop()`：获取父节点及其插入位置。

在父节点插入中间键，并更新其子节点。

如果父节点也满，则继续处理祖父节点，直到根节点。

如果不存在父节点（即叶节点为根节点），则创建新根节点并更新根节点。

带有详细注释的代码如下：

```
def insert(self, key, value):
    node = self.root
    parent_stack = [] # 存储路径上的父节点和插入位置
    # 遍历树，直到找到叶节点
    while isinstance(node, InternalNode):
        pos = node.predict_position(key) # 预测插入位置
        if pos < 0:
            pos = 0
        if pos is None or pos >= len(node.keys):
            pos = len(node.keys) - 1
        while pos < len(node.keys) and key > node.keys[pos]:
            pos += 1
        while pos > 0 and key < node.keys[pos - 1]:
            pos -= 1
        parent_stack.append((node, pos)) # 记录当前节点及其插入位置
        node = node.children[pos]

    node.insert(key, value)

# 如果叶节点已满，进行分裂
if node.keys.array.count(None) == 0:
    mid_key, left, right = node.split() # 分裂叶节点
    if parent_stack:
        parent_node, pos = parent_stack.pop() # 获取父节点及其插入位置
        parent_node.keys.insert(pos, mid_key) # 在父节点插入中间键
```



```

parent_node.children[pos] = left # 更新父节点的子节点
parent_node.children.insert(pos + 1, right)
while len(parent_node.keys) > self.order - 1: # 如果父节点也满
    if parent_stack:
        grandparent_node, parent_pos = parent_stack.pop()
        mid_key, left, right = parent_node.split()
        grandparent_node.keys.insert(parent_pos, mid_key)
        grandparent_node.children[parent_pos] = left
        grandparent_node.children.insert(parent_pos + 1, right)
        parent_node = grandparent_node # 继续处理祖父节点
    else: #更新根节点
        mid_key, left, right = parent_node.split() # 分裂父节点
        new_root = InternalNode(self.order) # 创建新的根节点
        new_root.keys = [mid_key] # 设置新根节点的键
        new_root.children = [left, right] # 设置新根节点的子节点
        self.root = new_root # 更新根节点
        break
else: #更新根节点
    new_root = InternalNode(self.order) # 创建新的根节点
    new_root.keys = [mid_key] # 设置新根节点的键
    new_root.children = [left, right] # 设置新根节点的子节点
    self.root = new_root # 更新根节点

```

点

**search(self, key):** 搜索关键字 key，返回对应的数据。

**range\_query(self, start\_key, end\_key):**

范围查询，返回指定范围内的数据。

首先通过线性回归模型预测 start\_key 的位置，找到叶子结点，再利用叶子结点的 next 指针寻找下一个叶子结点，同时预测 end\_key 的位置，在最后一个结点找到后停止继续寻找。将满足范围的数据插入到 result 数组中，返回结果。

## 5. BookIndex 类

**类变量:**

**index\_by\_year:** 按年份索引的数据结构，为 Alex 类型。

**类函数:**

**\_\_init\_\_(self):** 初始化按年份索引的数据结构。

**insert\_books(self):** 从 MongoDB 中读取书籍数据并插入索引。

**query\_books(self, start\_year, end\_year):** 按年份范围查询书籍数据。

## 4. 算法思想及流程

### 4.1 题目一

将原始数据清理空值，并用利用正则表达式提取合乎规范的出版年后，将提取数据中“书名”等列到 csv 文件中，再建立 MongoDB 的数据库连接，读取清理后数据文件，按照 df 数据类型保存，再将其逐一转换为字典类型的数据存入到 MongoDB 数据库中。

### 4.2 题目二

**bf 过滤器:**

提供两种查询，若为简单判断数据是否存在，则首先初始化一个布隆过滤器 ff，然后将所有书名全部存入过滤器当中，调用 `ff.contains(book_name)` 判断该书名是否在过滤器中。

若为查询书籍详细信息，则首先利用 Deal 类中的 `convert_books_to_list()` 将所有 collection 中的信息经过处理后存入 `book_info_collection`。然后再利用 `query_book` 函数查询该书名的准确信息。

**cf 过滤器:**

(1) 添加元素 (add 方法):

对于要添加的元素，首先计算其指纹 (fingerprint)，这里使用 SHA-1 算法取前四位作为指纹。然后使用 `hash1` 计算元素的初始位置 `i1`，接着使用 `hash2` 计算指纹的位置 `i2`，并且将其与 `i1` 异或，得到第二个可能的位置。如果 `i1` 对应的桶有空间，直接将指纹添加到该桶中。如果 `i1` 桶已满，而 `i2` 桶有空间，则将指纹添加到 `i2` 桶中。如果两个桶都已满，则执行桶溢出处理：

1、选择桶内元素较少的桶作为起始桶。

2、依次将起始桶中的元素挤出，将新元素插入，并更新挤出的元素作为新的插入元素，重新计算新的 `i2` 位置，直到成功插入或达到最大允许重新插入次数 `max_kicks`。

(2) 查询元素 (contains 方法):

对于要查询的元素，同样先计算其指纹。使用 `hash1` 计算元素的位置 `i1`。使用 `hash2` 计算指纹的位置 `i2`，并且将其与 `i1` 异或，得到第二个可能的位置。在 `i1` 桶或 `i2` 桶中查找是否存在该元素的指纹。

**Deal 类:**

(1) 存储书籍信息 (store\_book 方法): 在存储书籍信息之前，先通过布隆过滤器判断该书籍标题是否已存在，如果存在则更新书籍的出版社、出版时间、评分和

评论数量；如果不存在，则将书籍信息插入到 MongoDB 的集合中，并将书名添加到布隆过滤器中。

(2) 查询书籍信息 (query\_book 方法): 根据书名使用布隆过滤器判断是否可能存在于数据库中，如果可能存在，则从数据库中查询具体信息；否则返回空。

(3) 显示所有书籍信息 (print\_all\_books 方法): 打印出数据库中所有的书籍信息。

Deal 类中向过滤器以及 book\_info 中添加图书信息流程图:

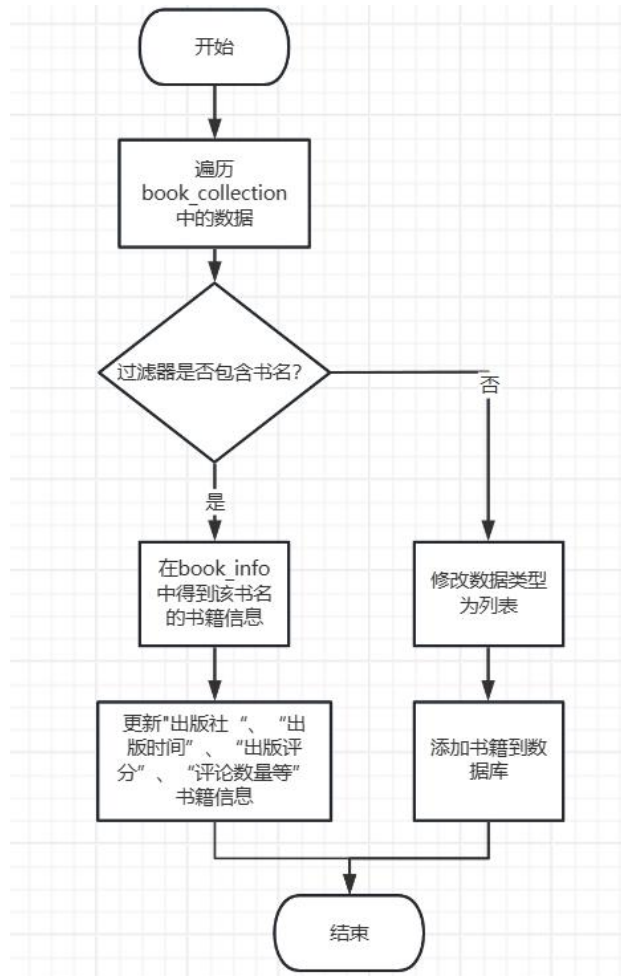


图 4 Deal 类中 store\_book 流程图

总体流程图:

