

简易 key-value 型 Database 设计报告

517030910206

陈景宇

2018.7

目录

- 目录.....2
- 一、 基本信息.....3
- 二、 提供接口.....4
- 三、 模块实现.....6
- 四、 功能实现.....7
- 五、 特点..... 10
 - 1、 不定长 value..... 10
 - 2、 存储缓存的使用 10
 - 3、 范围查找 10
- 六、 测试及分析11
 - 1、 正确性测试11
 - 1.1 大量数据测试11
 - 1.2 非寻常测试11
 - 1.3 综合测试 12
 - 2、 性能测试 13
 - 2.1 关于存储缓存性能优化的测试 13
 - 2.2 操作时间与数据量的关系 14
 - 2.3 操作时间与节点大小的关系 19
- 七、 待改进..... 20
- 八、 致谢以及参考 21

一、基本信息

-项目名称：简易的 key-value 型数据库

-主要数据结构：B+树

-开发环境：

操作系统：Windows 10 家庭版(17134)

处理器：Intel i7-7700HQ 2.8GHz

安装内存：16 G

系统类型：64 位操作系统，基于 x64 的处理器

-概述：实现<key, value>型数据的增加，删除，修改，查找，范围查找功能

二、提供接口

提供<unsigned long long, string>类型的<key, value>存储，string 的长度不定，但是建议 key 的长度小于 50 字节

- 打开

```
void DataBase::open(string rootname)
```

- rootname 为 index 文件的根文件名
- 一旦成功打开 index 根文件，那么将按照 index 根文件记录 b+树的跟与叶文件的开头，创建缓存

- 关闭

```
void DataBase::close()
```

- 储存缓存内容，结束整个程序

- 存储

```
bool DataBase::Insert(unsigned long long key,string &record)
```

- unsigned long long 类型的 key
- string 类型的 record
- 储存 record, 如果 key 已经存在, 返回 false, 那并且返回相应的信息给用户, 如果 key 不存在, 返回 true, 并且返回给用户成功存储。存储操作将记入缓存并在适当的时候储存。

- 删除

```
bool DataBase::delete_one(unsigned long long key)
```

- unsigned long long 类型的 key
- 删除 key 所对应的 record 数据, 如果 key 存在那么返回 true, 并且给用户成功删除的信息, 如果 key 不存在那么返回 false, 并且给用户返回删除失败的信息。存储操作将记入缓存并在适当的时候储存。

- 修改

```
bool DataBase::change(unsigned long long key,string &record)
```

- unsigned long long 类型的 key
- string 类型的 record
- 如果 key 存在，那么返回 true，并且返回给用户成功修改的信息，如果 key 不存在那么返回给用户修改失败的信息。修改操作将计入缓存并在适当的时候储存。

- 查找

```
string DataBase::Find(unsigned long long key)
```

- unsigned long long 类型的 key
- 如果 key 存在那么返回 key 所对应的 record 数据，如果 key 不存在那么返回给用户"NONE"

- 范围查找

```
map<unsigned long long,string> DataBase::RangeFind(unsigned long long  
key1,unsigned long long key2)
```

- unsigned long long 类型的 key1, key2
- 将结果以 map 的形式返回，first 记录查到的 key，second 记录数据。

三、模块实现

IndexFile.h

存放了索引文件的 B+树实现，存放了对索引文件（即 B+树叶子节点）的各项操作，包括插入，删除，查找，打印索引等功能。

DataFile.h

存放了数据文件的行一条数据，删除一条数据，查为操作，包括加入找一条数据等

DataBase.h

存放了数据库的行为，包括了上面列举出来的个个接口（删除，插入，查找，更改，范围查找等）

Cache.h

存放了缓存的实现及各种行为（从缓存中读、存数据，将缓存中的数据储存到硬盘等）。

MyTest.cpp

存放了测试函数，用来测试数据库的各项功能。

DataBase.cpp

简易的用户界面的实现。

四、功能实现

- 打开

- 对应的索引文件的根文件，如果打开失败则要求用户重新输入正确的文件名
- 根据索引文件生成 B+树的根节点对象，关闭文件，并由此生成数据库对象
- 初始化数据读写缓存

- 关闭

- 储存缓存内容，停止程序运行

- 储存

- 首先先读取存储缓存 buffer 中是否存在 key 如果存在，则返回 repeat key
- 否则查找 b+树，如果查找到相应的 key 那么也返回 repeat key，如果没有查到相应的 key 那么将这对<key,value>写入写缓存中。

- 删除

- 首先查找存储缓存中是否存在，如果存在那么直接在存储缓存中删除
- 如果存储缓存中不存在，那么到 b+树中进行查找，如果找到，将节点读入缓存，在缓存中删除。
- 如果在存储缓存和 b+树中都没法找到相应的 key 那么删除失败

