简易key-value型

Database 设计文档

516030910175 李珊

2017.07

目录

[⼀、基本信息 2](#_Toc489040578)

[二、提供接口 3](#_Toc489040579)

[三、模块实现 7](#_Toc489040580)

[四、功能实现及流程图（均指原始接⼝） 8](#_Toc489040581)

[五、特点 13](#_Toc489040582)

[-空间重利用 13](#_Toc489040583)

[-缓存设计 14](#_Toc489040584)

[-一致性刷新 15](#_Toc489040585)

[-索引文件与重利用空间文件一致性刷新 16](#_Toc489040586)

[六、测试及分析 17](#_Toc489040587)

[七、待改进 26](#_Toc489040588)

⼀、基本信息

* 项目名称：简易key-value型数据库 –
* 主要数据结构：B+树 –
* 开发环境：Visual studio 14.0 –
* 测试环境：Windows
* 处理器：Inter® Core™ i5-6200U CPU @ 2.30GHz 2.40GHz
* 内存：8 GB
* 概述：实现<key, value>型数据的增添、删除、修改、查找功能。

二、提供接口

提供<int,int>类型的存储。限制了<key,value>都是八位数的int。

**-打开**

void open()

如果数据库存在，就打开，不存在就创建，读取索引文件建立B+树，读取可重利用的地址索引文件建立vector。

void TestOpen()

第二个接口用于测试用的新建数据库，方便比对操作的进行。

**-关闭**

void close()

刷新索引文件和重利用地址文件,刷新数据文件，刷新缓存。

**-数据合法性检测**

bool valid(int key, int value)

-判断输入的数据是否符合要求

- key：待存储的关键词；value：待存储的值

-要求数据的长度都是八位数（一千万除外）。

**-缓存检查**

int InCache(int key)

-检查接下来要操作的这个关键词是否在缓存中，如果存在返回在缓存中的位置，不存在返回-1

- key：待操作的关键词

**-存储**

bool buffer\_store(int key, int value)

bool o\_store(int key, int value);

void store(int key, int value)；

- 提供三种接口：缓存接口与数据文件接口和⽤户⽤接⼝，用户用接口会直接输出提示信息如 “存储成功!”或提示数据库中已有该词条等

- key：待存储的关键词；value：待存储的值

– 存储该<key,value>词条

- 不支持重复存储，即若数据库中已保存此key相关数据，此次存储操作将不对数据库和缓存进行任何修改（若使⽤用户接口，则会输出提示信息） -

**- 删除**

bool o\_remove(int key);

void remove(int key);

- 提供两种接口：测试⽤接口与⽤户⽤接口，用户⽤接口会直接输出提示信息如 “删除成功”或提示数据库中无该词条等

- key：待删除的关键词

- 删除该key存储的value

- 若原数据库中不存在该关键词数据，此次删除操作将不对数据库进行任何修改（若使用户接口，则会输出提示信息） -

**- 修改**

bool o\_modify(int key, int new\_value);

void modify(int key, int new\_value) ;

- 提供两种接口：测试用接口与用户用接口，用户用接口会直接输出提示信息如 “修改成功”或提示数据库中无该词条等

- key：待修改的关键词；new\_value：该关键词对应的新值

- 若原数据中不存在该关键词数据，此次修改操作将不对数据库和缓存进行任何修改（若使用用户接口，则会输出提示信息）

**- 查找**

int o\_search(int key)；

void search(int key)；

bool searchExam(int key, int CorrectValue)

- 提供三种接口：原始接口，测试用接口与用户用接口。

- 若原数据中不存在该关键词数据，原始查询操作将返回0，若存在，则返回value；测试接口中返回查询到的值和正确值对比结果；用户接口，则会输出提示信息;