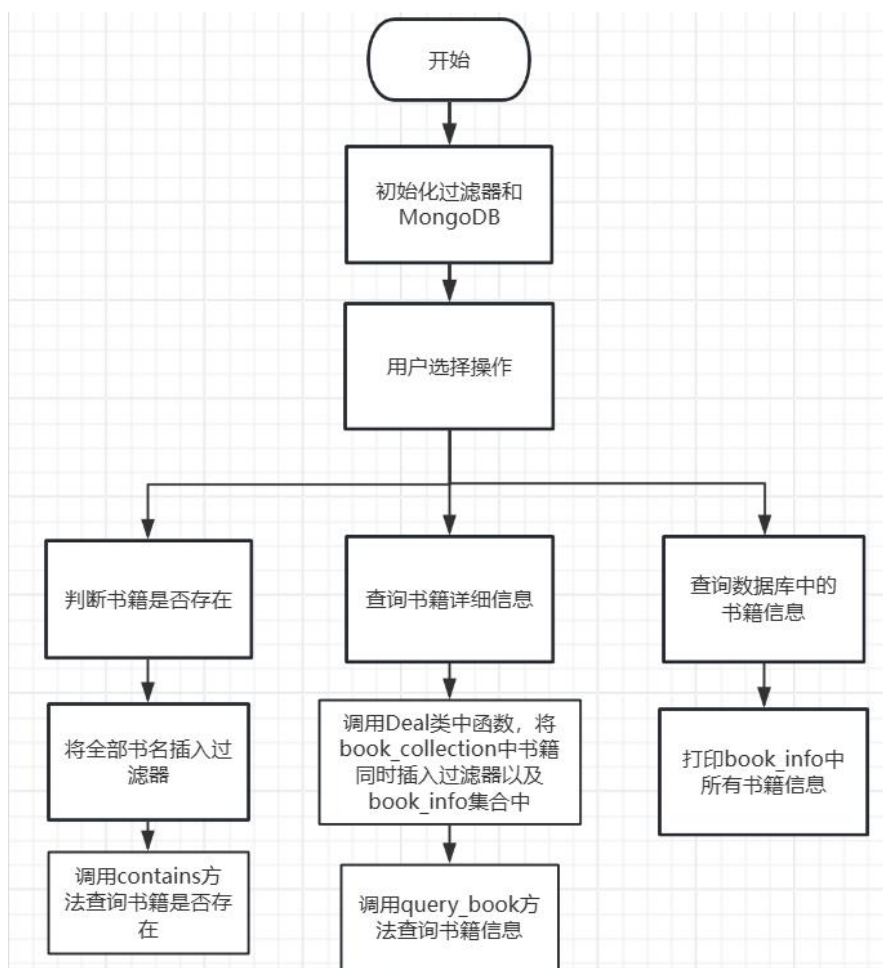


图 5 总体过滤数据流程图

### 4.3 题目三

使用 B $\epsilon$  树 (Be Tree) 来优化书籍数据的存储和查询操作。B $\epsilon$  树通过在每个节点中引入缓冲区来减少 I/O 操作，从而提高写入性能和查询性能。

#### BeTreeNode 类:

每个节点包含键值对 (keys)、子节点 (children) 和缓冲区 (buffer)。数据首先进入到节点的缓冲区，当缓冲区已满时，清空缓冲区，并将新加入的数据存储到下层节点，存储给下层节点时，同样优先插入到缓冲区中。数据插入到缓冲区时，数据先插入到节点的缓冲区中，缓冲区满时执行 flush() 操作。flush 操作中，将缓冲区中的数据利用非满插入批量插入到键值对列表中或传递给子节点进行递归插入。非满节点插入数据时，在非满节点中插入数据，必要时进行节点分裂。

#### BeTree 类:

包含根节点和最小度数。进行插入操作时，若根节点没有满，那么直接插入到根节点中；根节点满时，进行分裂，然后递归插入数据到适当的子节点中。实现范

围查询时，在树中进行范围查询，确保缓冲区的数据在查询前被处理。

### BookBeTreeIndex 类:

创建两个 B $\epsilon$  树索引，分别按出版年份和评论数量进行索引。

插入书籍数据时，从 MongoDB 中读取书籍信息，并根据出版年份插入到第一个 B $\epsilon$  树中。

查询书籍时，先按出版年份进行范围查询，再在结果中按评论数量进行二次范围查询，最终返回符合条件的书籍列表。

进行范围查询时的流程图:

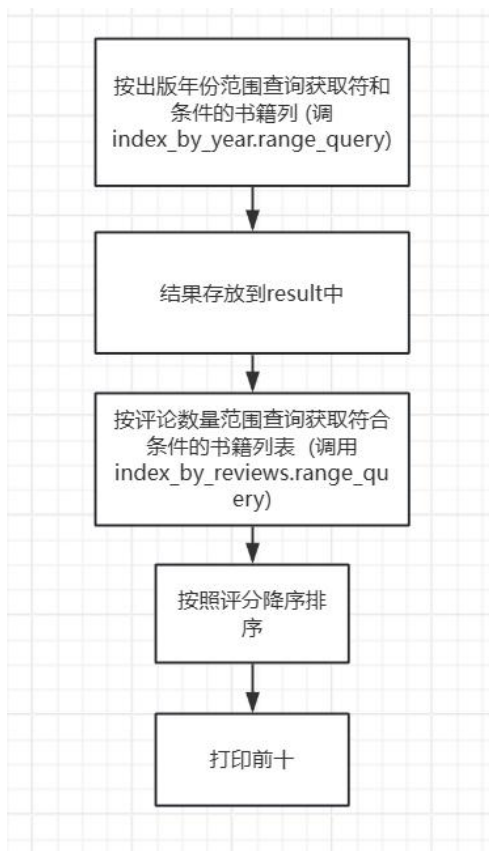


图 6 范围查询流程图

### 学习索引算法思想

ALEX 树是基于 B+树和 RMI 结构改进，每个结点上有一个线性回归模型和一部分连续存储空间（数组）非叶节点（内部节点）的数组存储叶节点的指针。叶节点（数据节点）有两个数组，一个用来存储关键字，另一个保存关键字对应的其他信息。其中关键字数据利用 Gapped Array (GA) 要保留一些间隔，用来提高数据动态更新时的性能。结点上的模型用来拟合数组中的数据分布，从而预测输入的 Key 的位置 每个结点能够在数据更新时根据情况动态的分裂和合并。

AELX 的查询操作：根节点开始，我们迭代地使用该模型来“计算”指针数

组中的一个位置，并且我们跟随指针到下一级的子节点，直到我们到达一个数据节点 通过构造，内部节点的模型具有完美的准确性，不存在误差。我们使用数据节点中的模型来预测搜索关键字在数组中的位置，如果预测有误则从预测位置开始进行指数搜索。

插入操作：对于数据插入，找到插入位置的方式同查询一样 当被插入的数据结点的关键字数组非满时，为了找到新元素的插入位置，使用数据节点中的模型来预测插入位置。如果预测的位置不正确(在此位置插入会破坏关键字的大小排序)，则进行指数搜索以找到正确的插入位置。如果预测位置是一个间隙，则进行插入。 如果预测位置不是间隙，则将附近元素向最近的间隙方向移动一个位置，在插入位置形成一个间隙。然后，我们将元素插入到新创建的间隙中。插入关键字后将关键字对应的其他数据放入工作负载空间中。当被插入的数据结点的空间已满时，需要对结点进行扩展或者分裂两种操作。每次新创建出一个结点后，都对该结点进行模型训练。

