简单KEY-VAIUE数据库

设计文档

516015910018

魏小渺

0. 目录

1. 项目概况

项目名称

简单KEY-VALUE数据库

引索结构

B+ Tree

功能摘要

利用B+树这一核心数据结构实现了一个简易的<int-string>类型数据库。支持全体int范围作为键，不定长度string作为值。外加对数据文件的管理类Data实现了对数据库内插入数据，删除数据，修改数据，读取数据和范围查找这些基本功能。另外还集成了Cache，Buffer，Holder这些用于性能优化的附加结构。

开发环境

Xcode Version 9.3.1

测试环境

- 机型：MacBook pro
- 处理器：2.7 GHz Intel Core i5
- 内存： 8 GB 1867 MHz DDR3
- 硬盘： APPLE SSD SM0128G
- 系统： macOS High Sierra vision 10.13.5

2. 用户接口

针对数据库的基本功能，本数据库以类(class DB)的方式提供了原始接口和IO集成接口。其中原始接口完成较为单一的逻辑任务，IO接口则由单个或多个原始接口和命令行输入输出流组成。

2.1 创建项目

```
void DB:: create(string projName);
void DB:: IOcreate();
```

该接口会分别调用B+树引索类Tree和数据文件管理类Data的创建文件接口，分别在硬盘上生成名称为{projName}.idx和{projName}.dat文件，准备写入。将见4.基本功能-4.1。

IOcreate()集成了cin/cout流，以IO的方式调用create(string projName)，创建项目。

2.2 打开原有项目

```
bool DB:: open(string projName);
void DB:: IOopen();
```

该接口会分别调用B+树类Tree和数据文件管理类Data的打开文件接口。若存才对应名称的dat和idx文件，则从kpt文件内读入上次关闭该项目时储存的信息(总节点数，根节点位置，数据总数)，并初始化B+树引索和数据文件类。将见4.基本功能-4.1。

IOopen()在打开不存在项目时询问是否创建新项目。

2.3 插入数据

```
void DB:: insert(int key, string value);
void DB:: IOinsert();
```

该接口分别调用了Tree和Data内部的插入函数，集成了Holder(data文件空间管理类)和Cache(数据读取缓存类)，对data文件进行了空间性能的优化，同时Cache类完成了数据查找的时间优化。具体实现详见 4.基本功能 - 4.2插入数据。IO接口检查数据库中是否存在对应键值。

2.4 查找数据

```
Pair DB:: get(int key);
void DB:: IOget();
```

该接口分别调用Tree和Data类的查找方法，集成Cache类进行热点时间查找优化。由key在B+树种查找到对应Pair(储存对应value)在data文件中的序列化位置。从而再在Data类中读取Pair。具体实现详见4.基本功能-4.3查找数据。若不存在则返回空Pair。

2.5 范围查找

```
vector<Pair> DB:: rangeGet(int k1, int k2);
void DB:: IOrangeGet();
```

要求输入int k1，k2且k1<k2。通过迭代的方法调用Tree和Data内的查找方法，返回数据库中所用键值在[k1,k2]的Pair向量。

2.6 删除数据

```
int DB:: remove(int key);
void DB:: IOremove();
```

该接口调用Data和Tree内部删除key和删除Pair的接口，集成Holder，Cache，进行空间优化和Cache结构调整。其中Data内部集成的Buffer也会更新删除节点信息，优化数据文件读写过程。若数据库中不存在则返回-1；否则返回被删除数据序列化位置。详见4-4.4删除数据。

2.7 修改数据

```
int DB:: remove(int pos, int key, string value);
void DB:: IOremove();
```

该接口通过Data的修改方法更改data文件pos位置Pair对应value。同时维护Cache类更新。IO接口还掉用了get(int key)接口判断数据库中是否存在对应键值。详见4-4.5修改数据

2.7 保存数据

```
void DB:: save();
```

关闭数据库之前的数据保存。分别调用Data和Tree的关闭函数。Data类flush数据写入缓存Buffer，保存数据序列化位置；Tree保存根节点在引索文件的位置和总结点数，用于再次初始化数据库。

2.7 创建大规模数据

```
void DB:: BigSize(int size, string value);
```

用于在打开数据库前自动创建名为testDB的包含size个数据的大规模数据库，每条数据的value均为给定值，键值为[1,size]，并对数据不加以存在检查地写入。创建完成后可对数据进行以上操作。

3. 代码模块

本数据库使用面向对象的设计思想。将数据库视为DB()类。数据库类又主要包括了对数据文件的读写和对引索文件的读写。针对这两项主要功能，又分别设计了Data()类和Tree()类分别管理数据文件的读写和利用index文件维护B+树，实现key到数据序列化位置的查找。

同时，用于提高查找时间性能的Cache类，用于提高文件写入时间性能的Buffer类和用于提高数据文件空间性能的Holder类也均由DB类集成封装。最终将优化后的用户接口作为DB的公开方法提供用户使用。

3.1 database.cpp (database.hpp)

数据库DB类所在文件。封装各个辅助类，并在该层面编写了2中的用户接口。

3.2 bptree.cpp (bptree.hpp)

包含B+树节点类和B+树类。

B+树节点持有该节点在index中的序列化位置，是否叶子节点，子节点数目，存有的键值数组和子节点位置数组(若为叶子节点，则为对应数据位置)的信息。

利用B+树这一核心数据结构，实现了创建、打开、保存index文件；由键值搜索数据序列化位置；插入新的键值；删除原有键值的；修改某一键值的一系列操作。

3.3 datafile.cpp (datafile.hpp)

包含键值对Pair类和文件管理Data类。

Pair类持有了某条数据的键(int key)和值(char contains[size])的信息。同时还记录了自己在data文件中的序列化位置(通过Pair的构造函数内的counter保持)。对于超过size的字符值，Pair还持有向下读取扩容Pair的信息(int readNext)来实现可变长value。

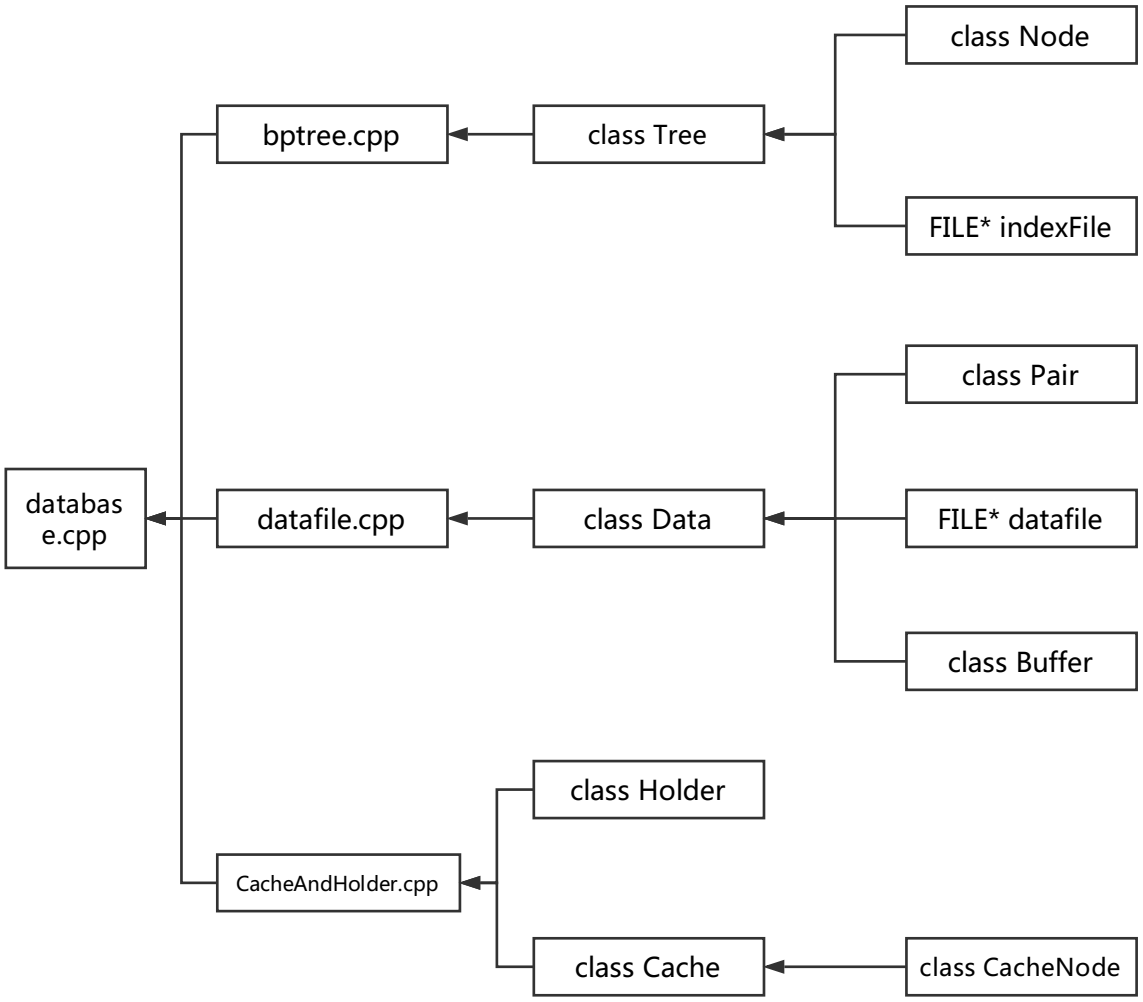
Data类以Pair为单位读写datafile，实现对数据的操作。同时持有辅助类vector<Pair> Buffer，优化数据写入。Buffer的设计详见5.附加结构-5.1Buffer。