- key：待查找的关键词

- CorrectValue：正确的值；

- 查找该关键词的值

三、模块实现

- handle.h

接口实现，提供了上述接⼝

- BPlusNode.cpp

B+树的节点类，对节点包含的信息进行封装

- Tree.cpp

数据库的主体部分，B+树索引文件类，实现对索引文件的操作和B+树的操作封装（对B+树的操作即对索引文件进行相应操作）

-CorrectTest.h

正确性测试

-PerformTest.h

性能测试

四、功能实现及流程图（均指原始接⼝）

**- 打开**

- 尝试打开索引，数据文件，重利用地址文件，若本不存在，则创建⽂件

- 读取索引文件，建⽴B+树

- 读取重利用地址文件，建立vector

**- 关闭**

- 刷新所有数据文件，关闭⽂件

**- 存储**

- 查找缓存buffer，若存在该关键词，终⽌存储，直接返回

- 否则，查找B+树，若存在该关键词，终⽌存储，直接返回

- 否则，以<key,value>的vector形式存入二维向量的缓存，当缓存中达到限值定的时候，将缓存中的数据存入数据文件，并清空缓存vector。

- 存入数据文件的时候，先看有没有可以重利用的空间。如果有就用，没有就加在文件末尾。

**- 删除**

- 查找数据读缓存buffer;若存在该关键词，删除

- 如果缓存中没有，再查找B+树，若不存在该关键词，终⽌删除，直接返回

- 删除该关键词，空间回收；

**- 修改**

-查找缓存，若存在该关键词，直接修改。

- 若不存在，查找B+树，若不存在该关键词，终⽌修改，直接返回

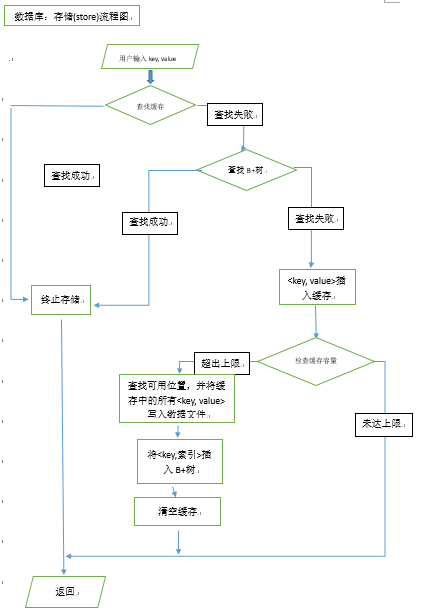
- 否则，利⽤用查找结果修改索引文件中的该关键词的位置，覆盖写入

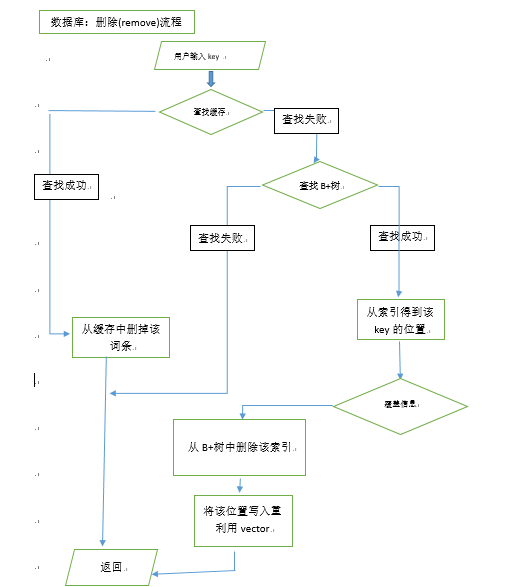
**- 查找**

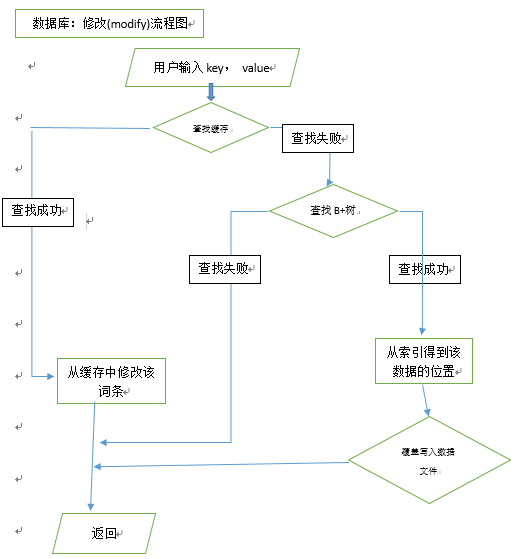
- 查找数据读缓存buffer，若存在该关键词，返回其对应的值

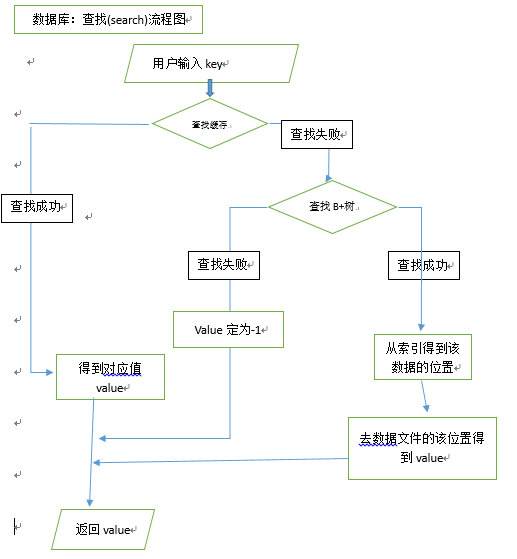
- 否则， 查找B+树，若存在该关键词，得到其在数据文件中的值，返回该值

- 若不存在，返回0；









五、特点

-空间重利用

-主要数据结构：

Vector<int>DeleteLocation;

-主要文件：

“DeleteIdx.txt”