叶子结点的插入流程图：

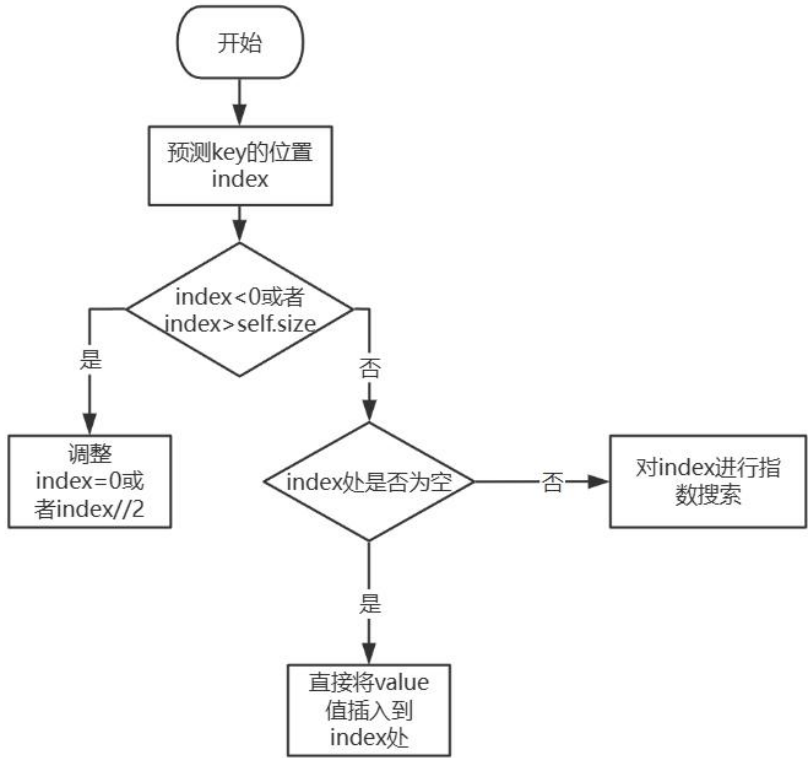


图 7 叶子结点的插入流程图

## ALEX 树索引的插入流程图：

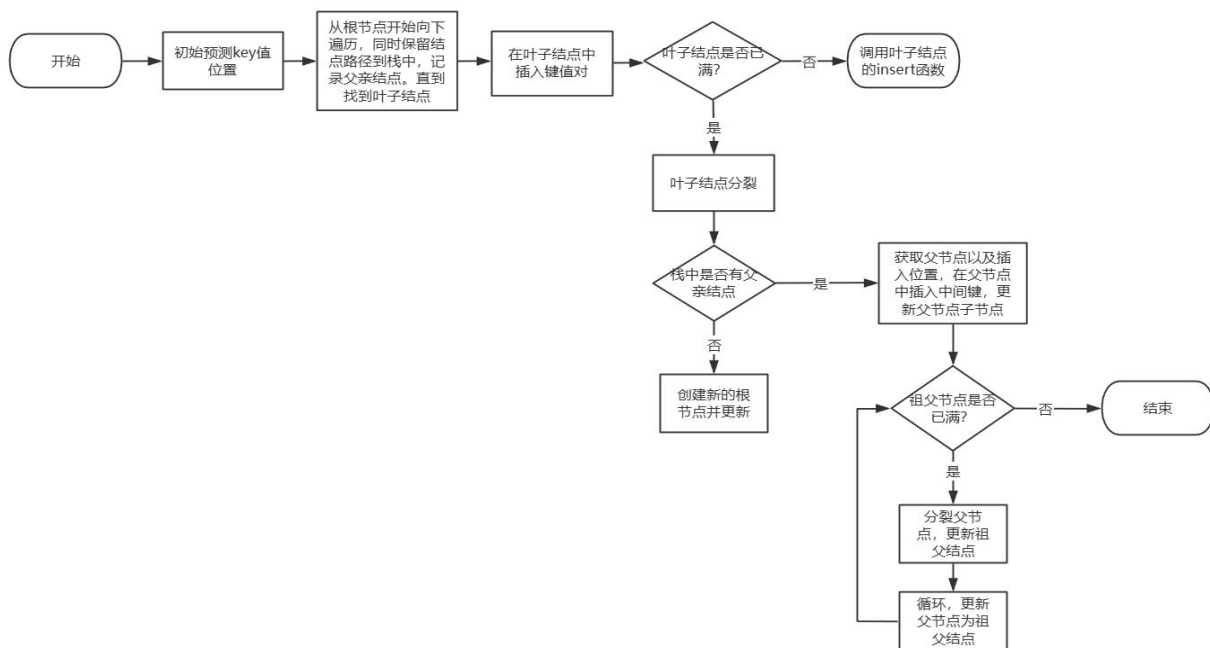


图 8 ALEX 树索引的插入流程图

## 5. 程序代码

整体代码结构：

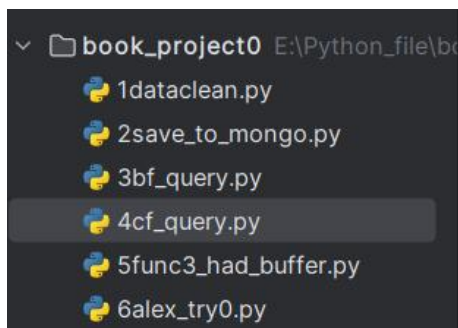


图 9 整体代码结构

1dataclean.py 与 2save\_to\_mongo.py 对应第一题，分别完成了数据清洗以及将数据存入数据库的功能。

3bf\_query.py 与 4cf\_query.py 对应第二题，分别实现按 bf 过滤器实现数据更新以及按 cf 过滤器实现数据更新。

5func3\_had\_buffer.py 与 6alex\_try0.py 对应第三题，分别实现按 B $\epsilon$ -Tree 索引与 AL EX 学习索引实现满足查询条件的书籍信息查询。

## 1dataclean.py:

```
import pandas as pd
import numpy as np
import re

# 读取数据
df = pd.read_csv("C:\\学习资料二\\大数据存储与管理\\课设\\book_douban.csv", encoding='ANSI')
print(df.columns)

# 删除包含 None、0 或者为空的行
df.replace(['None', 0, ""], np.nan, inplace=True)
df.dropna(inplace=True)

# 提取出版年份
def extract_max_year(date_str):
    if isinstance(date_str, str):
        # 使用正则表达式匹配所有四位年份
        year_matches = re.findall(r'\b(?:19|20)\d{2}\b', date_str)
        if year_matches:
            # 将匹配的年份字符串转换为整数列表
            years = [int(year) for year in year_matches]
            # 返回最大年份
            return max(years)
    return np.nan

df['出版时间'] = df['出版时间'].apply(extract_max_year)

# 删除无法提取年份的行
df.dropna(subset=['出版时间'], inplace=True)

# 将数据类型转换为整数
```



```

df['出版时间'] = df['出版时间'].astype(int)

# 选择需要的列
df = df[['Unnamed: 0', '书名', '作者', '出版社', '出版时间', '评分', '评论数量']]

# 将清洗后的数据保存到新文件
df.to_csv("C:\\学习资料二\\大数据存储与管理\\课设\\cleandata.csv", encoding='ANSI', index=False)

print("数据清洗完成并已保存到 cleandata.csv 文件中")

```

## 2save\_to\_mongo.py:

```

import pymongo
import pandas as pd
from pymongo import MongoClient
from pprint import pprint
# 读取清洗后的 CSV 文件
df = pd.read_csv("C:\\学习资料二\\大数据存储与管理\\课设\\cleandata.csv", encoding='ANSI')

# 选择需要的列
df = df[['书名', '作者', '出版社', '出版时间', '评分', '评论数量']]

# 连接 MongoDB 数据库
client = MongoClient("mongodb://localhost:27017/") # 请根据实际情况修改连接字符串
db = client['book_database'] # 创建或连接数据库
collection = db['book_collection'] # 创建或连接集合

# 将 DataFrame 转换为字典并插入到 MongoDB
data = df.to_dict(orient='records')
collection.delete_many({})
collection.insert_many(data)

print("数据已成功插入到 MongoDB 数据库中")
# 打印集合中的数据
print("\n 集合中的数据: ")
for document in collection.find():
    pprint(document)

```

### 3bf\_query.py:

```
import sys

from bitarray import bitarray
import hashlib
from pymongo import MongoClient
from pprint import pprint

class MyBloomFilter:
    DEFAULT_SIZE = 2 << 24
    SEEDS = [3, 13, 46, 71, 91, 134]

    def __init__(self):
        self.bits = bitarray(self.DEFAULT_SIZE)
        self.bits.setall(0)
        self.func = [self.SimpleHash(self.DEFAULT_SIZE, seed) for seed in self.SEEDS]

    def add(self, value):
        for f in self.func:
            self.bits[f.hash(value)] = 1

    def contains(self, value):
        return all(self.bits[f.hash(value)] for f in self.func)

    class SimpleHash:
        def __init__(self, cap, seed):
            self.cap = cap
            self.seed = seed

        def hash(self, value):
            result = int(hashlib.md5(value.encode('utf-8')).hexdigest(), 16)
            return (self.seed * result) % self.cap

class Deal:
```

```

def __init__(self, filter):
    self.filter = filter
    self.mongo_client = MongoClient('mongodb://localhost:27017/')
    self.database = self.mongo_client['book_database']
    self.collection = self.database['book_collection']
    self.book_info_collection = self.database['book_info']

def convert_books_to_list(self):
    self.book_info_collection.delete_many({})
    for book in self.collection.find():
        self.store_book(book)

def store_book(self, book):
    book_title = book['书名']
    if self.filter.contains(book_title):
        existing_book = self.book_info_collection.find_one({'书名': book_title})

        if existing_book:
            # 更新出版社
            publishers = existing_book.get('出版社', [])
            publishers.append(book['出版社'])

            # 更新出版时间
            times = existing_book.get('出版时间', [])
            times.append(book['出版时间'])

            # 更新评分
            scores = existing_book.get('评分', [])
            scores.append(book['评分'])

            # 更新评论数量
            reviews = existing_book.get('评论数量', [])
            reviews.append(book['评论数量'])

            self.book_info_collection.update_one(
                {'书名': book_title},
                {'$set': {
                    '出版社': publishers,
                    '出版时间': times,
                    '评分': scores,
                    '评论数量': reviews
                }})
    else:
        self.book_info_collection.insert_one(
            {'书名': book_title, '出版社': book['出版社'], '出版时间': book['出版时间'],
            '评分': book['评分'], '评论数量': book['评论数量']})