3.4 CacheAndHolder.cpp (hpp)

包含数据查找时间优化类Cache和数据文件空间优化类Holder。

设计详见5.附加结构-5.1Cache和5.3Holder。

3.5 各个模块组织关系


class Node
bptree.cpp
class Tree
FILE* indexFile
class Pair
databas
e.cpp
datafile.cpp
class Data
FILE* datafile
class Buffer
class Holder
CacheAndHolder.cpp
class Cache
class CacheNode

4.基本功能

4.1 创建/打开/保存数据库

1 创建数据库：在database.out同一目录创建指定名称项目，生成.dat和.idx文件。若已存在同名项目，则覆盖原项目。

2 打开数据库：输入路径+项目名称。无路径名称则在database.out同一目录寻找文件。并由.dat文件.idx文件和.kpt文件初始化数据库。若未找到文件，则判断是否创建同名数据库。

3 保存数据库：flush Data类的Buffer，fclose(FILE* indexfile)，将数据和引索写入磁盘。同时生成.kpt文件保存当前数据总数，B+树节点总数和B+树根节点在引索文件中的序列化位置，供再次打开本数据库时初始化使用。

4.2 插入数据

1. 得到用户输入<key-value>。

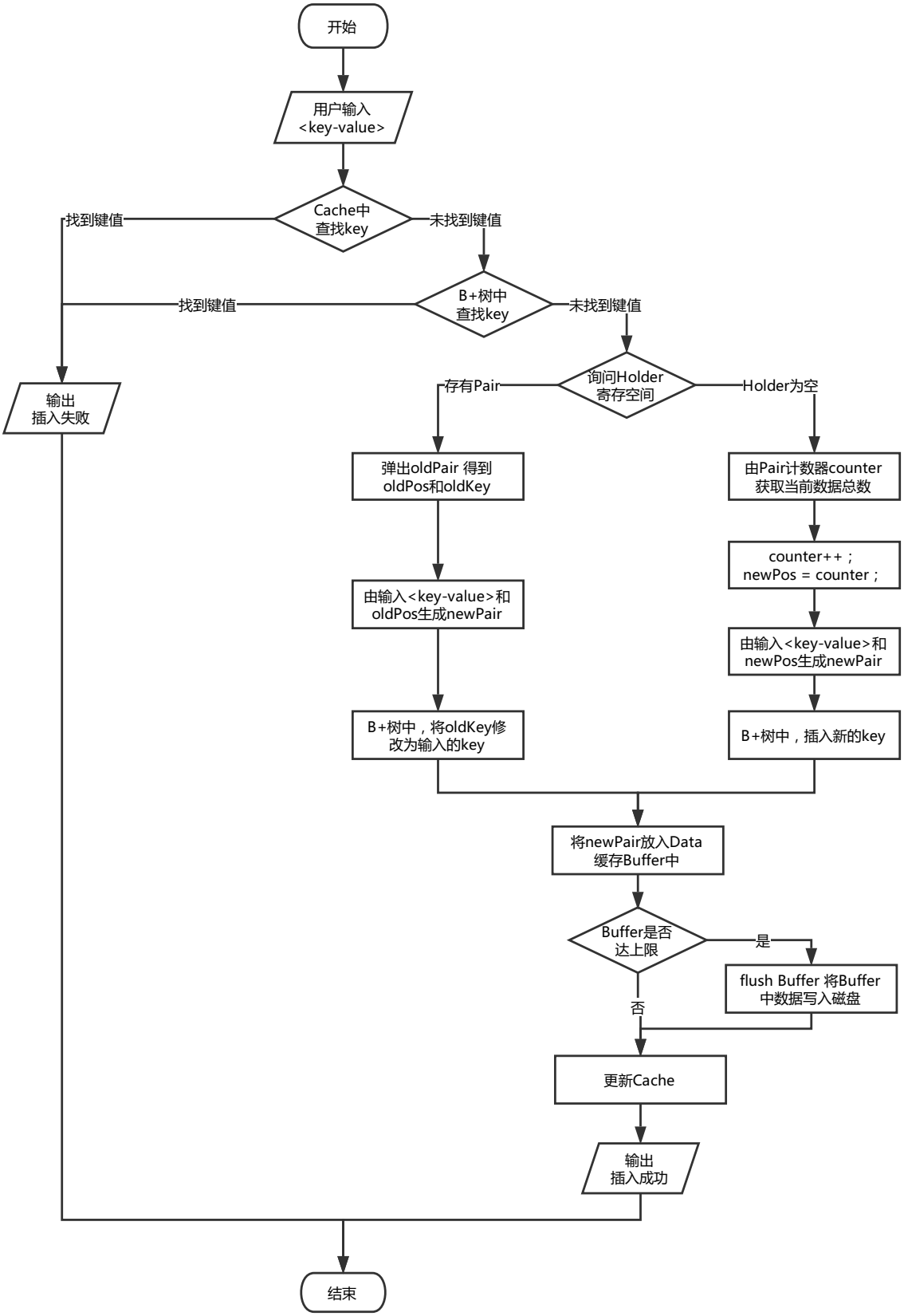
2. 先后在Cache和B+树中分别搜索键值，一旦找到已存在相同键值则停止操作并结束，否则继续插入操作。

3. 询问Holder是否寄存有剩余空间；若Holder中存有被删除数据Pair信息，则在数据文件中的缓存Buffer中放入修改过键值的同一位置Pair，B+树中修改oldKey为key。

4. 若Holder为空，则由Pair中的counter获取数据总数，Buffer中放入counter下一位置new Pair，B+树中插入新键值。

5. 在Cache顶部更新新插入数据信息，结束。

插入数据流程图：


开始
用户输入
<key-value>
Cache中
查找key
找到键值
未找到键值
B+树中
查找key
找到键值
未找到键值
输出
插入失败
询问Holder
寄存空间
存有Pair
Holder为空
弹出oldPair 得到
oldPos和oldKey
由输入<key-value>和
oldPos生成newPair
B+树中，将oldKey修
改为输入的key
由Pair计数器counter
获取当前数据总数
counter++ ;
newPos = counter ;
由输入<key-value>和
newPos生成newPair
B+树中，插入新的key
将newPair放入Data
缓存Buffer中
Buffer是否
达上限
是
flush Buffer 将Buffer
中数据写入磁盘
否
更新Cache
输出
插入成功
结束

4.3 查找数据

1. 得到用户输入key。

2. 先在Cache搜索key，若找到则输出对应value，结束。否则在B+树中查找key。

3. 若B+树中查找失败，则输出为未收录的键值，结束。否则可由B+树得到该条数据在dat文件中的序列化位置target Pos。

4. 在Data的Buffer中搜索pos为target Pos的Pair，若找到该Pair，则返回其value。否则在datafile中读取target Pos，返回其value。