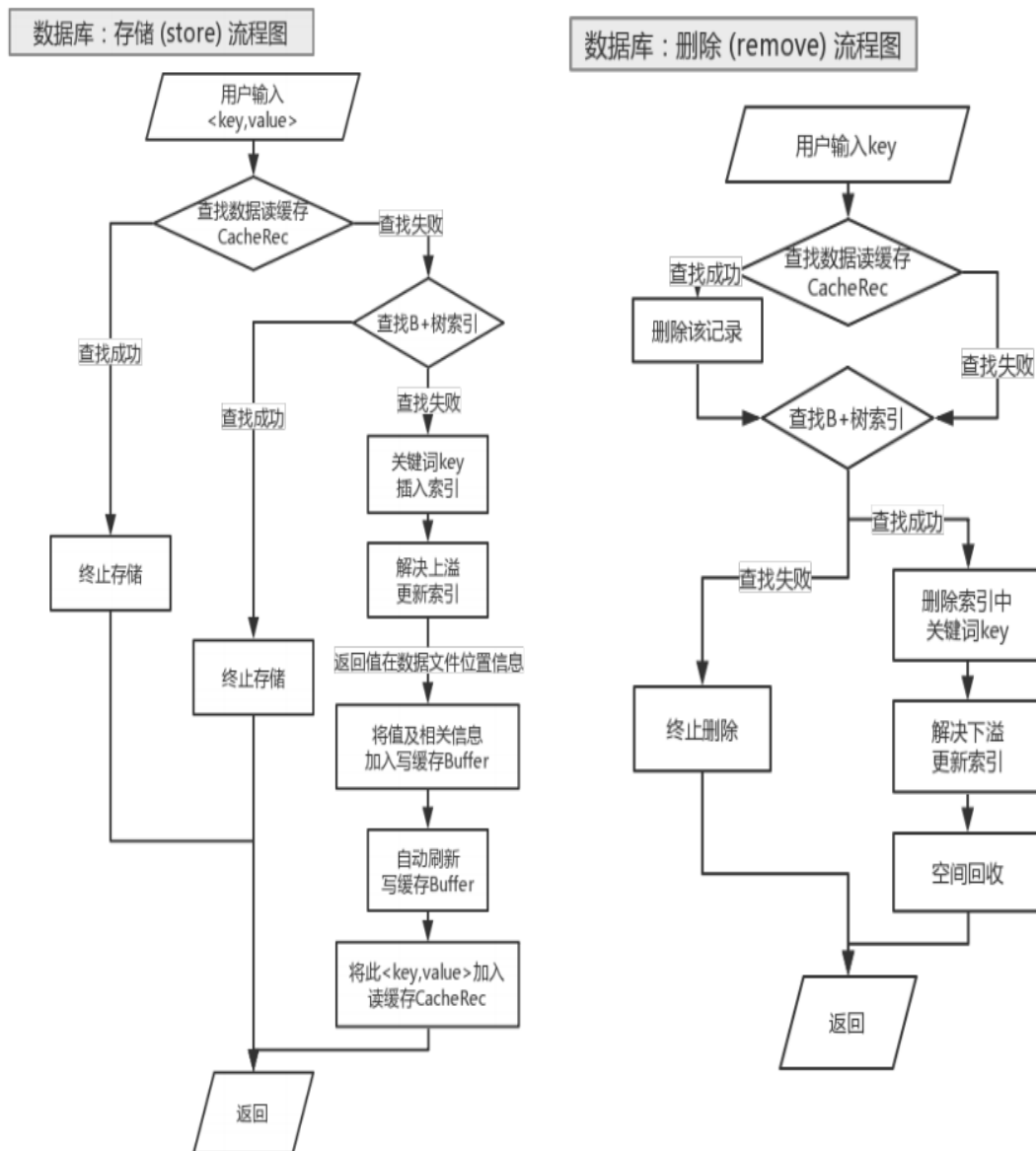
- 修改

- 查找，存储缓存中是否存在 key，再查找 b+树中是否存在 key，如果在 b+树中，则将该节点读入缓存，并在缓存中修改，如果在存储缓存中找到 key 直接改变存储缓存中 key 所对应的值
- 如果没有找到相应的 key 那么返回给用户相应的信息

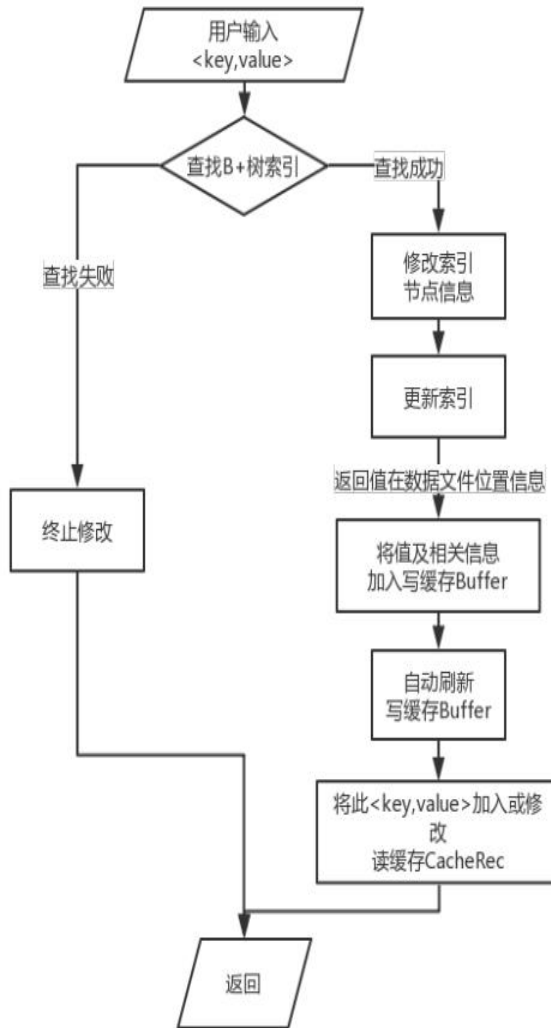
- 查找

- 查找存储 buffer 中是否存在相应的 key 如果存在那么返回所对应的 value, 如果 buffer 中不存在, 那么在 b+树中进行搜索, 如果 key 也不再 b+树中那么返回给用户搜索不到的信息

