程序设计ⅡProject

Database文档

版本 <1.0>

文档信息

|  |  |
| --- | --- |
| 标题： | Database文档 |
| 文件位置： |  |
| 版本： | 1.0 |
| 提交人： | 田嘉禾 |
| 提交日期： | 1/1/2015 |
| 状态： |  |

修订历史记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **日期** | **版本** | **说明** | **作者** | **审核人** |
| 1/1/2015 | 1.0 | Database | 田嘉禾 | <姓名> |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

Database文档

# 简介

## 目的

本文档为程序设计ⅡProject：Database的设计文档，用于说明程序的实现情况

## 范围

本文档包括对使用数据结构的说明、对测试结果的说明以及对性能情况的分析

# 软件架构

Database类包含一个BplusTree index，和一个用于文件读写的fstream datafile，支持增删改查指定key-data对和随机增删改查的操作。支持对数据库文件进行重构，清除冗余信息。

Database中的函数会对数据文件进行操作，同时调用index的函数对索引进行更新。

BplusTree类为索引类，包含node\* root，为树的根节点。支持增删改查操作，在对象析构时会调用save()将索引内容存储到磁盘上，下次重新打开时可以load()。支持split(), borrow(), merge()等操作，用于维护树的性质。

struct node用num表示recs数组中元素个数，node\* first指向最左端的孩子节点，如果first为NULL，则表示该节点为leaf，否则为internal node；recs数组预留degree-1的空间，当rec填满时，会分裂当前节点，当rec数不够一半时，会从其他节点借或同其他节点合并。

key

key

key

…

key

key

key

key

…

key

key

key

key

…

key

key

key

key

…

key

struct rec包括string key和union{node\* ptr; unsigned long pos;}，当rec在internal node上，ptr用于指向孩子节点；当rec为leaf时，pos表示data文件中数据存储的位置。