5. 在Cache顶部更新新插入数据信息，结束。

4.4 删除数据

1. 得到用户输入key。

2. 在Cache中查找对应数据，若存在则删除。

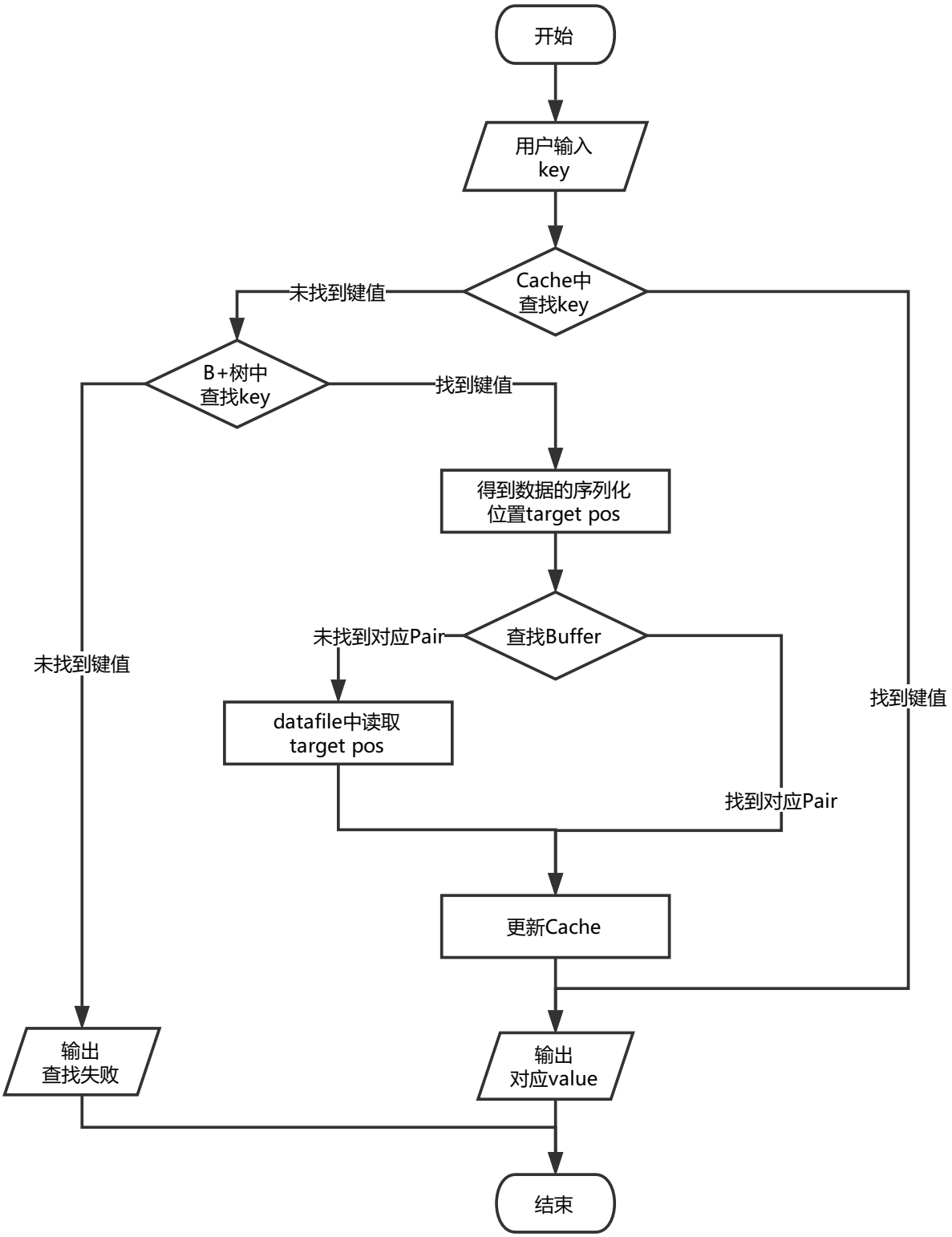
3. 在B+树中查找键值，若未找到，则输出为未收录的键值，结束。否则删除对应键值，并返回数据的序列化位置。

4. 在Holder类中保持被删除Pair类的复制，包括其序列化位置。

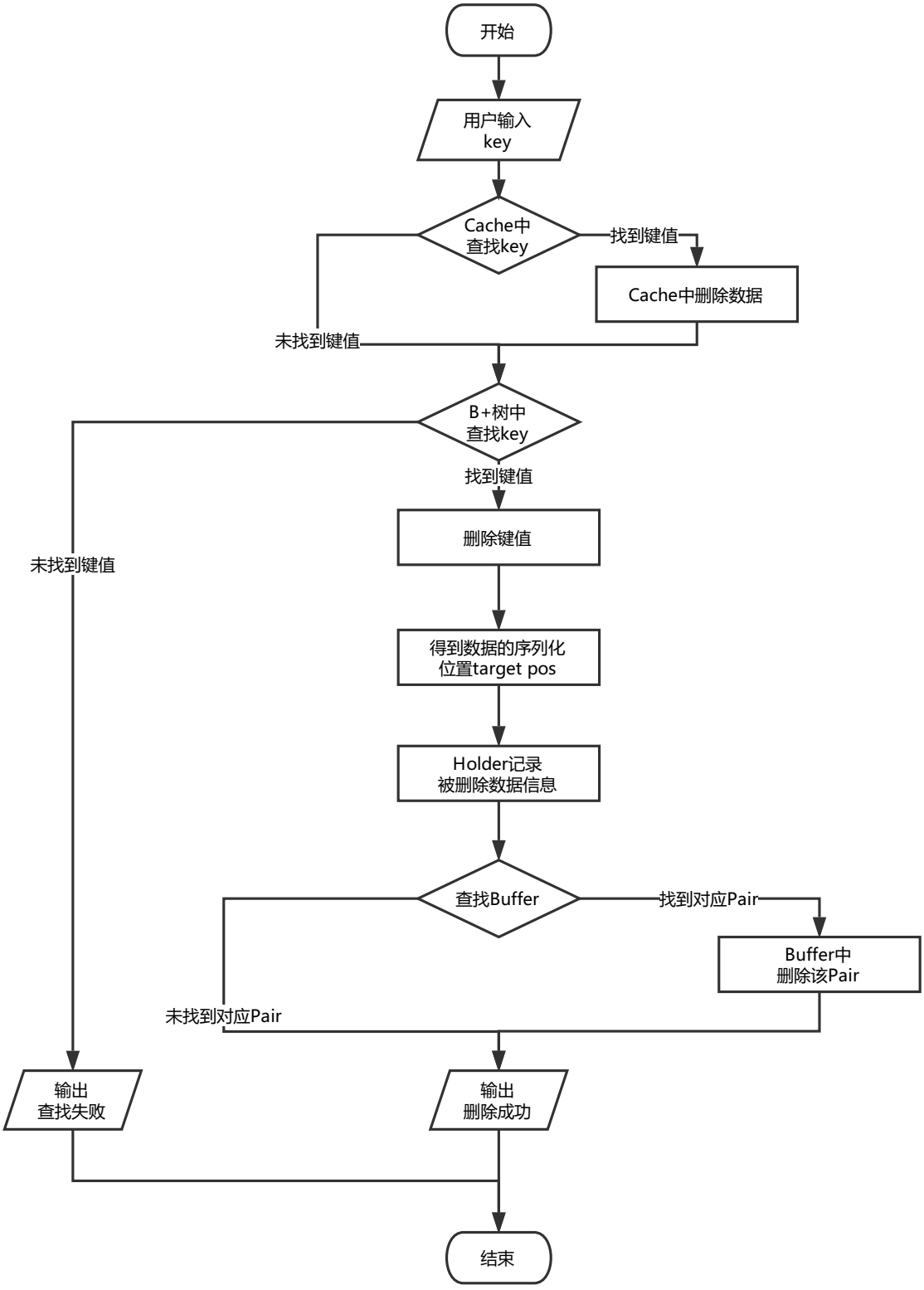
5. 在Data的Buffer中查找对应数据，若找到则删除。

6*.删除造作不会改写data文件，被删除数据位置由Holder保持并会被覆盖写入。

查找数据流程图：


开始
用户输入
key
Cache中
查找key
未找到键值
找到键值
B+树中
查找key
找到键值
未找到键值
得到数据的序列化
位置target pos
查找Buffer
未找到对应Pair
找到对应Pair
datafile中读取
target pos
更新Cache
输出
查找失败
输出
对应value
结束