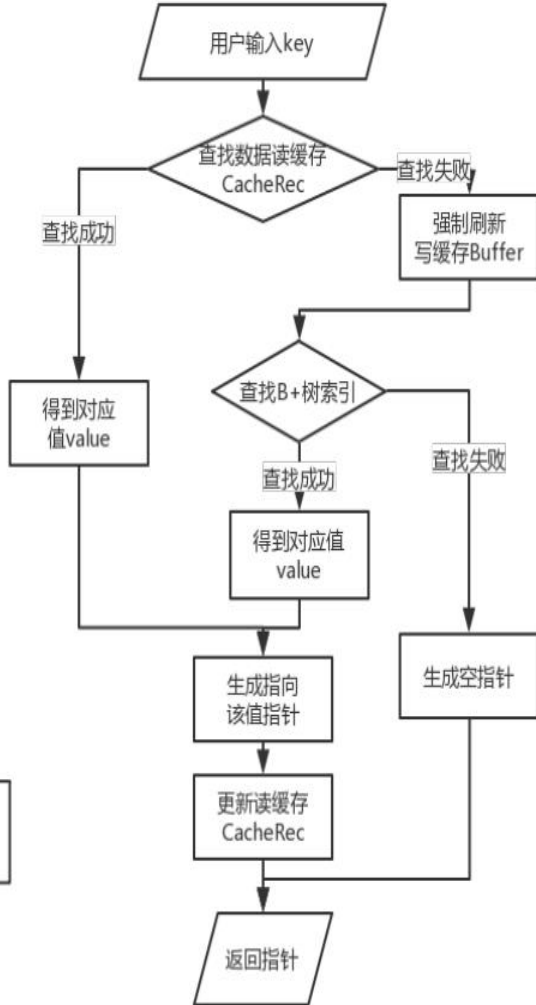
以下为工作流程图



数据库：修改 (modify) 流程图



数据库：查找 (fetch) 流程图



五、特点

1、不定长 value

- 可以接受不定长的 `string`
- 解决办法，在数据文件中用分隔符分隔数据，读取数据文件时，按分隔符将文件以 `vector` 形式读入，索引文件中不再记录数据文件的偏移量，而是其在 `vector` 中的位置，但这样做可能会造成空间和时间上的浪费。

2、存储缓存的使用

- 为何使用缓存：考虑到一般用户的习惯，刚刚写入的数据往往会再一次查阅，又因为该数据库的实现会有多个索引文件，直接操作会导致运行时间很长。因此我们需要记录一定数量的近期储存的 `<key,value>` 对到缓存中方便下一次的读取，提高效率。

- 如何实现：采用了 `map` 的数据结构

```
map<string,IndexFile> unsaved_file;  
map<string,DataFile> unsaved_data;
```

记录未储存的文件，其中，`IndexFile` 和 `DataFile` 分别为索引对象和数据对象。

- 功能：
 - 利用数据使用用的局部性(`locality`)，将近期使用用过的 `<key, value>` 保存。
 - 维护缓存容量，一旦缓存超过容量，储存缓存内容，并清空缓存。
- 一致性：
 - 何时更新：缓存中储存对象超过100个时
 - 如何更新：由于 `map` 直接储存了文件对象，储存时遍历 `map`，调用对象的 `store()` 方法进行储存更新，覆盖原文件。

3、范围查找

- 如何实现：由于使用了 `B+` 树的数据结构，进行范围查找时只需找到第一个存在的 `<key, value>` 对，然后通过链表结构快速查找。

六、测试及分析

1、正确性测试

1.1 大量数据测试

- 建立数据库，分别执行下列操作做
 - 随机储存 TIMES 条数据
 - 随机删除部分数据
 - 读取全部数据
 - 随机修改数据
 - 读取全部数据，进行对比
 - 随机储存 TIMES 条数据
 - 读取全部数据，进行对比
- 该测试主要是为了验证大量数据下的最正确性
- TIMES 分别取成 1000,5000,10000
- 三次操作全部正确