```

```

        }}
    )
else:
    self.filter.add(book_title)
    book['出版社'] = [book['出版社']]
    book['出版时间'] = [book['出版时间']]
    book['评分'] = [book['评分']]
    book['评论数量'] = [book['评论数量']]
    if '_id' in book:
        del book['_id'] # 移除 _id 字段
    self.book_info_collection.insert_one(book)

def query_book(self, title):
    if self.filter.contains(title):
        return self.book_info_collection.find_one({'书名': title})
    else:
        return None

def print_all_books(self):
    print("\n 集合中的数据: ")
    for document in self.book_info_collection.find():
        pprint(document)
    # for book in self.book_info_collection.find():
    #     print(book)

# 示例用法
if __name__ == "__main__":
    # 初始化布隆过滤器
    filter = MyBloomFilter()
    deal = Deal(filter)

    # 交互界面
    while True:
        print("请输入要进行的操作：(1、简单判断书籍是否存在 2、查询书籍
        详细信息 3、显示 mongodb 中所有数据信息 0、退出)")
        choose = input()
        if choose == '1':
            print("请输入要查询的书名:")
            book_name = input()
            ff = MyBloomFilter()

```

```

        mongo_client = MongoClient('mongodb://localhost:27017/')
        database = mongo_client['book_database']
        collection = database['book_collection']
        for book in collection.find():
            ff.add(book['书名'])
        if ff.contains(book_name):
            print("yes, exists")
        else:
            print("does not exist")
    elif choose == '2':
        print("请输入要查询的书名:")
        book_name = input()
        deal.convert_books_to_list()
        print(deal.query_book(book_name))
    elif choose == '3':
        deal.print_all_books()
    elif choose == '0':
        sys.exit()

```

## 4cf\_query.py:

```

import sys
import hashlib
from pymongo import MongoClient
from pprint import pprint

class CuckooFilter:
    def __init__(self, size, bucket_size=4, max_kicks=500):
        self.size = size
        self.bucket_size = bucket_size #桶的大小为 4
        self.max_kicks = max_kicks
        self.buckets = [[] for _ in range(size)] #由 size 个空列表组成
        self.hash1 = self.get_hash_function()
        self.hash2 = self.get_hash_function()

    def get_hash_function(self):
        return lambda x: int(hashlib.md5(x.encode('utf-8')).hexdigest(), 16) % self.size
#用 hashlib 中的 md5 算法计算字符串 x 的哈希值，然后将而且对 self.size

```

## 取模

```
def fingerprint(self, item):
    return hashlib.shal(item.encode('utf-8')).hexdigest()[:4] #指纹取前四位

def add(self, item):
    fp = self.fingerprint(item)
    i1 = self.hash1(item)
    i2 = i1 ^ self.hash2(fp)

    if len(self.buckets[i1]) < self.bucket_size:
        self.buckets[i1].append(fp)
        return True

    if len(self.buckets[i2]) < self.bucket_size:
        self.buckets[i2].append(fp)
        return True

    i = i1 if len(self.buckets[i1]) < len(self.buckets[i2]) else i2
    for _ in range(self.max_kicks):
        evicted_fp = self.buckets[i].pop(0)
        self.buckets[i].append(fp)
        fp = evicted_fp
        i = i1 if i == i2 else i2
        i2 = i ^ self.hash2(fp)

    if len(self.buckets[i]) < self.bucket_size:
        self.buckets[i].append(fp)
        return True

    return False

def contains(self, item):
    fp = self.fingerprint(item)
    i1 = self.hash1(item)
    i2 = i1 ^ self.hash2(fp)
    return fp in self.buckets[i1] or fp in self.buckets[i2]

class Deal:
    def __init__(self, filter):
        self.filter = filter
```

```

self.mongo_client = MongoClient('mongodb://localhost:27017/')
self.database = self.mongo_client['book_database']
self.collection = self.database['book_collection']
self.book_info_collection = self.database['book_info']

def convert_books_to_list(self):
    self.book_info_collection.delete_many({})
    for book in self.collection.find():
        self.store_book(book)

def store_book(self, book):
    book_title = book['书名']
    if self.filter.contains(book_title):
        existing_book = self.book_info_collection.find_one({'书名': book_title})

        if existing_book:
            # 更新出版社
            publishers = existing_book.get('出版社', [])
            publishers.append(book['出版社'])

            # 更新出版时间
            times = existing_book.get('出版时间', [])
            times.append(book['出版时间'])

            # 更新评分
            scores = existing_book.get('评分', [])
            scores.append(book['评分'])

            # 更新评论数量
            reviews = existing_book.get('评论数量', [])
            reviews.append(book['评论数量'])

            self.book_info_collection.update_one(
                {'书名': book_title},
                {'$set': {
                    '出版社': publishers,
                    '出版时间': times,
                    '评分': scores,
                    '评论数量': reviews
                }}
            )
        else:
            self.book_info_collection.insert_one({
                '书名': book_title,
                '出版社': book['出版社'],
                '出版时间': book['出版时间'],
                '评分': book['评分'],
                '评论数量': book['评论数量']
            })
    else:
        self.book_info_collection.insert_one({
            '书名': book_title,
            '出版社': book['出版社'],
            '出版时间': book['出版时间'],
            '评分': book['评分'],
            '评论数量': book['评论数量']
        })

```

```

else:
    self.filter.add(book_title)
    book['出版社'] = [book['出版社']]
    book['出版时间'] = [book['出版时间']]
    book['评分'] = [book['评分']]
    book['评论数量'] = [book['评论数量']]
    if '_id' in book:
        del book['_id'] # 移除 _id 字段
    self.book_info_collection.insert_one(book)

def query_book(self, title):
    if self.filter.contains(title):
        return self.book_info_collection.find_one({'书名': title})
    else:
        return None

def print_all_books(self):
    print("\n 集合中的数据: ")
    for document in self.book_info_collection.find():
        pprint(document)

# 示例用法
if __name__ == "__main__":
    # 初始化布谷鸟过滤器
    filter = CuckooFilter(size=2 << 24)
    deal = Deal(filter)

    # 交互界面
    while True:
        print("请输入要进行的操作：(1、简单判断书籍是否存在 2、查询书籍
        详细信息 3、显示 mongodb 中所有数据信息 0、退出)")
        choose = input()
        if choose == '1':
            print("请输入要查询的书名:")
            book_name = input()
            filter = CuckooFilter(size=2 << 24)
            mongo_client = MongoClient('mongodb://localhost:27017/')
            database = mongo_client['book_database']
            collection = database['book_collection']
            for book in collection.find():
                filter.add(book['书名'])

```



```

        if filter.contains(book_name):
            print("yes, exists")
        else:
            print("does not exist")
    elif choose == '2':
        print("请输入要查询的书名:")
        book_name = input()
        deal.convert_books_to_list()
        print(deal.query_book(book_name))
    elif choose == '3':
        deal.print_all_books()
    elif choose == '0':
        sys.exit()

```

### 5func3\_had\_buffer.py:

```

from pymongo import MongoClient
from pprint import pprint

```

'''

B $\epsilon$ 树是将 B 树的每个节点空间分为两个部分，一部分按照原来的功能存储关键字和指向下层节点的指针；

另一部分空间用作缓冲区(buffer),即写入的数据优先缓存在当前节点的缓冲区中，只有当缓冲区已满，才能将缓冲区中的内容批量传递给下层节点。

下层节点也会先将数据存在缓冲区中。

'''

```

class BeTreeNode:

```

```

    def __init__(self, t, leaf=False):
        self.t = t # 最小度数 (Minimum degree)
        self.leaf = leaf # 是否为叶子节点
        self.keys = [] # 键值对
        self.children = [] # 子节点
        self.buffer = [] # 缓冲区，缓冲区的大小为 t，保证了不超过最小度数