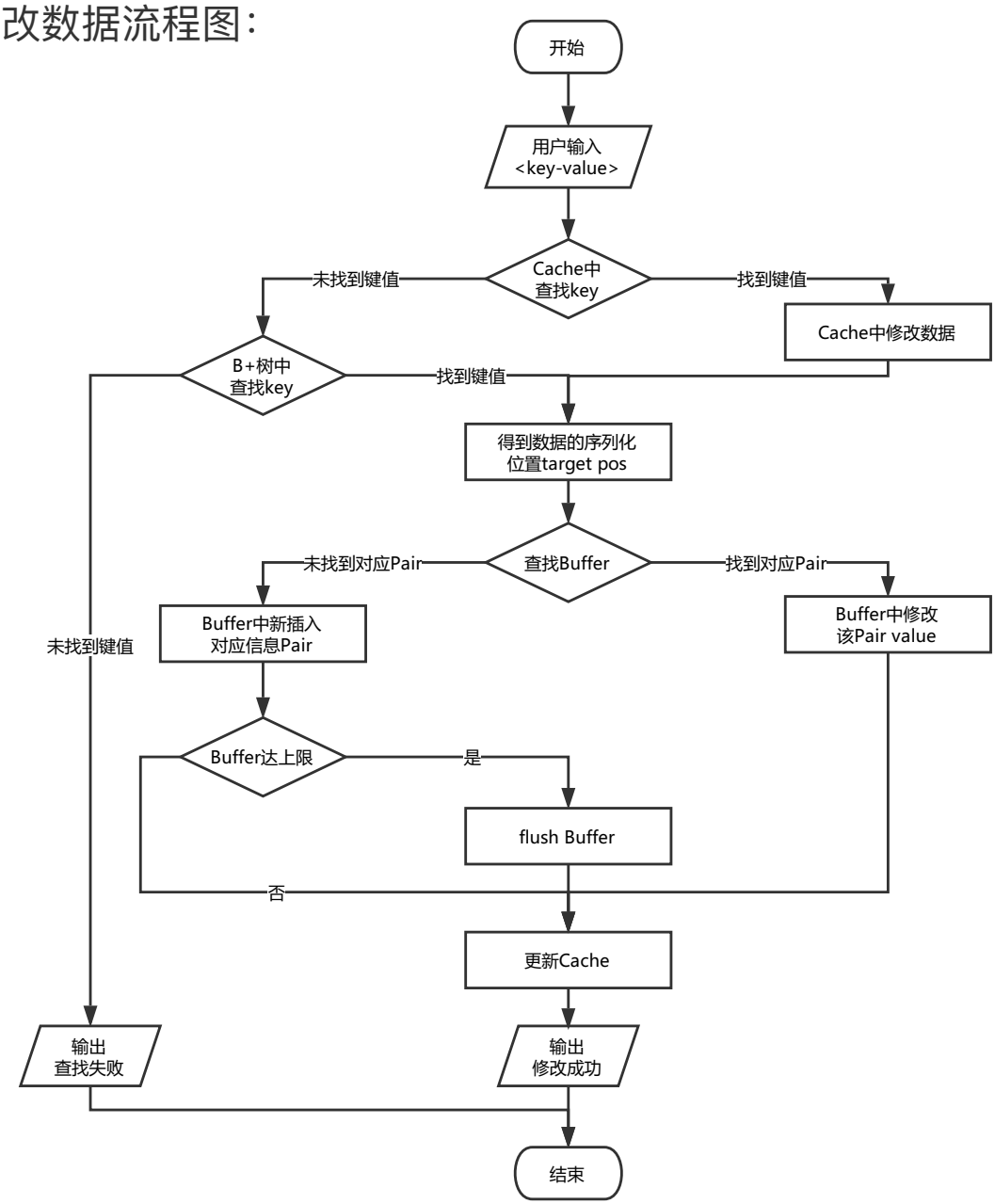
删除数据流程图：


开始
用户输入
key
Cache中
查找key
找到键值
Cache中删除数据
未找到键值
B+树中
查找key
找到键值
删除键值
未找到键值
得到数据的序列化
位置target pos
Holder记录
被删除数据信息
查找Buffer
找到对应Pair
Buffer中
删除该Pair
未找到对应Pair
输出
查找失败
输出
删除成功
结束

4.5 修改数据

1. 得到用户输入<key-value>。
2. 分别在Cache中和B+Tree中查找，若未找到则结束。
3. 若找到键值，则读取其数据序列化位置。
4. 在Buffer查找并修改。若未找到则在Buffer内插入新的Pair。
5. 检查Buffer是否达上限，若是则flush。
6. 更新Cache信息。

修改数据流程图：



5.附加结构

5.1 Cache（查找数据时间优化类）

Cache的主要结构为一双向链表，链表的每个节点持有key-value数据信息。考虑到访问热点数据时需要不断地刷新Cache头部，让热点数据的访问耗时更少；同时，为了防止数据库变为内存数据库，Cache容量不宜过大。故选用双向链表这一结构进行小范围搜索，使得每次刷新的复杂度为O(1).

Cache的接口包括：

```
void Cache:: put(int key, string v);
```

该接口会向Cache头部新插入带有<key-value>信息的CacheNode并在Cache容量达上限后自动删除链表尾部节点。

在更新Cache时，若Cache中无操作数据，则调用该接口。具体为数据库插入数据、查找数据和修改数据时调用。

```
string Cache:: search(int key);
```

该接口从链表头部搜索，查找给定key所对应的value并返回。若查找成功，则自动将该结点移动到列表头部。

在一切需要搜索B+树的操作前，都会调用该接口进行时间优化。

```
void Cache:: remove(int key);
```

该接口从链表头部搜索，查找给定key，找到后删除该节点并将Cache长度减一。

5.2 Buffer （读写数据文件时间优化类）

Buffer集成在Data类中，主要结构为vector<Pair>，持有一个最大容量信息。当Buffer储存达到上限时，将Buffer内的数据一起写入文件，进行了文件写入的时间优化。进行读数据文件操作前，都会先检查Buffer，进行时间优化。

Buffer的函数包括：

```
void Data:: putInBf(Pair a);
```

该接口向Buffer内插入一个信息完全的Pair，等待被写入文件。并自动检查Buffer内Pair个数。若达到上限则调用flush()。

```
void Data:: flush();
```

当Buffer容量达到上限时，flush将直接调用fseek(), fwrite()函数，将缓存内部Pair写入数据文件。并在完成后调用fflush()函数确保数据被写入磁盘。

flush()被调用时当且仅当Buffer插入新数据达到上限。除此之外任何操作不会调用该接口。因为在对数据库进行各种操作时，同时维护Buffer，有效的数据信息只会被datafile和Buffer其一持有。

```
Pair Data:: searchInBf(int pos);
```

该接口根据给定的序列化位置信息在Buffer里查找对应Pair并返回。在数据库查找数据，修改数据和删除数据时，searchInBf()用来在Buffer里找到待操作文件。

5.3 Holder （数据文件空间优化类）

Holder主要结构为queue<Pair>存放被删除后的Pair。持有其在数据文件内的位置信息。可在新数据插入时覆盖原有信息，避免数据文件过大。Holder的接口设置类似于stl中的queue，不再赘述。

5.4 一致性证明

Cache和Holder作为Database的私有成员类，在高层次对程序进行优化；Buffer作为Data的私有类，直接面对底层的FILE* datafile。加之index文件和data文件，数据的存储和访问源在三个层次上达到了5个，那么数据的一致性可以保证吗？答案是肯定的。