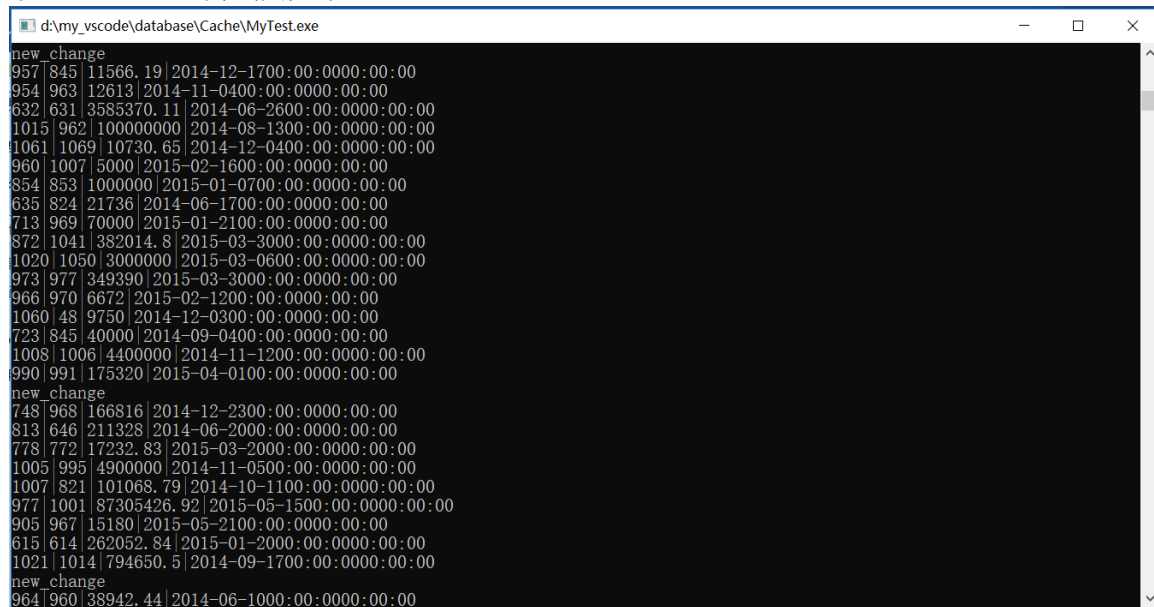
1.2 非寻常测试

- 建立数据库，分别执行以下操作
 - 随机储存 10000 条数据
 - 循环以下步骤多次
 - 储存一个特定关键字以及值，读取该关键字
 - 删除该关键字，读取该关键字
 - 删除该关键字，读取该关键字
 - 修改该关键字，读取该关键字
 - 存储该关键字以及一个不同的值
- 该测试主要是为了测试设计删除，存储，修改的异常操作
- 循环出现 value、false、false、false、new value

1.3 综合测试

- 建立一个数据库，执行老师提供的测试方法，先插入入 times 数据，后循环以下步骤多次：

- 随机读取一条数据
 - 每循环 37 次，随机删除一条记录
 - 每循环 11 次，随机添加一条记录并且读取该条记录
 - 每循环 17 次，随机替换一条记录为新记录。
- 该测试主要是设计综合存储，删除，修改，读取，操作
- 初次使用此测试时，程序会崩溃，经检查发现，bug 来源于整数除法的截断错误。
- 解决bug后，在每次循环读取中，都得到与插入时相同的值，或者该关键字被删除且读取返回false，测试图如下。



```
d:\my_vscode\database\Cache\MyTest.exe
new_change
957|845|11566.19|2014-12-1700:00:0000:00:00
954|963|12613|2014-11-0400:00:0000:00:00
632|631|3585370.11|2014-06-2600:00:0000:00:00
1015|962|100000000|2014-08-1300:00:0000:00:00
1061|1069|10730.65|2014-12-0400:00:0000:00:00
960|1007|5000|2015-02-1600:00:0000:00:00
854|853|1000000|2015-01-0700:00:0000:00:00
635|824|21736|2014-06-1700:00:0000:00:00
713|969|70000|2015-01-2100:00:0000:00:00
872|1041|382014.8|2015-03-3000:00:0000:00:00
1020|1050|3000000|2015-03-0600:00:0000:00:00
973|977|349390|2015-03-3000:00:0000:00:00
966|970|6672|2015-02-1200:00:0000:00:00
1060|48|9750|2014-12-0300:00:0000:00:00
723|845|40000|2014-09-0400:00:0000:00:00
1008|1006|4400000|2014-11-1200:00:0000:00:00
990|991|175320|2015-04-0100:00:0000:00:00
new_change
748|968|166816|2014-12-2300:00:0000:00:00
813|646|211328|2014-06-2000:00:0000:00:00
778|772|17232.83|2015-03-2000:00:0000:00:00
1005|995|4900000|2014-11-0500:00:0000:00:00
1007|821|101068.79|2014-10-1100:00:0000:00:00
977|1001|87305426.92|2015-05-1500:00:0000:00:00
905|967|15180|2015-05-2100:00:0000:00:00
615|614|262052.84|2015-01-2000:00:0000:00:00
1021|1014|794650.5|2014-09-1700:00:0000:00:00
new_change
964|960|38942.44|2014-06-1000:00:0000:00:00
```

2、性能测试

2.1 关于存储缓存性能优化的测试

因为考虑到数据的局部性，我们要搜索的往往是我们刚刚加入进去数据，也就是我们对刚刚操作的过得数据，再次访问的概率更大，同时，由于索引文件为多个文件，多次直接修改操作时间耗费巨大，所以这里我们采用了储存缓存，即刚刚更改过得数据，不直接写入磁盘，而是在缓存区暂时储存，这样当我们进行的操作的速度会大大加快。

这里我们采用最小节点为 2，最大节点数为 5 的 b+树，插入节点 5000 个，比较插入时间。

```
finish!  
time:135.274s!  
请按任意键继续. . .
```

```
finish!  
time:28.671s!  
root_name:index_Brch728.idx  
请按任意键继续. . .
```

我们可以明显的看出，采用了储存缓存之后，插入的速度将大大加快，后者所占用的时间是前者的 20%

- 思考：

两者的差距较大，和预计是相符的，我在建立索引及数据文件时，直接将 b+树的节点储存为文件，用文件路径代替指针，这样对文件的直接操作会大量的耗费时间，使用缓存后，可以减少对文件的直接操作，达到提高效率的目的。