-主要方法：

用Deleteidx.txt储存可以利用的地址，存入vector中，每次删除从vector中添加一个地址，每次插入就插到vector的最后一个地址，然后把最后一个删掉。关闭的时候重写Deleteidx.txt文件，将vector中剩下的地址写进去。

下一次开的时候还可以用。

-实现功能：

管理数据文件中可用空间信息

因本数据库中每一个数据所占据的空间是固定的，所以只要记录起始位置。

- 被调⽤用环境及作⽤用：

打开：初始化，载入之前剩下的空间信息进入DeleteLocation。

删除：添加空间信息进入DeleteLocation；

存储：先考虑DeleteLocation里面的空间，用掉一个删掉一个。

关闭：刷新重利用空间的文件信息。

**-缓存设计**

-这里的缓存是一个临时存储区。

- 主要数据结构：

vector<vector<int>> buffer;

- 实现功能：

- 利用数据使用的局部性，将近期存储的<key,value>保存。增删查改操作都可以直接对缓存操作。

- 维护缓存容量，如果缓存超过容量，就把缓存中的数据全部写入到数据文件，然后清空缓存。

-被调用环境及作用：

-存储：需要先确认该<key,value>不在缓存中也不在B+树中，否则直接终止存储。若原来不存在，则先存储到缓存中，当缓存超过一定值，就把缓存中的数据全部写入到数据文件，然后清空缓存。

-查找：在进行关键词的查找时，若在缓存中，可直接读取。免去B+树的查找。

-修改：在进行修改时，如果关键词在缓存中，此时该关键词还未写入文件，所以可以直接在缓存中修改数据，避免对文件的操作。

-删除：在进行删除时，如果关键词在缓存中，直接从缓存中删除数据，终止删除。避免去B+树查找和对索引文件和数据文件的操作。

**-一致性刷新**

- 主要数据结构：

vector<vector<int>> buffer;

-实现功能：

-提供自动和强制两种模式：前者是在写缓存容量满以后对存储数据文件和索引文件进行刷新。后者是在用户想要退出程序不再操作时，调用close(); 刷新缓存，数据文件，重写索引文件和可重利用的空间文件。

-被调用环境及作用：

-在存储(store)时，会将相关的数据写入缓存，当缓存达到一定数量时，写入文件。

-在停止程序时，将文件刷新，以便下一次使用。

-参数分析：

-见性能分析

**-索引文件与重利用空间文件一致性刷新**

-主要数据结构：

-BPlusTree tree;