```

```

def insert_to_buffer(self, k, data):
    self.buffer.append((k, data))
    if len(self.buffer) >= self.t: # 缓冲区满，执行 flush 操作
        self.flush()

def flush(self):
    if self.leaf: #为叶子结点，直接将缓冲区中的键值对插入到 keys 列表中。
        buffer = sorted(self.buffer, key=lambda x: x[0])
        for k, data in self.buffer:
            self._insert_non_full(self, k, data) #这个操作已经涉及了排序
    else: #将键值对传递给子节点
        for k, data in self.buffer:
            i = len(self.keys) - 1
            while i >= 0 and k < self.keys[i][0]:
                i -= 1
            i += 1
            if len(self.children[i].buffer) >= self.t:
                self.children[i].flush()
            self.children[i].buffer.append((k, data)) #这里是一个个的添加的，每次都会检查缓冲区是否已满
        self.buffer = []

def _insert_non_full(self, x, k, data):
    i = len(x.keys) - 1

    # 检查 x 是否为叶子节点
    if x.leaf:
        # 如果 x 是叶子节点，则将 (k, data) 插入到 x 中
        x.keys.append((None, None)) # 在 keys 的末尾添加一个占位符
        while i >= 0 and k < x.keys[i][0]:
            x.keys[i + 1] = x.keys[i] # 将 keys 向右移动以腾出空间
            i -= 1
        x.keys[i + 1] = (k, data) # 将 (k, data) 插入到正确的位置
    else:
        # 如果 x 不是叶子节点
        while i >= 0 and k < x.keys[i][0]:
            i -= 1
        i += 1

        # 检查索引 i 处的子节点是否需要分裂

```

```

        if len(x.children[i].keys) == 2 * self.t - 1:
            self._split_child(x, i) # 如果子节点已满，则分裂它
            if k > x.keys[i][0]:
                i += 1

        # 递归地将 (k, data) 插入到适当的子节点中
        x.children[i].insert_to_buffer(k, data) # insert_to_buffer 处理在子节
点中的插入

```

```

def _split_child(self, x, i):
    t = self.t
    y = x.children[i]
    z = BeTreeNode(t, y.leaf)
    x.children.insert(i + 1, z)
    x.keys.insert(i, y.keys[t - 1])
    z.keys = y.keys[t:(2 * t - 1)]
    y.keys = y.keys[0:(t - 1)]
    if not y.leaf:
        z.children = y.children[t:(2 * t)]
        y.children = y.children[0:t]

```

```

class BeTree:
    def __init__(self, t):
        self.root = BeTreeNode(t, True)
        self.t = t # 最小度数

    def insert(self, k, data):
        root = self.root
        if len(root.keys) == 2 * self.t - 1: # 根节点已满
            s = BeTreeNode(self.t, False)
            s.children.insert(0, root)
            # 分裂根节点，对根进行分裂是增加树高度的唯一途径
            s._split_child(s, 0)
            s._insert_non_full(s, k, data)
            self.root = s
        else:
            root._insert_non_full(root, k, data)

    def range_query(self, start_key, end_key):
        return self._range_query(self.root, start_key, end_key)

```

```

def _range_query(self, x, start_key, end_key):
    x.flush() # 在查询之前确保缓冲区中的数据已被处理
    result = []
    i = 0
    while i < len(x.keys) and x.keys[i][0] < start_key:
        i += 1
    while i < len(x.keys) and x.keys[i][0] <= end_key:
        result.append(x.keys[i][1])
        if not x.leaf:
            result += self._range_query(x.children[i], start_key, end_key)
        i += 1
    if not x.leaf:
        result += self._range_query(x.children[i], start_key, end_key)
    return result

```

# 连接到 MongoDB

```

mongo_client = MongoClient('mongodb://localhost:27017/')
database = mongo_client['book_database']
collection = database['book_collection']

```

class BookBeTreeIndex:

```

    def __init__(self, t):
        self.index_by_year = BeTree(t)
        self.index_by_reviews = BeTree(t)          #两个索引

```

def insert\_books(self):

```

    for book in collection.find():
        publish_year = book['出版时间']
        self.index_by_year.insert(publish_year, book)

```

def query\_books(self, start\_year, end\_year, min\_reviews):

```

    # 首先根据出版时间进行范围查询
    result_by_year = self.index_by_year.range_query(start_year, end_year)
    # 然后在结果中根据评论数量进行二次范围查询
    for book in result_by_year:
        review_count = book['评论数量']
        self.index_by_reviews.insert(review_count, book)
    result=self.index_by_reviews.range_query(min_reviews, float('inf'))
    return result

```

```

# 初始化并插入数据
book_index = BookBeTreeIndex(3)
book_index.insert_books()

# 范围查询
start_year = 1990
end_year = 2020
min_reviews = 50000
books = book_index.query_books(start_year, end_year, min_reviews)

# 按评分降序排序并输出前十
sorted_books = sorted(books, key=lambda x: float(x['评分']), reverse=True)[:10]

print("\n 出版时间在 1990 和 2020 之间且评论数量大于 50000 的书籍按评分降序排序前十：")
for book in sorted_books:
    pprint(book)

# 交互界面
if __name__ == "__main__":
    while True:
        print("请输入要进行的操作：(1、查询书籍详细信息 0、退出)")
        choose = input()
        if choose == '1':
            print("请输入要查询的出版年份区间和最少评论数量（格式：起始年份 结束年份 最少评论数量）:")
            query_params = input().split()
            if len(query_params) == 3:
                start_year = int(query_params[0])
                end_year = int(query_params[1])
                min_reviews = int(query_params[2])
                books = book_index.query_books(start_year, end_year, min_reviews)

                sorted_books = sorted(books, key=lambda x: float(x['评分']), reverse=True)[:10]

                print(f"\n 出版时间在 {start_year} 和 {end_year} 年之间且评论数量大于 {min_reviews} 的书籍按评分降序排序前十：")
                for book in sorted_books:
                    pprint(book)

```

```
        else:
            print("输入格式错误，请重新输入")
    elif choose == '0':
        sys.exit()
    else:
        print("无效的选择，请重新输入")
```

## 6alex\_in\_and\_le.py:

```
from pymongo import MongoClient
from pprint import pprint
import numpy as np
from sklearn.exceptions import NotFittedError
from sklearn.linear_model import LinearRegression

class GappedArray:
    def __init__(self, size):
        self.size = size
        self.array = [None] * size
        self.gaps = [True] * size
        self.count = 0

    def insert(self, index, value):
        if self.is_full():
            raise Exception("Array is full")

        if index < 0 or index >= self.size:
            raise IndexError("Index out of range")

        if self.gaps[index]:
            self.array[index] = value
            self.gaps[index] = False
        else:
            self.create_gap(index)
            self.array[index] = value
            self.gaps[index] = False

        self.count += 1
```

```

def create_gap(self, index):
    # Find the nearest gap
    nearest_gap_index = -1
    for i in range(self.size):
        if self.gaps[i]:
            if nearest_gap_index == -1 or abs(i - index) < abs(nearest_gap_index - index):
                nearest_gap_index = i

    if nearest_gap_index == -1:
        raise Exception("No gaps available")

    # Move elements to create a gap at the desired index
    if nearest_gap_index < index:
        for i in range(nearest_gap_index, index):
            self.array[i] = self.array[i + 1]
            self.gaps[i] = self.gaps[i + 1]
    else:
        for i in range(nearest_gap_index, index, -1):
            self.array[i] = self.array[i - 1]
            self.gaps[i] = self.gaps[i - 1]

    self.gaps[index] = True

def search(self, value):
    for i, item in enumerate(self.array):
        if item == value:
            return i
    return -1

def get_items(self):
    return [item for item in self.array if item is not None]

def is_full(self):
    return self.count >= self.size

def __len__(self):
    return self.count

```

```

import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.exceptions import NotFittedError

class InternalNode:
    def __init__(self, order):
        self.order = order
        self.keys = []
        self.children = []
        self.model = LinearRegression()

    def insert(self, key, child):
        self.keys.append(key)
        self.children.append(child)
        self.keys.sort()
        self.update_model()

    def update_model(self):
        if len(self.keys) > 1:
            X = np.array(self.keys).reshape(-1, 1)
            y = np.arange(len(self.keys))
            self.model.fit(X, y)

    def predict_position(self, key):
        try:
            return int(self.model.predict(np.array([[key]]))[0])
        except NotFittedError:
            return 0
        except Exception as e:
            print(f"Error in prediction: {e}")
            return 0

    def split(self):
        mid_index = len(self.keys) // 2
        mid_key = self.keys[mid_index]

        left = InternalNode(self.order)
        right = InternalNode(self.order)

        left.keys = self.keys[:mid_index]
        left.children = self.children[:mid_index + 1]

```



```

left.update_model()

right.keys = self.keys[mid_index + 1:]
right.children = self.children[mid_index + 1:]
right.update_model()

return mid_key, left, right

```

```
class LeafNode:
```

```

    def __init__(self, order):
        self.order = order
        self.keys = GappedArray(order)
        self.data = GappedArray(order)
        self.next = None
        self.model = LinearRegression()