另外，由于使用 B+树储存，我的最大节点数对操作时间的影响较大，详细的结果见 2.3。

2.2 操作时间与数据量的关系

注：为了简化试验，该项测试中统一使用节点大小为127，缓存上限为100个文件(节点)。为了检测数据库时间性能与数据量的关系，设计了如下操作流程：

1. 向数据库中写入 n 条记录，记录写入所需总时间Time1；
2. 通过关键字读取 n 条记录，记录读取所需总时间Time2；
3. 修改数据库中全部 n 条记录，记录修改所需总时间Time3；
4. 随机删除数据库中 $n/5$ 条记录，记录删除所需总时间 Time4；

-插入： n 分别取 10k, 100k, 500k, 1000k

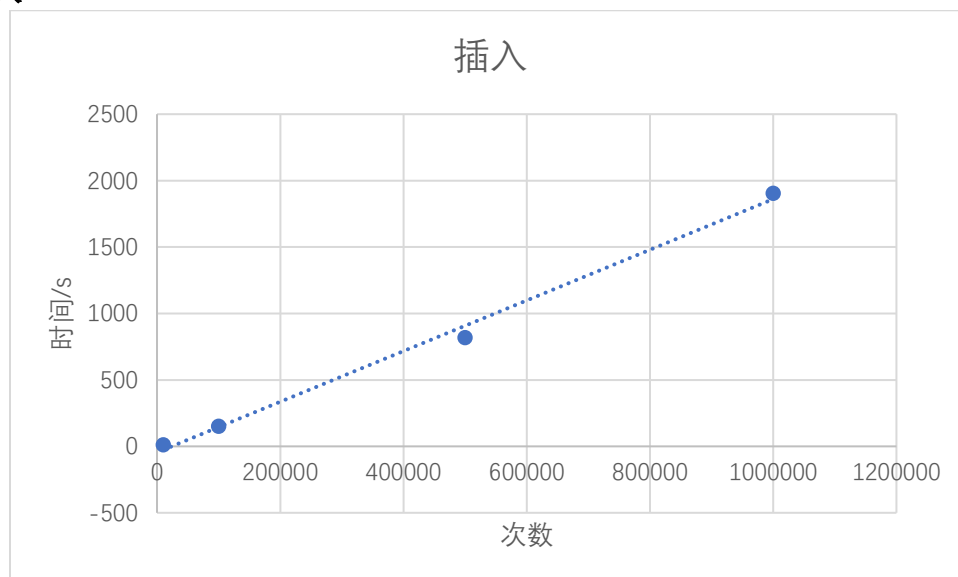
-查找： n 分别取 5k, 50k, 100k, 500k

-修改： n 分别取 1k, 5k, 10k, 50k

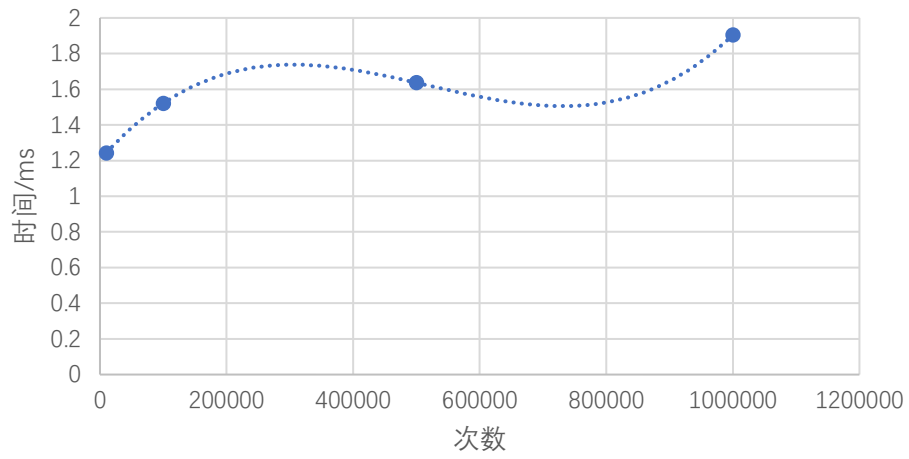