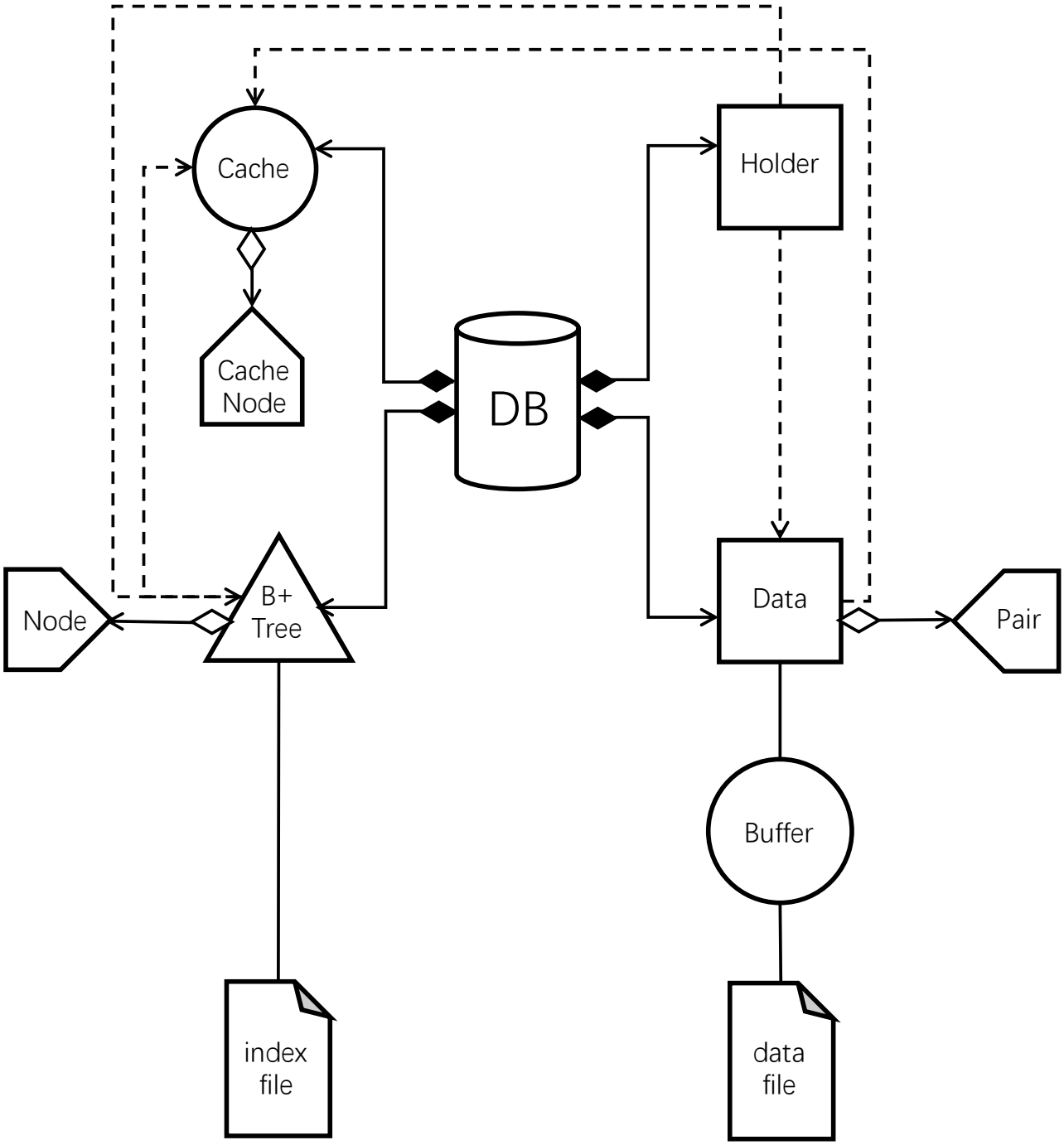
对于Cache类，Cache对数据的添加和删除(统称为更新Cache)都是建立在Data类和B+树信息成功更新后的。所以可以保证Cache内持有的数据信息为最新信息且完全包含于B+树和Data持有信息。

对于Holder类，空间的持有和释放对应于数据的删除和插入。只对数据在文件中的序列化位置进行控制，不影响一致性。

对于Buffer类，无论是读取还是插入数据，Buffer总优先于FILE* datafile，在flush前。可知Buffer内为最新的有效信息；flush之后Buffer被清空，其信息全被写入datafile，可知：Buffer内的信息与datafile信息交集为空，没有不一致数据的生成空间。Data类作为二者并集，持有最新的有效数据信息。

综上：所有信息引索 = B+树信息 = index file；所有数据信息 = Data类信息 = （Buffer信息 || datafile信息）⊇ Cache的信息。

5.5 完整程序结构


Cache
Holder
Cache
Node
DB
Node
B+
Tree
Data
Pair
Buffer
index
file
data
file

6. 功能特性

6.1 不定长value

不定长value的实现依赖于class Pair的设计。

Pair内部用一个定长字符数据char[SIZE]来保持value。对于长度小于等于SIZE的输入，Data类的insertPair函数会分配一个Pair储存数据 (长度不足的用'\0'不足方便读取datafile)。同时Pair类内部的计数器自增1，并以当前总数作为pos（除非显示的使用带有pos的构造函数，在Holder中会调用）。

对于长度大于SIZE的输入，insertPair会不加以询问Holder地连续地分配 [value.size() / SIZE] ([ ]为向上取整数，记为N，下同) 个Pair，这些Pair内的成员变量int readNext从第一个开始依次为 N-1, N-2, ……, 1, 0；其pos分别为counter+1, counter+2, ……counter+N。随后这些Pair会被放入Buffer或者Datafile中。

对于超长value对应的key，B+树只会插入key和首个Pair的pos，再次查询时，得到首个Pair后会检查其readNext，一次向后读取pos+1，pos+2…知道出现readNext为0的Pair。

删除操作同样会使Holder持有N和空位。对修改后为超长value的数据一律执行为删除后插入。

6.2 时间性能优化

程序的时间性能优化主要依赖于附加的Cache类和Buffer类。二者在查找引索前和读写文件前会进行各自所属的内存上的进行对应操作，避免了访问磁盘，时间性能得到优化。

在读取数据的过程中，在进行B+树内的搜索前首先会在Cache内进行搜索。若找到数据，则避免了在B+树内搜索的时间消耗。该优化也会发生在插入数据和修改数据前的键值检查过程中。

若程序已经进行了B+树搜索并得到目标数据在data文件中的序列化位置时后，由于Buffer缓存的设计，使得程序不会立即在data文件中读取数据，而是先在Buffer(内存中)进行目标数据搜索，若存在该数据则可以避免数据读取文件的耗时。

在写入数据到data文件的过程中，Buffer能够一次打开文件写入多条数据，并调用fflush()确保数据被写入磁盘。避免了分多次写入的时间消耗。

修改和删除数据在Buffer里先行，对于datafile来说可以进行操作合并。Buffer中同一键值得插入和删除在内存进行，flush时对datafile没有操作；Buffer中同一键值的插入和修改也在内存进行，flush时只对datafile进行一次插入操作。

6.3 空间性能优化

程序的空间性能优化主要依赖于Holder类。

未进行优化时，插入数据操作时需要利用Pair类内的计数器获取新插入节点产生的顺序信息，并以此信息作为该数据的序列化位置Pos。在B+树中叶子节点上插入这一对key-pos信息；再利用Data类在data文件末尾写入数据。删除数据时，只会在B+树上删除引索信息，Data类至多在Buffer内删除对应数据，不会对datafile进行改动。

插入和删除方式的缺陷导致程序暴露了两个问题。一是新数据不断在data文件后追加，删除过的数据没有真正在data文件上被删除，导致data文件会占据比有效信息更大的磁盘空间；二是引索文件和实际数据文件(并非Data类)描述数据有偏差，数据库一致性受到影响。

Holder类的设计可以在插入和删除前有效的利用和回收data文件占据的磁盘空间，使得该空间始终小于数据库有效数据的最大阈值。当Holder为空时，可以断言data文件内信息均为有效信息；当Holder不空时，可以断言data文件内信息与Holder内信息的差集为全体有效信息(均假设Buffer经过flush后)。数据信息的一致性得到了维护。

另外，利用Holder内的空间插入数据时，B+树调用Tree:: replaceKey()函数，index文件空间性能得到优化。Tree:: replaceKey()函数复杂度低于Tree:: insertKey()。插入数据的时间性能也有所提升。

7.测试程序设计

7.1 用于正确性的测试程

用于正确性的测试程序分为大量数据测试，非正常操作测试和综合操作测试。

7.1.1 大量数据的测试程序在创建空白数据库项目后，会执行以下操作：

1. 随机存储nrec条数据；
2. 数据删除部分数据；
3. 查找全部键值对，与1中插入数据比对；
4. 随机修改数据；
5. 查找全部键值对，与1中插入数据比对；
6. 随机储nrec条数据；
7. 查找全部键值对，与1中插入数据比对。