# 测试说明

正确性测试：

1.随机存储nrec条数据

2.每次fetch并比较是否正确

3.每13次remove和replace一条数据

4.检查remove后是否还能获取数据

5.检查replace后数据是否一致

性能测试：

1.随机写nrec条记录

2.随机读nrec条记录

3.执行下面的循环nrec×5次：

(a) 随机读一条记录

(b) 每循环37次，随机删除一条记录

(c) 每循环11次，随机添加一条记录并读取这条记录

(d) 每循环17次，随机替换一条记录为新记录

4.删除所有记录，每删除一条记录，随机查找10条记录

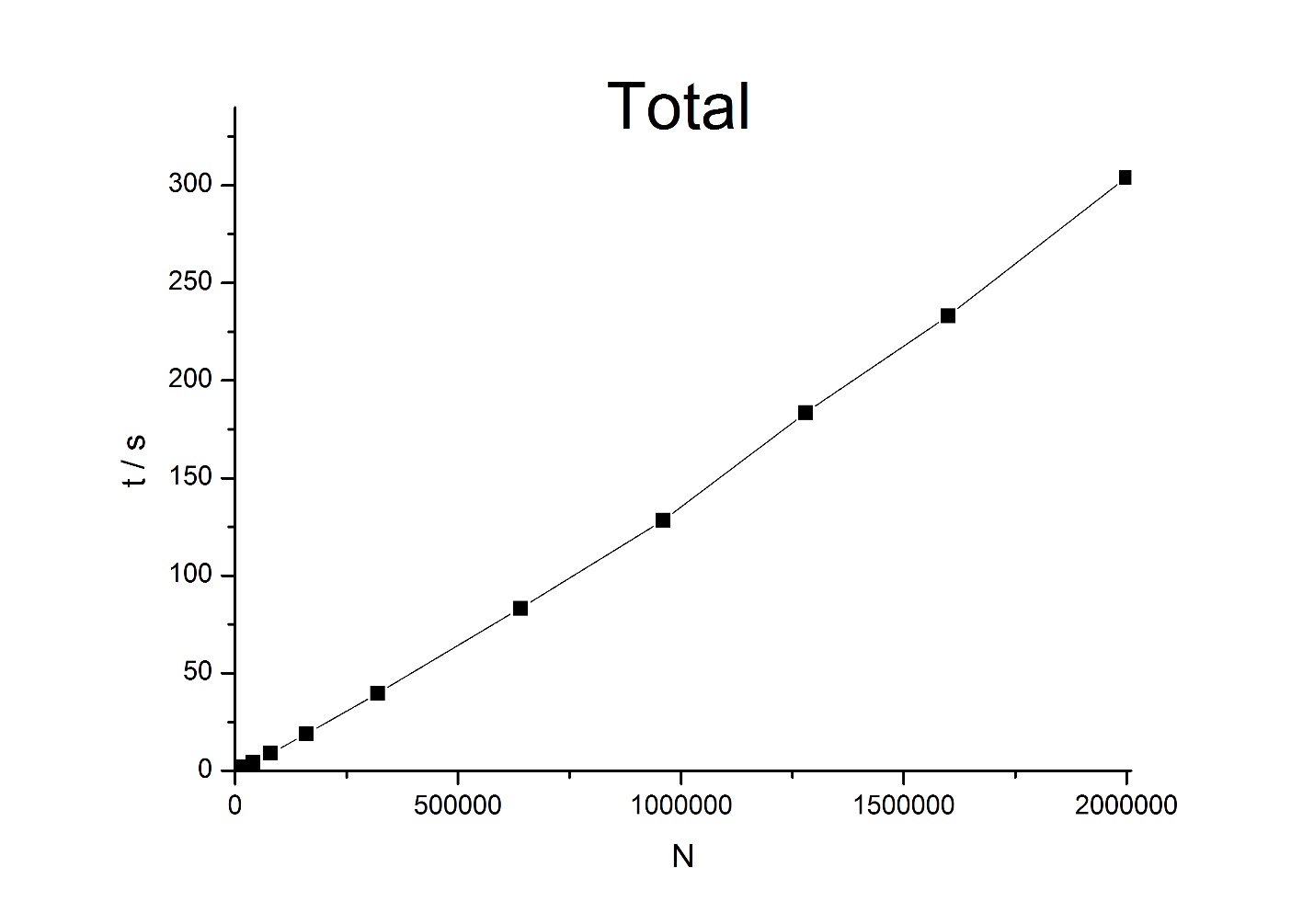
# 性能分析

对于百万级别的插入和查找操作，B+树的效率很高。这是由于degree足够大，使得树的level非常低，最多不超过6。本测试采用degree为16，则最多15个孩子，15^6=11400000，绝对不会超过6层。

对每层访问最多进行15次字符串比较，从而访问索引最多需6\*15=90次字符串比较，又因为目标记录位于每个孩子的概率相同，所以平均下来需6\*（15\*16/2）/15=48次字符串比较，（实际中比这个要小）忽略指针读写所花费的时间，这样操作下来是很快的。

store操作花费时间约为fetch操作的1.5倍，这是由于除了找出数据的位置，还需要对硬盘进行读写而产生的花费；Part4花费时间约为Part3的两倍，这是由于前者fetch的次数约为后者的两倍。对于replace操作，所花费的时间基本等同于store；对于remove操作，因为不需要对数据文件进行操作，只需要修改index，和fetch的时间基本持平。

当nrec=1000000时，总时间约130秒，共进行1450000次store，20000000次fetch，1500000次remove，300000次replace。



索引文件大小约为40MB，数据文件大小约为250MB，这是由randkey()和randdata()函数生成字符串大小决定的。因为数据量不大，索引文件可以全部存到内存里，这也就导致B+树对于磁盘读写的优势体现不出来。最终从图上可以看出，总时间随着数据规模的增长呈近似线性的增长。

当数据量达到某些临界值，每次操作所花费的时间略有增长。一方面，这是由于B+树中节点数增加引起的开销，但因为访问节点的次数没有显著增长（从0到48），而索引存在内存里，层数增加带来的访问消耗几乎可以忽略不计，所以效果不显著。

若是将索引存到磁盘上，每增加一层就会带来对磁盘的一次额外访问，而磁盘访问速度明显比内存要慢，增长应该会比较明显。

另一方面，当数据库足够大时，定期对数据库进行刷新，清楚冗余信息，也会带来开销。

B+树的优势在于，当树存在磁盘上时，比其他数据结构对磁盘的访问次数少很多，且树的结构非常稳定。因此，若数据量很大，索引文件不能存在内存里时，使用B+树的效果就体现出来了。

和二叉树相比，B+树进行的48次字符串比较，二叉树可以存储248=2.8\*1014条数据，远大于degree为16的B+树能存储的11400000条数据。所以，如果索引全部存在内存里，还是红黑树效率高一些。