-删除： n 分别取 5k, 10k, 50k, 100k

-数据量选取考虑了程序性能

-插入

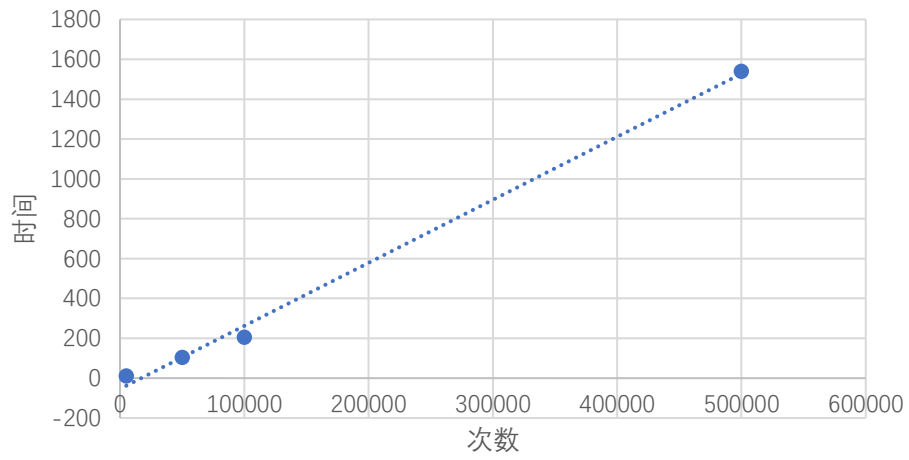


单次插入时间

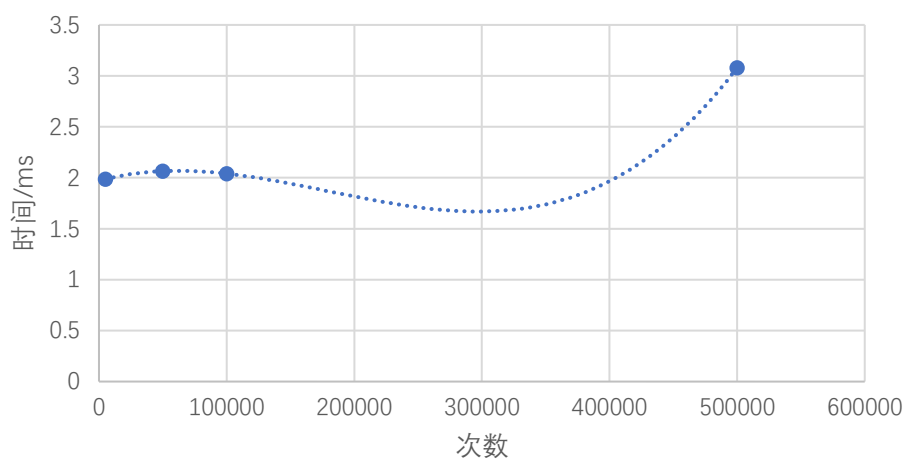


-查找

查找

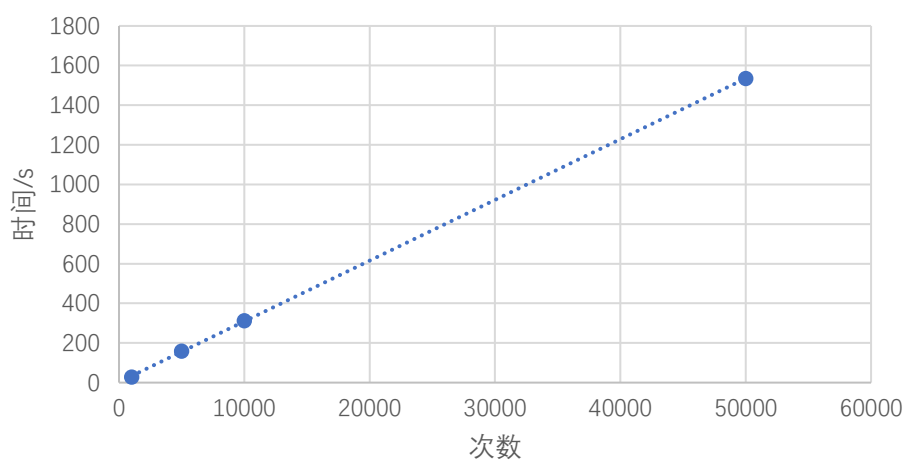


单次查找时间

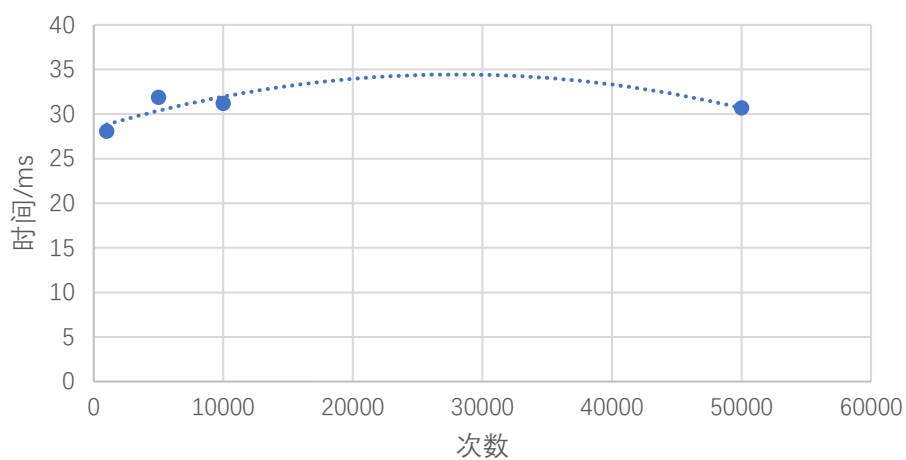


-修改

修改

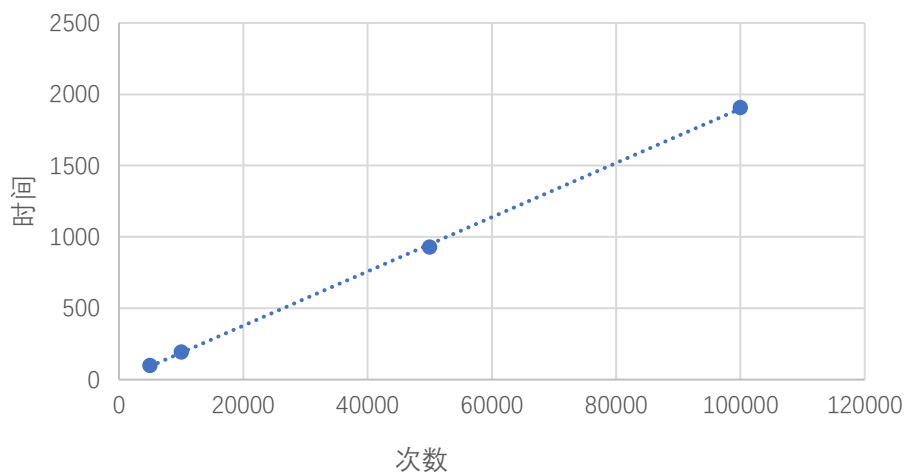


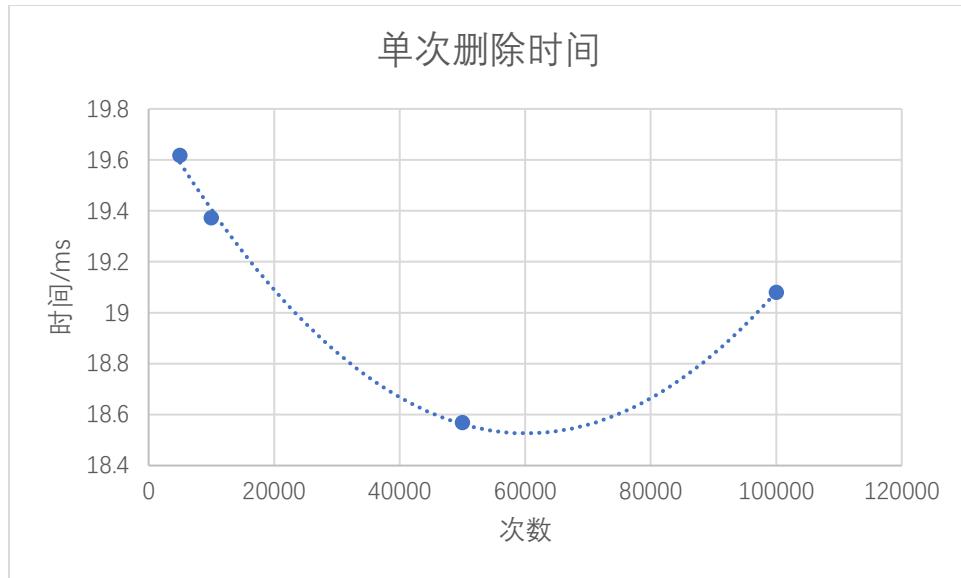
单次修改时间



-删除

删除





分析上述图表展示数据，可得到以下的结果：

- 数据库的插入新数据、查找数据、替换数据和删除数据操作，所需时间和数据库中的数据总量近似表现为线性关系；
- 随着数据库数据总量的增加，单次操作时间近似稳定，但有上升趋势，即呈现非线性趋势