-Vector<int>DeleteLocation

-实现功能：

在删除和存储操作时，都是对B+树进行操作，要查找或修改时，直接通过B+树里的信息可以找到数据。只有在用户打算结束程序或刷新的时候才会刷新索引文件和重利用空间文件。通过把B+树和vector中的信息重新写入文件来更新。

-被调用环境及作用：

close() 刷新文件

六、测试及分析

**1. 正确性测试**

1.1大量数据测试(correctness\_test.cpp)：

- 建立数据库，分别执⾏行下列操作：

- 随机存储TIMES条数据 - 随机删除部分数据

- 读取全部数据（包括删除后的数据），⽐对是否与插入相同

- 随机修改数据

- 读取全部数据，进⾏⽐对

- 随机存储TIMES条数据

- 读取全部数据，进⾏⽐对 - 该测试主要涉及大数据量下数据库的正确性

- 在TIMES分别取100、1000、10000、100000、1000000时，三次读取操作，全部正确

1.2 非寻常测试(abnormal.cpp)：

- 建立数据库，分别执行下列操作：

- 随机存储10000条数据

- 循环以下步骤多次：

- 存储⼀之前不存在的关键字及值<key,value>，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

- 存储该关键字及一个不同的值，读取该关键字

- 该测试涉及读取、删除、修改不存在于数据库中的可能造成异常的操作 - 每次循环中，都按照“原value 、-1、-1、-1、新value”的顺序返回读取内容， 表明测试正确

1.3 综合测试(round\_test.cpp)：

- 建立数据库，执行老师提供ppt上部分操作，先插入nrec数据，后循环以下步骤多次：

- 随机读一条记录。

- 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读取这条记录。

- 每循环17次，随机替换一条记录为新记录。

- 该测试涉及综合存储、删除、修改、读取操作 - 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返回 -1， 表明测试正确。

2. 性能测试

2.1 空间性能

空间重利用对文件大小影响(space\_test.cpp)

- 本次测试执行操作为：插入20万数据，删除7万数据，再插入20万数据。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 无空间重利用 | 空间重利用 | 差占重利用百分比 |
| 数据文件 | 7.2 | 6.0 | 20% |

单位（M）

- 结果分析：

-由于格式都是确定的每条数据长度也都是确定的，为19.

- 不使用空间重利用后，再插入的数据只可增加到数据文件末尾。

可估算数据文件大小为： (200000 + 200000) \* 19 / 1024 / 1024 = 7.2 M

- 而使用空间重利用后，再插入的数据可使用之前的空间。可估算数据文件大小为： (200000 - 70000 + 200000) \* 19 / 1024 / 1024 = 6.0 M

- 结果符合预期。

- 在删除数据量越多、数据量越大的情况下，空间重利用对文件大小影响越大，这不难理解。

2.2 时间性能

2.2.1 缓存(模仿用户行为)对操作时间影响(TimeTest)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 无缓存 | 有缓存 | 差占有缓存百分比 |
| 1000次操作 | 1．697 | 1．586 | 6．99% |
| 10000次操作 | 18．474 | 15．827 | 16．7% |
| 100000次操作 | 195．22 | 159．69 | 22．25% |

- 本次测试操作模仿⽤用户操作，执⾏行以下循环N次：

-随机插入10条记录：

随机查找刚插入的这条记录；

有时（随机）修改这条记录；

有时 （随机）删除这条记录

-该操作模仿了用户插⼊入数据后，查看某条数据是否插入正确，进行修改或删除的操作

- 结果分析： - 不使用读缓存时，在查找、删除、修改记录时，都需要重新搜索树，到数据文件中 抓取相应信息。

- 在插入后不久，进行查找时，使用读缓存，效率明显提升。实现该缓存主要也是从用户使用习惯上考虑，该缓存很适应用户的习惯操作，因此具有普遍意义。

2．2．2 缓存对只有存储操作的影响

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量/万 | 无缓存 | 有缓存 | 差值占百分比 |
| 0．1 | 0．174 | 0．193 | -10.0% |
| 1 | 1．376 | 1．525 | -9．78% |
| 10 | 13．132 | 15．387 | -14.34% |

-本次测试比较有无缓存对插入操作时间的影响。测试操作为：随机插入⼀定量数据，⽐较插入时间.