```

```

    def insert(self, key, value):
        index = self.predict_position(key)
        if index is None or index >= self.order:
            index = self.order // 2
        if index < 0:
            index = 0
        if index < self.order and self.keys.gaps[index]:
            self.keys.insert(index, key)
            self.data.insert(index, value)
        else:
            exp = 1
            while index + exp < self.order and self.keys.array[index + exp] is

```

```
not None:
```

```

            exp *= 2
            i = index + exp
            if i >= self.order:
                i = self.order - 1
            while i > index and self.keys.array[i] is not None:
                i -= 1
            self.keys.insert(i, key)
            self.data.insert(i, value)
        self.update_model()

```

```

    def update_model(self):

```

```

valid_keys = []
valid_indices = []
for i, key in enumerate(self.keys.array):
    if key is not None:
        valid_keys.append(key)
        valid_indices.append(i)
if len(valid_keys) > 1:
    X = np.array(valid_keys).reshape(-1, 1)
    y = np.array(valid_indices)
    self.model.fit(X, y)

def predict_position(self, key):
    try:
        valid_keys = [k for k in self.keys.array if k is not None]
        if len(valid_keys) > 1:
            predicted_index = int(self.model.predict(np.array([[key]]))[0])
            # Ensure the predicted index is within valid range
            return max(0, min(predicted_index, len(valid_keys) - 1))
        except NotFittedError:
            return 0
    except Exception as e:
        print(f'Error in prediction: {e}')
        return 0

def split(self):
    mid_index = self.order // 2

    left = LeafNode(self.order)
    right = LeafNode(self.order)

    left.keys.array[:mid_index] = self.keys.array[:mid_index]
    left.data.array[:mid_index] = self.data.array[:mid_index]
    right.keys.array[:self.order - mid_index] = self.keys.array[mid_index:]
    right.data.array[:self.order - mid_index] = self.data.array[mid_index:]

    left.update_model()
    right.update_model()

    left.next = right
    right.next = self.next

```

```
return self.keys.array[mid_index], left, right
```

```
class ALEX:
```

```
    def __init__(self, order):
        self.order = order
        self.root = LeafNode(order)
```

```
    def insert(self, key, value):
        node = self.root
        parent_stack = []
        while isinstance(node, InternalNode):
            pos = node.predict_position(key)
            if pos < 0:
                pos = 0
            if pos is None or pos >= len(node.keys):
                pos = len(node.keys) - 1
            while pos < len(node.keys) and key > node.keys[pos]:
                pos += 1
            while pos > 0 and key < node.keys[pos - 1]:
                pos -= 1
            parent_stack.append((node, pos))
            node = node.children[pos]
```

```
node.insert(key, value)
```

```
if node.keys.array.count(None) == 0:
    mid_key, left, right = node.split()
    if parent_stack:
        parent_node, pos = parent_stack.pop()
        parent_node.keys.insert(pos, mid_key)
        parent_node.children[pos] = left
        parent_node.children.insert(pos + 1, right)
        while len(parent_node.keys) > self.order - 1:
            if parent_stack:
                grandparent_node, parent_pos = parent_stack.pop()
                mid_key, left, right = parent_node.split()
                grandparent_node.keys.insert(parent_pos, mid_key)
                grandparent_node.children[parent_pos] = left
                grandparent_node.children.insert(parent_pos + 1, right)
                parent_node = grandparent_node
            else:
```

```

        mid_key, left, right = parent_node.split()
        new_root = InternalNode(self.order)
        new_root.keys = [mid_key]
        new_root.children = [left, right]
        self.root = new_root
        break

    else:
        new_root = InternalNode(self.order)
        new_root.keys = [mid_key]
        new_root.children = [left, right]
        self.root = new_root

def search(self, key):
    node = self.root
    while isinstance(node, InternalNode):
        pos = node.predict_position(key)
        node = node.children[pos]
    index = node.keys.search(key)
    if index != -1:
        return node.data.array[index]
    return None

# def range_query(self, start_key, end_key):
#     results = []
#     node = self.root
#
#     while isinstance(node, InternalNode):
#         node = node.children[0]
#
#     while node is not None:
#         for i in range(node.order):
#             if node.keys.array[i] is not None and start_key <= node.key
s.array[i] <= end_key:
#                 results.append(node.data.array[i])
#             node = node.next
#     return results

"""
结合 B+树范围查询进行修改
"""

```

```

def range_query(self, start_key, end_key):
    results = []
    node = self.root

    # 找到第一个大于等于 start_key 的叶子节点
    while isinstance(node, InternalNode):
        pos = node.predict_position(start_key)
        if pos < 0:
            pos = 0
        if pos >= len(node.keys):
            pos = len(node.keys) - 1
        node = node.children[pos]

    # 遍历叶子节点链表，直到找到第一个大于 end_key 的键
    while node is not None:
        for i, key in enumerate(node.keys.array):
            if key is not None and start_key <= key <= end_key:
                results.append(node.data.array[i])
            if key is not None and key > end_key:
                break
        node = node.next

    return results

```

# 连接到 MongoDB

```

mongo_client = MongoClient('mongodb://localhost:27017/')
database = mongo_client['book_database']
collection = database['book_collection']

```

class BookAlexIndex:

```

    def __init__(self):
        self.index_by_year = ALEX(6) # 按年份索引的数据结构

    def insert_books(self):
        for book in collection.find():
            publish_year = book['出版时间']
            self.index_by_year.insert(publish_year, book) # 按年份插入书籍

    def query_books(self, start_year, end_year):

```

```

result = self.index_by_year.range_query(start_year, end_year)
return result

```

# 初始化并插入数据

```
book_index = BookAlexIndex()
```

```
book_index.insert_books()
```

```
start_year = 1970
```

```
end_year = 1990
```

```
books = book_index.query_books(start_year, end_year)
```

```
print(books)
```

```
sorted_books = sorted(books, key=lambda x: float(x['评分']), reverse=True)[:10]
```

```
print(f'\n 出版时间在 1970 和 1990 年之间的书籍按评分降序排序前十：')
```

```
for book in sorted_books:
```

```
    pprint(book)
```