- 删除与修改数据操作用时远高于插入、查找数据，且非线性趋势更明显。

结合上述的结果，我们可以分析：

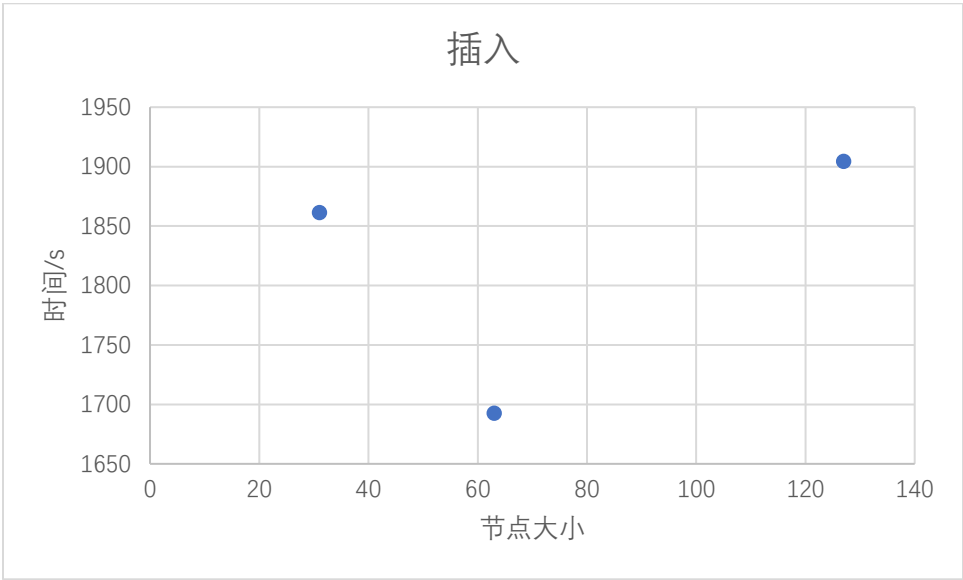
- 向空数据库中插入数据、查找数据、删除数据、修改数据都与数据库数据总量呈近似线性关系，这可能是由如下原因导致的：
 - 插入数据和查找数据用时的大部分都花费在遍历一个节点拥有的键(key)，这种遍历用时是线性的；
 - 删除数据和修改数据用时的大部分都花费在对原有数据的覆盖修改上，这种重新写文件的用时是线性的；
 - 当数据增加，B+ 树的高度以对数形式增加，相对于线性用时，在数据量较大而且节点不是过小时，在树层之间跳跃的时间充分小于遍历节点信息的时间，使得总用时表现出近似的线性性质；
- 替换和删除数据涉及更加复杂的内存操作和数据结构变化。
- 对于数据足够多之后，替换和删除操作用时相对数据变化量的关系的逐渐线性化，这种变化也与数据结构中的节点大小相关，更详细的结果见2.3