分别取nrec = 1 000，10 000，100 000，1 000 000，每次执行上述程序三次，并记录比对结果。当结果偏离预期时，抛出错误。

7.1.2 非正常操作的测试程序在创建空白数据库项目后，会执行以下操作：

1. 随机存储10 000条数据；
2. 插入一组已知数据；
3. 查找2中插入的数据；

4. 删除2中数据，再查找该数据；

5. 修改2中数据，再查找该数据；

6. 插入一组与2中数据同键不同值的数据，查找该数据；

要求以上操作返回原value，Pair(-1)，Pair(-1)，新value。并循环多次进行检查。偏离预期时抛出错误。

7.1.3 综合操作测试程序在向空白数据库写入nrec条数据后，会执行以下操作：

1. 随机查找一条数据；

2. 每循环37次，随机删除一条数据；

3. 每循环11次，随机插入一条数据并记录这条数据；

4. 每循环17次，书记修改修改一条数据，并分别使用等长的数据和更长的数据修改；

5. 循环1-4 nrec*5次后删除全部数据，每次删除同时随机查找10个数据；

要求1中查找值为插入值，对删除数据返回Pair(-1)代表不存在的数据。若偏离预期结果，则抛出错误。

7.2 用于时间性能分析的测试程序

7.2.1 数据库规模大小对单次插入查找时间的影响

本数据库采用B+树进行引索。在B+树节点大小固定的情况下，数据库现有数据的规模会影响到B+树层高产生影响，从而对自此基础上的单次操作时间产生影响。

为了更直白地反应引索结构对时间的影响，在测试中，Buffer和Cache的容量均设为1，且不进行重复访问。

程序进行以下操作：

1. 连续向空白数据库内按自然数顺序插入数据；

2. 当总数为nrec时，向数据库内插入1条数据，记录耗时；

3. 查找1中插入的数据，记录耗时；

4. 重复1-3过程，计算操作时间均值；

特别地，在nrec为0和100 0000时也会记录下次的插入时间。

多次运行程序，在数据总量nrec从0增长到1 000 000时，可以分别得到以上两种操作单次耗时均值。取定B+树节点大小为256，则每隔10 0000次记录插入一次，可以认为这一次插入的数据不新分割新的节点，不改变B+树层高。排除B+树调整时的时间影响。插入数据时间=B+树引索时间+index文件写入时间+data文件写入时间。

具体数据以及结果分析见8

7.2.2 B+树节点大小对操作时间的影响

当B+树节点容量为最小值2时，B+树可视为一棵十分难以维护的二叉树（要求叶节点高度一致且完全）。由于其层高过高，对于B+树的查找过程涉及多次引索的读取，降低了时间效率；反之B+树节点容量超过数据总量时，所有数据存放入根节点，B+树退化为根节点上的线性结构。读取内容过多，内存开销过大，失去了B+树设计的意义。 因此B+树节点的大小需要在二者效益之间权衡。

当B+树节点容量为SIZE时，进行以下操作：

1. 向空白数据库连续插入1000条数据，记录耗时；

2. 连续查询该1000条数据，记录耗时；

3. 连续修改该1000条数据，记录耗时；

4. 连续删除该1000条数据，记录耗时；

为了更直白地反应B+树节点大小对操作时间的影响，在测试中，Buffer和Cache的容量均设为1，且不进行重复访问。

在SIZE从6增长到1000时，所生成B+树的高度由9递减至0；B+树内的操作过程会得到简化，同时内存消耗增大。当SIZE达到1000时，操作将仅仅访问内存中的根节点，亦不存在初始化过程，所以无需读写引索文件，可定义此时的时间开销为最优时间开销。

具体数据以及权衡策略见8

7.2.3 Buffer容量对数据库读写数据时间的影响

在数据文件写入数据时，每次都单独地打开data文件并写入会增加额外的时间开销。Buffer将待写入的数据集中存放在内存中，在数据达到容量后统一写入，可以减少打开文件的时间开销。

在向数据文件访问数据时，同样可以现在Buffer内线性搜索需要的数据，避免打开数据文件。然而当Buffer容量过于大时，在Buffer内的线性查找又会增加额外的开销。

在设定Buffer的容量为SIZE，程序进行以下操作：

1. 向空白数据库连续插入nrec条数据，记录时间；
2. 查找全部数据，记录时间；
3. 清除数据库，nrec += nrec; 重复1-2过程；
4. 改变Buffer容量大小，记录各个容量对应时间；

查找过程中Cache容量为1，不进行重复查找，避免Cache影响结果。当SIZE=1时，效果等同于无Buffer。在nrec从1000增至10000过程中，记录SIZE从1增至nrec时上述操作的时间，可分别的到不同数据量下插入操作耗时与Buffer容量的关系。

具体数据以及结果分析见8

7.2.4 Cache对不同分布的热度数据查找时间的影响

Cache类的设计旨在在较高层面上建立一个位于内存中的小型数据库，针对高频率高热度的数据查找效率做出优化，完全避免了B+树中查找引索个data文件中读取数据。

然而过大容量的Cache会完全架空底层数据库的实现，数据库变为内存数据库，况且也并不能避免对文件进行读写。用链表实现Cache时，查找数据O(n)的复杂度不一定能够带给程序比访问引索文件和数据文件更少的时间开销。

由此可见，Cache类最大的优化效力是针对高频高热度的数据查找，且容量不宜多大。

设定Cache容量为SIZE，进行以下操作：

1. 向数据库插入100 000条数据，key为前100 000自然数；

2. 查找[50000-SIZE/2, 50000+SIZE/2]的数据，更新Cache；

3. 随机生成key服从均值为50000，方差为DIV的正态分布，查找条该数据；

4. 重复3过程100 000次，记录过程时间；

Buffer容量定位1，消除影响。在Cache容量从0(对照组)增长到100 000时，分别记录不同DIV下访问大量正态数据流的时间，可以得到Cache优化效力最高时对应的长度和数据类型。

具体数据以及结果分析见8

7.2.5 引索文件流在系统缓存的存在性验证测试

类比于数据文件的读写，在读写引索文件时依然会每次打开引索文件。然而不同于数据文件，在存有1 000 000条<int-string>(string长度均为100)的数据库中，数据有文件达到127mb，引索文件仅有8.7mb。而且数据文件的Buffer对程序优化显著。 故可知，多数据库的操作过程中中，引索文件的读写时间占比小于读写数据文件。引索文件读写不是主要的时间消耗。

结合文件流类FILE的特性，考虑到存在系统级别的优化，本程序只做证明层面的测试，过程如下：

1. Buffer容量设为1，消除影响；

2. 向空数据库内连续插入nrec跳数据，记录时间；

3. 修改程序，在B+树调用FILE接口fwrite()后，立即执行fflush()接口，确保系统级别的缓存被更新，确实有读写文件的过程发生；

4. 重复2过程，记录时间；

nrec从1增至100 000过程中，可以分别比较强制调用fflush()和不调用的性能差异，做出验证。

具体数据以及结果分析见8

7.3 用于空间性能分析的测试程序

7.3.1 大量数据增删时，Holder对data文件大小的影响

Holder将保持被删除数据在data文件中的序列化位置。并在新插入时，利用fseek()这一FILE的接口，在指定位置上复写上新的数据，减少data的空间消耗。

本程序验证这一优化，进行以下操作：

1. 向空白数据库插入nrec条数据，每条数据为长度为100；

2. 依次删除所有数据后重新插入nrec条数据；

3. 关闭数据库，记录.dat文件的大小；

当nrec从10 000 增至100 000时，分别记录有无Holder的数据库产生的.dat文件大小，比对分析。

具体数据以及结果分析见8

7.3.2 大量数据增删时，Holder对index文件大小的影响

与数据文件类似，Holder在新插入数据时，会调用B+树的replaceKey(int oldKey, int newKey)接口，而并非直接插入新的B+树节点。减少了index文件的空间消耗。