## 6. 程序运行结果

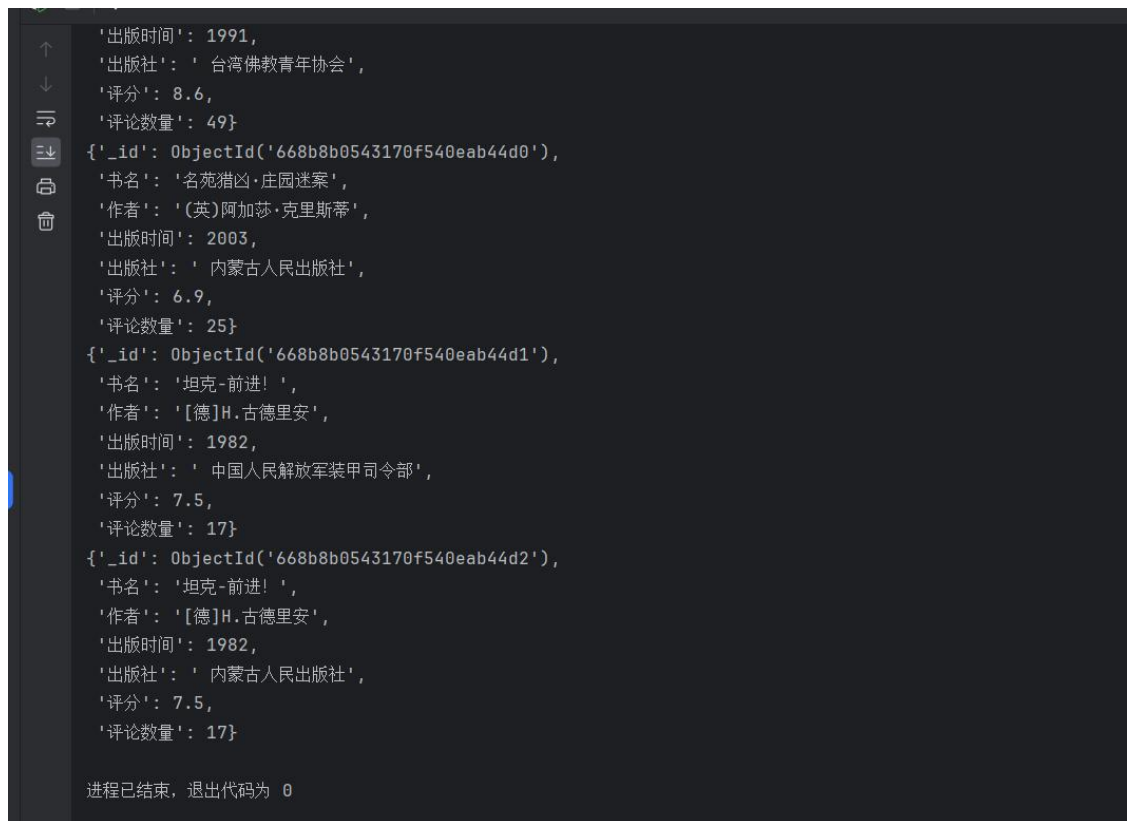
### 6.1 题目一运行结果

经过清洗后的数据保存到 cleandata.csv 文件中：

	A	B	C	D	E	F	G	H
1	Unnamed: 0	书名	作者	出版社	出版时间	评分	评论数量	
2	440	白马啸西风	金庸	海口-南海出版社	1993	7.5	14130	
3	443	盗墓笔记	南派三叔	上海文化出版社	2011	8.6	11038	
4	445	最游记	峰仓和也	河南大学出版社	2002	8	8772	
5	446	蜂蜜與四葉草 1	羽海野千花	玉皇朝出版集团	2004	9.1	8037	
6	450	贫穷贵公子	[日] 森永あい	东立出版社	2001	8.2	6494	
7	452	Level E	富樫義博	集英社授權天下出版	1995	9.2	5341	
8	455	中国式青春	今何在	四川人民出版社	2006	7.4	5046	
9	456	又一春	大风刮过	S.A	2006	8.3	4900	
10	457	桃花债 上	大风刮过	威向	2008	8.4	4362	
11	458	风非离	风维	丰书馆文化	2003	8	4287	
12	459	纨绔	公子欢喜	龙马	2007	8.5	4253	
13	460	白木園圓舞曲 01	齋藤千穗	大然文化	1995	8.2	3683	
14	462	忽爾今夏	亦舒	天地圖書有限公司	1988	7.7	3596	
15	463	电影少女	桂正和	天下出版有限公司	2003	7.7	3390	
16	466	桃花债(下)	大风刮过	威向	2008	8.5	3063	
17	467	朝花夕拾	亦舒	天地圖書有限公司	1986	8.2	2868	
18	468	珊瑚跳跳的仙太郎(1)	布浦翼	尖端出版社	1994	8.9	2835	
19	470	思凡	公子欢喜	龙马	2007	8.1	2693	
20	472	归途	蓝淋	威向	2005	8.4	2545	
21	473	纽约纽约	罗川真理茂	大然出版社	1998	8.9	2460	
22	476	戀物語 01	齋藤千穗	大然文化	1993	8.2	2244	
23	477	看朱成碧(上)	款款	倍乐文化	2006	8.4	2208	
24	478	殊途	蓝淋	威向	2005	8.4	2207	
25	479	无处可寻	蓝淋	鲜欢文化	2005	8.2	2161	
26	480	百鬼夜行抄1 [東立]	今市子	東立出版社有限公司	1997	9	2047	
27	481	拥抱春天的罗曼史	新田祐克	远方出版社	2004	8.6	2000	
28	482	金庸全集	金庸	生活·读书·新知三联书店	1994	9.5	1994	
29	483	庸君	公子欢喜	龙马	2007	7.9	1937	
30	484	魔道祖师	墨香铜臭	平心工作室	2016	8.5	1886	

图 10 cleandata.csv

将存有清洗后的数据的 MongoDB 中的文件逐一打印，这里展示最后几份数据：



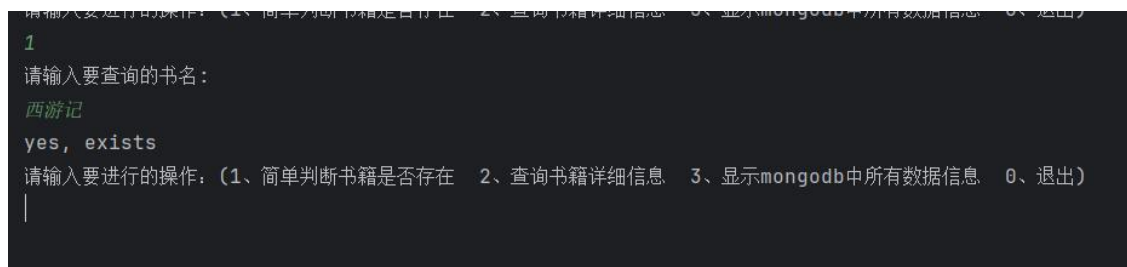
```
'出版时间': 1991,
'出版社': '台湾佛教青年协会',
'评分': 8.6,
'评论数量': 49}
{'_id': ObjectId('668b8b0543170f540eab44d0'),
'书名': '名苑猎凶·庄园谜案',
'作者': '(英)阿加莎·克里斯蒂',
'出版时间': 2003,
'出版社': '内蒙古人民出版社',
'评分': 6.9,
'评论数量': 25}
{'_id': ObjectId('668b8b0543170f540eab44d1'),
'书名': '坦克-前进!',
'作者': '[德]H.古德里安',
'出版时间': 1982,
'出版社': '中国人民解放军装甲司令部',
'评分': 7.5,
'评论数量': 17}
{'_id': ObjectId('668b8b0543170f540eab44d2'),
'书名': '坦克-前进!',
'作者': '[德]H.古德里安',
'出版时间': 1982,
'出版社': '内蒙古人民出版社',
'评分': 7.5,
'评论数量': 17}

进程已结束，退出代码为 0
```

图 11 MongoDB 中所存数据

## 6.2 题目二运行结果

(1) 经过 bf 过滤器过滤后的数据，简单查询书籍是否存在（即输出“yes”或者“no”）：



```
请输入要查询的书名：
西游记
yes, exists
请输入要进行的操作：(1、简单判断书籍是否存在 2、查询书籍详细信息 3、显示mongodb中所有数据信息 0、退出)
```

图 12 bf 过滤器简单查询书籍是否存在

(2) 经过 bf 过滤器过滤后的数据，精确查询《红楼梦》，可以显示列表类型的“出版社”、“出版时间”、“评分”、“评论数量”。

例如，下图中《红楼梦》一共有六个同的出版社，于是“出版社”、“出版时间”、“评分”、“评论数量”四个列表中的元素个数均为 6 个，分别存储了

各个出版社对应的信息。

```
C:\Users\1234\conda\envs\pytorch\python.exe E:\Python_file\book_project0\4cf_query.py
请输入要进行的操作：(1、简单判断书籍是否存在 2、查询书籍详细信息 3、显示mongodb中所有数据信息 0、退出)
2
请输入要查询的书名：
红楼梦
{'_id': ObjectId('668b6bad3f86d1a5c63a4e4b'), '书名': '红楼梦', '作者': '曹雪芹', '出版社': ['中华书局', '上海古籍出版社', '人民文学出版社', '三秦出版社', '金瓶出版社', '岳麓书社'], '出版时
请输入要进行的操作：(1、简单判断书籍是否存在 2、查询书籍详细信息 3、显示mongodb中所有数据信息 0、退出)
```

图 13 查询有多种出版社信息的《红楼梦》

```
社', '三秦出版社', '金瓶出版社', '岳麓书社'], '出版时间': [2012, 1995, 1996, 1994, 2006, 2004], '评分': [9.7, 9.5, 9.5, 9.6, 9.4, 9.5], '评论数量': [39, 1082, 111576, 66, 143, 928]}
```

图 14 查询有多种出版社信息的《红楼梦》续

(3) 经过 cf 过滤器过滤后的数据，简单查询书籍《小王子》是否存在（即输出“yes”或者“no”）：

```
行 3bf_query x 6alex_try0 (1) x 4cf_query x
C:\Users\1234\conda\envs\pytorch\python.exe E:\Python_file\book_project0\4cf_query.py
请输入要进行的操作：(1、简单判断书籍是否存在 2、查询书籍详细信息 3、显示mongodb中所有数据信息 0、退出)
1
请输入要查询的书名：
小王子
yes, exists
请输入要进行的操作：(1、简单判断书籍是否存在 2、查询书籍详细信息 3、显示mongodb中所有数据信息 0、退出)
|
```

图 15 简单查询《小王子》是否存在

(4) 经过 cf 过滤器过滤后的数据，精确查询《一个陌生女人的来信》，可以显示列表类型的“出版社”、“出版时间”、“评分”、“评论数量”。

例如，下图中《一个陌生女人的来信》一共有六个同的出版社，于是“出版社”、“出版时间”、“评分”、“评论数量”四个列表中的元素个数均为 6 个，分别存储了各个出版社对应的信息。