2.3 操作时间与节点大小的关系

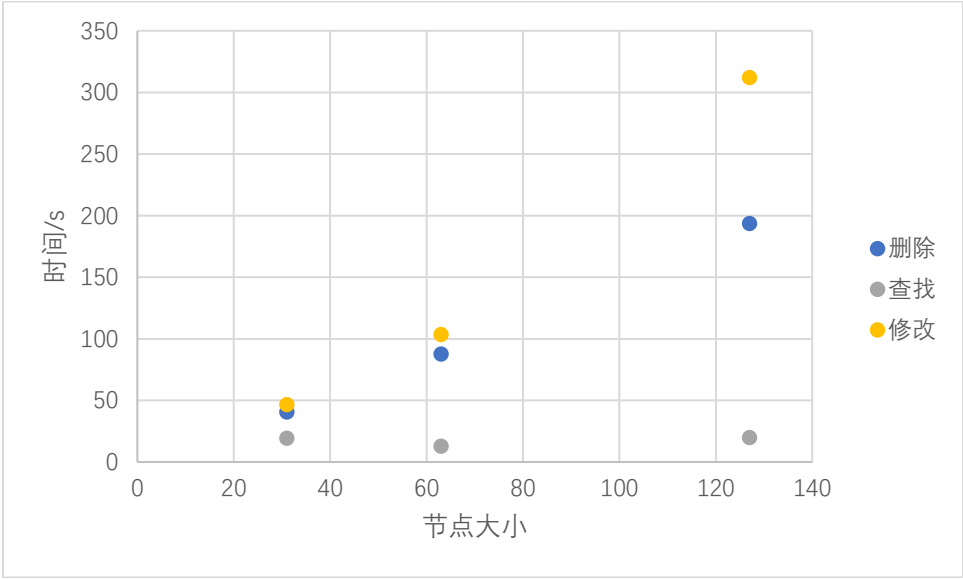
该试验过程和上一个试验相同，进行多次试验时更换节点大小maxsize。

分别取 nodesize 为 31, 63, 127 进行测试，得到如下结果：

-插入（1000000 条记录）



-删除、查找、修改（操作 10000 条数据）



分析上述图表展示数据，可得到以下的结果：

- 数据库的插入新数据、查找数据、替换数据和删除数据操作，所需时间和数据库中的节点大小密切相关。
- 对于插入操作，大概率存在一个最佳节点大小，对修改和删除操作，节点的越大，速度越慢，对于查找操作，节点大小几乎不影响速度。

七、待改进

- 空间：

- **空间利用：**索引文件在储存时储存了一些冗余信息，这导致文件过大且读写效率不高，应该将这些冗余信息删除。但由于时间限制及自身代码的缺陷，未能实现。

- **缓存：**缓存类不够完善，由于最初没有计划这一对象，后来发现插入效率过低才加入，导致仅能实现一些简单的功能。

- 时间：

- **算法：**查找 key 等过程使用了顺序查找，可以用更优秀的算法来节省时间，由于删除、修改等方法都在内部使用了查找函数，这种改进可以使得程序的整体效率提升。

- **文件读写：**文件储存时并非对原文件进行直接修改，而是将原有文件覆盖，这影响了文件的读写速度。

- 功能：

- **数据类型：**<key, value> 对中 key 的类型可以扩展到可比较 key 值的其他数据类型，而不仅仅只用正整数。

- **跨平台移植：**在技术水平达到之后，应使该程序在其他平台使用，而不仅仅在 window 10 操作系统上使用。

- 代码：

- **代码质量：**开发初期由于知识水平的限制，没能很好的进行规划，在经历多次大规模改写之后，代码可读性差，健壮性也不能让人满意，需要细致地进行重构。

- **模块组织：**虽然分了 4 个模块，但各个模块相互调用，结构不清晰，不能说实现了“模块化编程”。

- 整体架构：

- 经过测试发现自己的程序效率极低，单次操作在 ms 级，完全无法满足要求，经过自己思考以及请教老师助教，发现自己的实现思路存在很大问题，可能需要重新架构，换另外一种实现思路。

- 以上不足极大的影响了该数据库的质量，但由于时间关系，没能将上述问题解决。该数据库只是满足了基本的功能和结构，距离功能更完善、更具通用性和真正强大的数据库还有很远，可以做的还有很多，仅从效率方面就有很多可以改进的地方。

八、致谢以及参考

- 感谢暑假还抽出时间给我们上课戚老师，任老师
- 感谢在学习之余还耐心解答的班主任董志远，以及俞昕宜、曹金坤、冯二虎三位学长学姐的文档（它们给我的启发很大），以及其他的助教大大们，望手下留情。
- 感谢和我一起探讨问题的软工的大一小伙伴们
- 参考书籍《数据结构：清华大学出版》
- 参考网站：<https://www.zhihu.com/question/35382593>
<https://blog.csdn.net/cyl937/article/details/53585007>
<https://zhuanlan.zhihu.com/p/34916328>
<https://github.com/begeekmyfriend/bplustree/blob/disk-io/lib/bplustree.c>
https://blog.csdn.net/recall_yesterday/article/details/51476588
<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>