-结果分析：

在脱离用户操作习惯的情况下，有缓存比无缓存的存储速度还要慢。缓存设计得不是很好。本数据库的缓存设计是一个临时存储类的，该插入的数据还是会按照相同的步骤插入，但是如果有缓存，在插入前还要先在缓存区存放一下，增加了对缓存区的操作。其他的方面并没有改变。只有在用户操作习惯的情况下，这个缓存才是合理的。

2.2.3 操作时间与数据量关系

-节点大小固定的情况下，数据量对各个操作单次耗时影响：

Nodesize = 256单次操作耗时/μs

Nodesize = 256单次操作耗时/μs

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量/万 | 0．1 | 1 | 10 |
| 存储 | 189 | 154．3 | 141．63 |
| 查询 | 87 | 79．8 | 79．6 |
| 修改 | 69 | 58．7 | 55．44 |
| 删除 | 102 | 76．9 | 48．85 |

Nodesize = 512单词操作耗时/μs

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量/万 | 0．1 | 1 | 10 |
| 存储 | 309 | 189．9 | 162．65 |
| 查询 | 91 | 81．5 | 83.77 |
| 修改 | 72 | 53.3 | 56.52 |
| 删除 | 212 | 134.4 | 58.48 |

Nodesize = 1024单次操作耗时/μs

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量/万 | 0．1 | 1 | 10 |
| 存储 | 259 | 196．4 | 181．25 |
| 查询 | 111 | 86．5 | 81．65 |
| 修改 | 70 | 52．6 | 54．28 |
| 删除 | 177 | 134．8 | 66．13 |

分析：

数据量越大单次操作耗时越小，应该是因为在这些操作中，都有占相当一部分时间拿来重写索引文件了。重写索引文件时，打开索引文件写入的时间被越多的数据量分摊就会把平均耗时降低了。

2.2.4 操作时间与节点大小关系

1. 存储STORE

存储单次操作耗时对比/μs

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量（万） | 0．1 | 1 | 10 |
| 256 | 189 | 154.3 | 141.63 |
| 512 | 309 | 189.9 | 162.65 |
| 1024 | 259 | 196.4 | 181.25 |

2.查找

查找单次操作耗时对比/μs

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量（万） | 0．1 | 1 | 10 |
| 256 | 87 | 91 | 111 |
| 512 | 79.8 | 81.5 | 86.5 |
| 1024 | 79.6 | 83.77 | 81.65 |

3．修改

修改单次操作耗时对比/μs

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量（万） | 0．1 | 1 | 10 |
| 256 | 69 | 58.7 | 55.44 |
| 512 | 72 | 53.3 | 56.52 |
| 1024 | 70 | 52.6 | 54.28 |

修改操作中，单次操作耗费时间相近，测量上也许存在误差。

4．删除

删除单次操作耗时对比/μs

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量（万） | 0．1 | 1 | 10 |
| 256 | 102 | 76.9 | 48.85 |
| 512 | 212 | 134.4 | 58.48 |
| 1024 | 177 | 134.8 | 66.13 |

可能是因为测量的节点都是比较大的大节点，所以差异性体现不出来。

七、待改进

-数据库可存储的数据形式内容单一，限制太大。只能存储固定长度的<int,int>类型的数据。

-缓存只有在用户使用习惯下才能够发挥作用，提高效率。否则还会降低效率。

-B+树的实现未完善，高度目前只能达到三层。要存储大量数据，需要把分支数设置得很大。分支数太小了的时候测不了太多的数据。如果数据太大而没有及时修改分支数就会报错。