测试程序与7.3.1一致，同时记录idx文件大小并进行比对分析。

具体数据以及结果分析见8

8. 测试结果分析

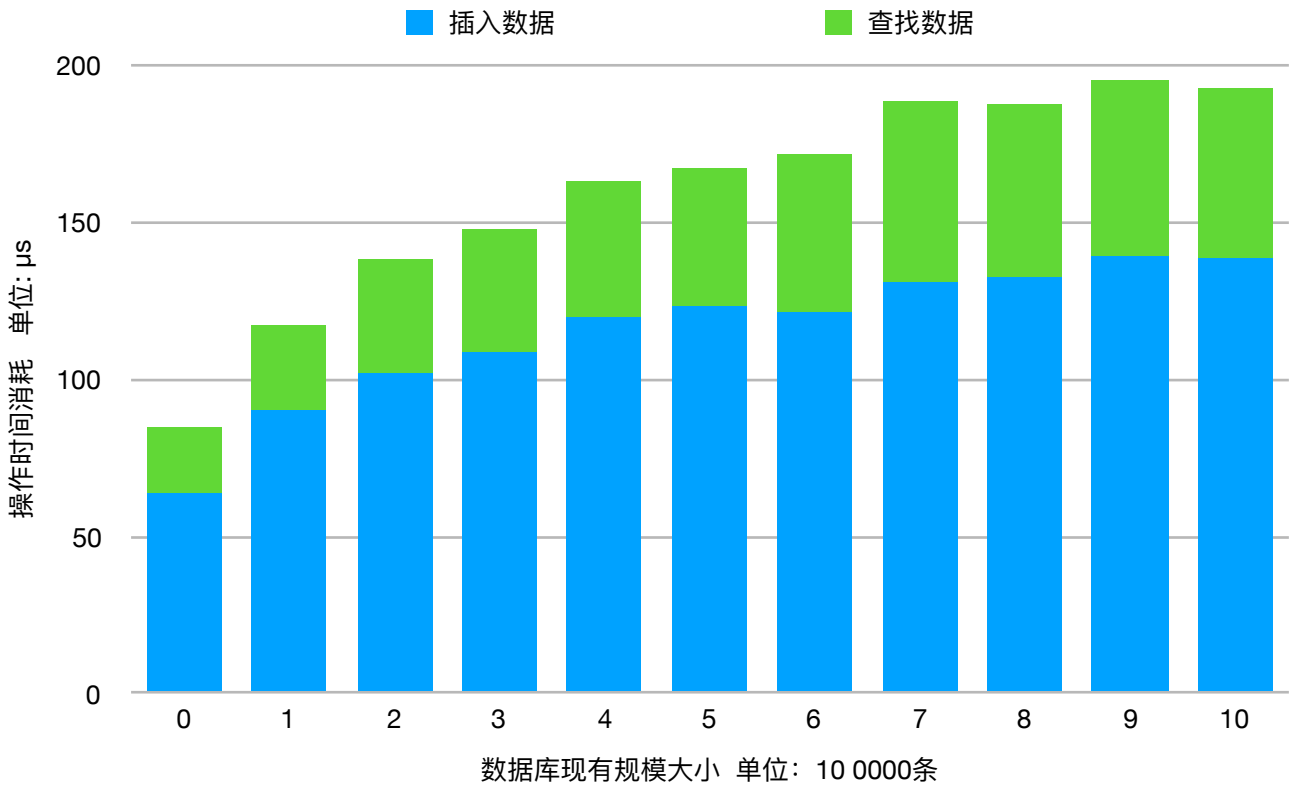
8.1 正确性测试结果分析

分别进行7.1中三次测试，测试7.1.1和测试7.1.2均无错误抛出；测试7.1.3统计结果与预期一致。故认为程序通过正确性测试。具体可见5.4一致性证明。

8.2 关于时间性能的测试结果分析

8.2.1 数据库规模大小对单词插入查找的影响

数据库规模/10万条	0	1	2	3	4	5	6	7	8	9	10
插入数据时间/μs	64	90	102	109	119	123	121	131	132	139	139
查找数据时间/μs	21	28	36	39	43	45	50	57	55	56	54



对于插入一条数据，主要的时间开销为B+树内搜索时间(包括读取index文件)，写入数据到data文件，B+树内新增键值(写入pos到index文件)。测试中value均等长且无需新增Pair和按顺序写入保证了写data文件的时间复杂度为O(1)；B+树节点容量定为256，并在10万次的整数倍时插入，保证了新增节点时不进行树的分裂和重构，使得B+树内新增键值的时间复杂度为O(1)；故对插入数据的主要时间影响因素为B+树内的搜索过程。