6.3 题目三运行结果

(1) 利用 $B^{\epsilon}$ -Tree 索引，将出版时间在 1990 和 2020 之间以及评论数量大于 50000 的书籍提取出来，并按照评分降序排序，输出前十，结果如下：



```
运行 3bf_query x 6alex_try0 (1) x 4cf_query x 5func3_had_buffer (1) x
C:\Users\1234\conda\envs\pytorch\python.exe E:\Python_file\book_project0\5func3_had_buffer.py

出版时间在1990和2020之间且评论数量大于50000的书籍按评分降序排序前十：
{'_id': ObjectId('668b8b0543170f540eaaae2e'),
 '书名': '红楼梦',
 '作者': '[清] 曹雪芹 著',
 '出版时间': 1996,
 '出版社': '人民文学出版社',
 '评分': 9.5,
 '评论数量': 111576}
{'_id': ObjectId('668b8b0543170f540eaac916'),
 '书名': '飘(上下)',
 '作者': '[美国] 玛格丽特·米切尔',
 '出版时间': 2000,
 '出版社': '译林出版社',
 '评分': 9.3,
 '评论数量': 72783}
{'_id': ObjectId('668b8b0543170f540eaacef0'),
 '书名': '三国演义(全二册)',
 '作者': '[明] 罗贯中',
 '出版时间': 1998,
 '出版社': '人民文学出版社',
 '评分': 9.2,
 '评论数量': 52278}
{'_id': ObjectId('668b8b0543170f540eaad3a3'),
 '书名': '三体Ⅲ',
 '作者': '刘慈欣',
 '出版时间': 2010,
 '出版社': '重庆出版社',
 '评分': 9.1,
 '评论数量': 53007}
```

图 16 Bε树索引实现范围查找

```
运行 3bf_query x 6alex_try0 (1) x 4cf_query x 5func3_had_buffer (1) x
'书名': '天龙八部',
'作者': '金庸',
'出版时间': 1994,
'出版社': '三联书店',
'评分': 9.1,
'评论数量': 53007}
{'_id': ObjectId('668b8b0543170f540eab3b15'),
 '书名': '活着',
 '作者': '余华',
 '出版时间': 1998,
 '出版社': '南海出版公司',
 '评分': 9.1,
 '评论数量': 118521}
{'_id': ObjectId('668b8b0543170f540eaaf538'),
 '书名': '白夜行',
 '作者': '[日] 东野圭吾',
 '出版时间': 2008,
 '出版社': '南海出版公司',
 '评分': 9.1,
 '评论数量': 170493}
{'_id': ObjectId('668b8b0543170f540eab2a4b'),
 '书名': '哈利·波特与魔法石',
 '作者': '[英] J. K. 罗琳',
 '出版时间': 2000,
 '出版社': '人民文学出版社',
 '评分': 9.0,
 '评论数量': 81795}
请输入要进行的操作：(1、查询书籍详细信息 0、退出)
```

图 17 Bε树索引实现范围查找

(2) 利用 ALEX 索引论文代码 api,实现范围查询,将出版时间在 2000 和 2020 之间以及评论数量大于 50000 的书籍提取出来,并按照评分降序排序,输出前十:

```

inserting
inserting
inserting
飘 (上下), [美国] 玛格丽特·米切尔, 译林出版社, 2000, 9.3, 72783
三体Ⅱ 刘慈欣, 重庆出版社, 2008, 9.2, 73112
百年孤独, [哥伦比亚] 加西亚·马尔克斯, 南海出版公司, 2011, 9.2, 91384
三体Ⅲ 刘慈欣, 重庆出版社, 2010, 9.2, 72624
白夜行, [日] 东野圭吾, 南海出版公司, 2008, 9.1, 170493
哈利·波特与魔法石, [英] J. K. 罗琳, 人民文学出版社, 2000, 9, 81795
小王子, [法] 圣埃克苏佩里, 人民文学出版社, 2003, 9, 209602
哈利·波特与火焰杯, [英] J. K. 罗琳, 人民文学出版社, 2001, 8.9, 67587
哈利·波特与阿兹卡班的囚徒, [英] J. K. 罗琳, 人民文学出版社, 2000, 8.9, 69633
哈利·波特与死亡圣器, [英] J. K. 罗琳, 人民文学出版社, 2007, 8.9, 53122

C:\ALEX-master\build\Debug\example.exe (进程 27340)已退出, 代码为 0。
按任意键关闭此窗口. . .|

```

图 18 C++实现 ALEX 索引输出结果

(3) 利用 Python 实现 ALEX 索引, 将出版时间在 1970 和 1990 之间的书籍提取出来, 并按照评分降序排序, 输出前十, 结果如下:

```

出版时间在1970和1990年之间的书籍按评分降序排序前十:
{'_id': ObjectId('668b8b0543170f540eaa879a'),
 '书名': '全本金瓶梅词话',
 '作者': '兰陵笑笑生',
 '出版时间': 1982,
 '出版社': '香港太平书局',
 '评分': 9.3,
 '评论数量': 32}
{'_id': ObjectId('668b8b0543170f540eaa87bd'),
 '书名': 'SUSAN SONTAG ON PHOTOGRAPHY',
 '作者': 'SUSAN SONTAG',
 '出版时间': 1978,
 '出版社': 'ALLEN LANE',
 '评分': 9.1,
 '评论数量': 28}
{'_id': ObjectId('668b8b0543170f540eaa877a'),
 '书名': '食人魔窟. 日本关东军细菌战部队的恐怖内幕',
 '作者': '森村诚一',
 '出版时间': 1982,
 '出版社': '群众出版社',
 '评分': 9.1,
 '评论数量': 36}
{'_id': ObjectId('668b8b0543170f540eaa85c0'),
 '书名': '備忘錄',
 '作者': '夏宇',
 '出版时间': 1987,
 '出版社': '科华图书出版公司',
 '评分': 9.1,
 '评论数量': 1000}

```

图 19 python 实现 ALEX 索引输出结果

```
{'_id': ObjectId('668b8b0543170f540eaa8679'),
  '书名': '属灵操练礼赞',
  '作者': '傅士德',
  '出版时间': 1982,
  '出版社': '香港基督徒學生福音團契',
  '评分': 9.0,
  '评论数量': 116}
{'_id': ObjectId('668b8b0543170f540eaa867b'),
  '书名': '欧里庇得斯悲剧二种',
  '作者': '欧里庇得斯',
  '出版时间': 1979,
  '出版社': '人民文学出版社',
  '评分': 8.9,
  '评论数量': 114}
{'_id': ObjectId('668b8b0543170f540eaa8635'),
  '书名': '大师在喜马拉雅山',
  '作者': '喇嘛尊者 Swami Rama',
  '出版时间': 1982,
  '出版社': '中国瑜珈出版社',
  '评分': 8.9,
  '评论数量': 191}
{'_id': ObjectId('668b8b0543170f540eaa866f'),
  '书名': '鲁迅杂文选',
  '作者': '鲁迅',
  '出版时间': 1973,
  '出版社': '上海人民出版社',
  '评分': 8.9,
```

图 20 python 实现 ALEX 索引输出结果

```
'出版时间': 1982,
  '出版社': '中国瑜珈出版社',
  '评分': 8.9,
  '评论数量': 191}
{'_id': ObjectId('668b8b0543170f540eaa866f'),
  '书名': '鲁迅杂文选',
  '作者': '鲁迅',
  '出版时间': 1973,
  '出版社': '上海人民出版社',
  '评分': 8.9,
  '评论数量': 122}
{'_id': ObjectId('668b8b0543170f540eaa85c1'),
  '书名': '摄影构图学',
  '作者': '【美】本·克来门茨',
  '出版时间': 1983,
  '出版社': '长城出版社',
  '评分': 8.9,
  '评论数量': 993}
{'_id': ObjectId('668b8b0543170f540eab1563'),
  '书名': '石涛画语录',
  '作者': '宾亚杰',
  '出版时间': 1970,
  '出版社': '西泠印社出版社',
  '评分': 8.8,
  '评论数量': 55}
```

进程已结束，退出代码为 0

图 21 python 实现 ALEX 索引输出结果

## 7. 致谢

在此，我要向所有在我完成这份课程设计报告过程中给予帮助和支持的人们表示最诚挚的感谢。

首先，我要感谢我的授课老师邓泽老师，本次课程设计是基于《大数据存储与管理》这门课的知识，感谢老师课上的细心讲解，让我明白了各种知识点，他的严谨治学态度和无私奉献精神使我受益匪浅。

其次，我要感谢我的同学们，在与他们的讨论和交流中，我获得了许多新的思路和灵感，他们的建议和帮助对我的报告撰写起到了积极的推动作用。

此外，我还要感谢学校提供的良好学习环境和丰富的资源，这些都为我的课程设计提供了坚实的支持。

最后，我要感谢我的家人和朋友，他们的理解和支持是我不断前进的动力。