B+树内的搜索过程时间复杂度为O(log n)，对于插入和查找，途中曲线趋势近似贴合这一结论。

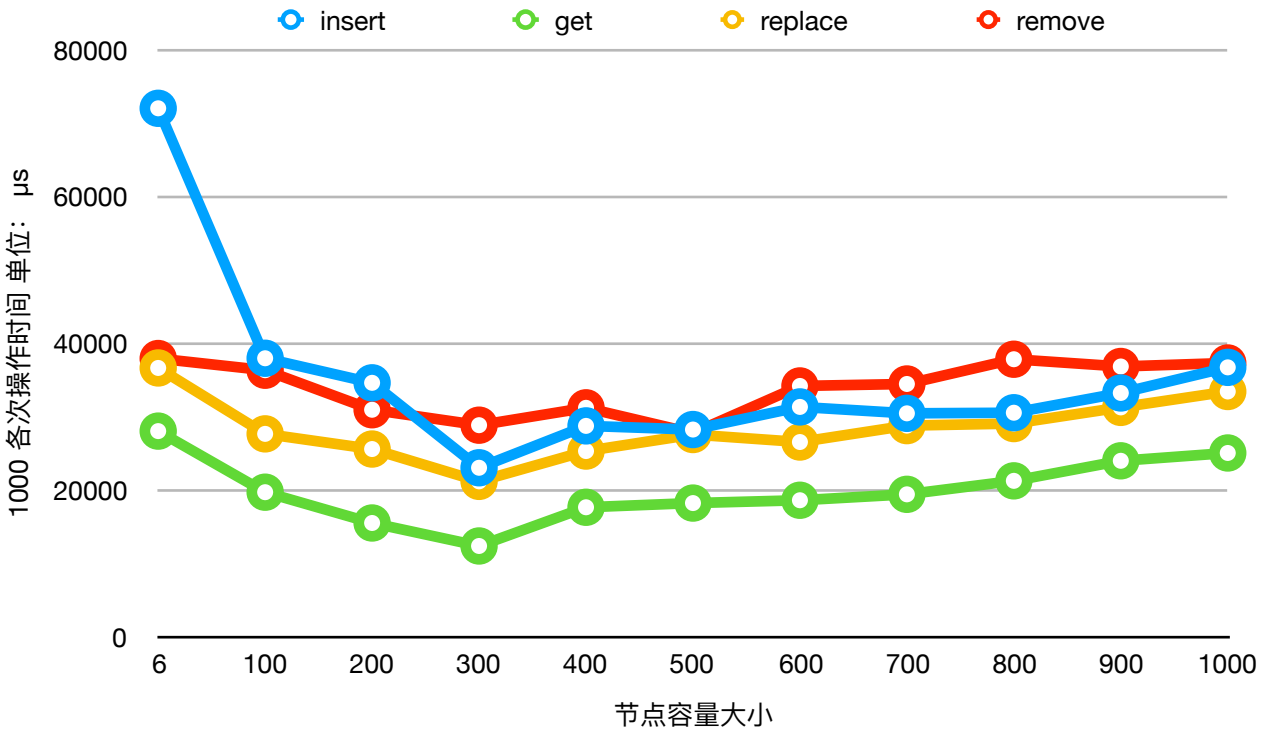
8.2.2 B+树节点大小对操作时间的影响

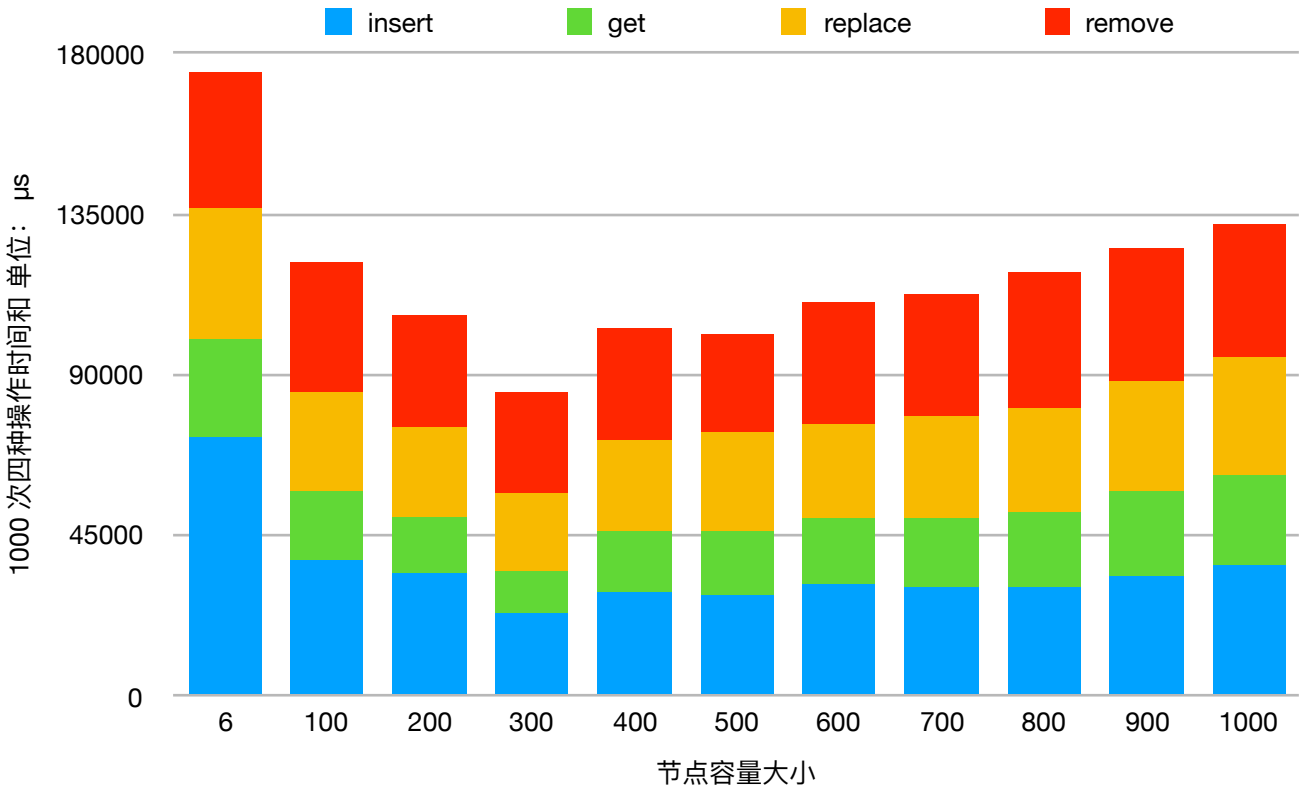
不同节点大小下，各操作耗时 单位：μs

SIZE	insert	get	replace	remove
6	72115	28000	36632	37919
100	37956	19652	27623	36319
200	34599	15444	25550	30919
300	22991	12302	21132	28810
400	28713	17592	25273	31183
500	28215	18166	27533	27737
600	31306	18551	26508	34173
700	30429	19361	28750	34441
800	30512	21203	29031	37823
900	33214	23942	31241	36831
1000	36712	25012	33414	37312

在数据总量(1000)恒定的情况下，改变B+树节点的大小会大幅度缩短B+树的层高。特别地，当节点大小等于1000时，B+树完全退化为在根节点上的线性结构。当节点大小为2时，B+树可视作一个非常

难以维护的BST(要求叶节点高度一致)，此时层高剧增，各个操作时间开销均达到SIZE=6时的十倍以上。在图表中不具有比较价值，故SIZE选取从6开始。可以明确地是，SIZE由2到6时，各个操作时间消耗急剧下降。





由图中可以看出：

1. 对于各个操作，大致趋势为减少到增加；

2. 操作时间下降阶段急剧，增长阶段缓慢；

对于增减趋势：减少的原因可以归结为B+树结构变得简单，维护次数变少，对叶子节点的递归寻址次数变少。增加的原因可以归结为单次读取节点时间变长，在节点内部做线性查找时间变长。

节点大小由2增至300的阶段，B+树的结构进行了大幅度的简化，对B+数的维护工作减少，插入操作更多的变为直接在节点中插入。同时，层数减少后，查找中寻址次数变少，所以时间减少。

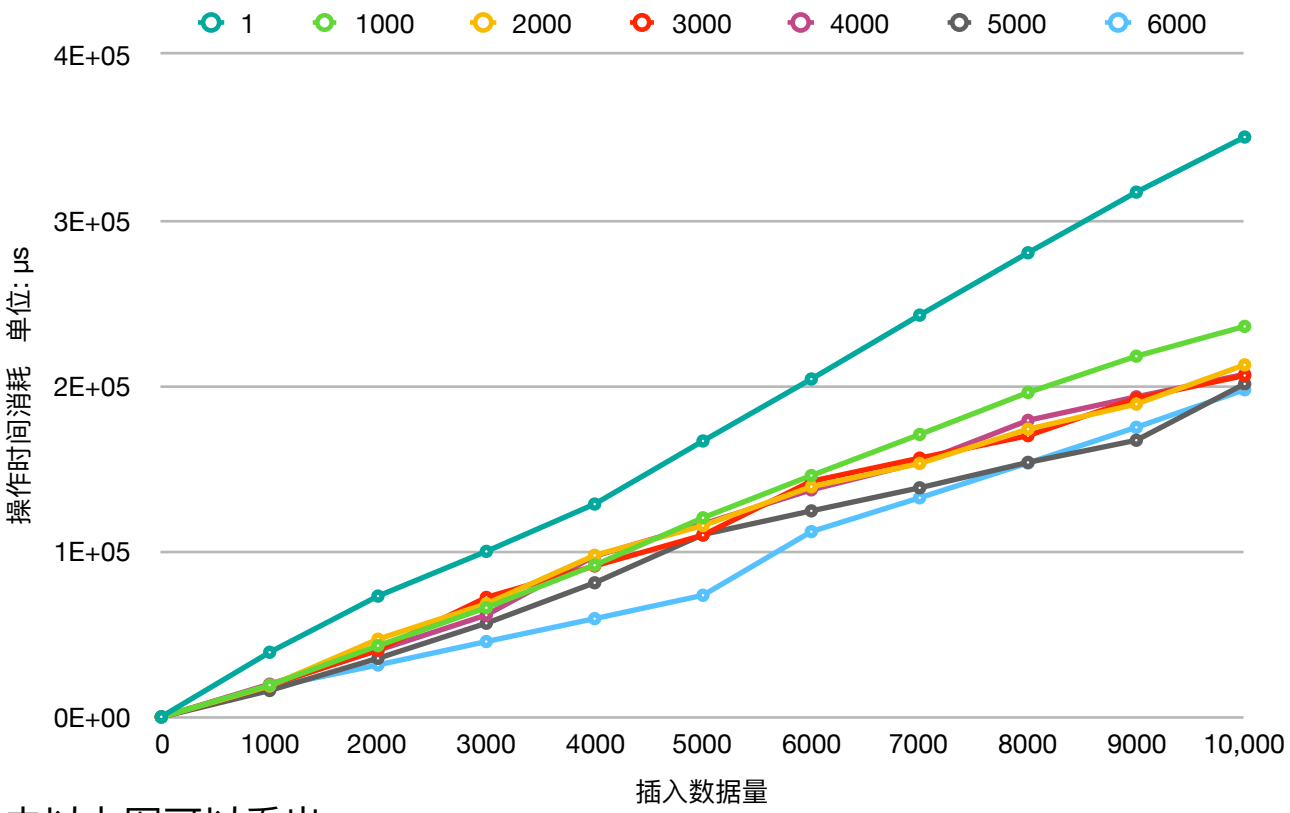
由300增至1000阶段，对B+树结构的简化已经比较充分了(300高度为1，1000时高度变为0)，在维护B+树上不能进一步减少时间。相反，单次读写节点时时间增大，节点内搜索时间变长。

增长过程缓慢可解释为单次读写B+树节点时间开销不及读写数据文件的开销，所以在总时间上显得增长缓慢。但B+树维护工作比较复杂，高层数的B+树多次读写引索文件寻址时间开销与读写数据文件相当，所以对B+数结构的简化让时间开销下降较多。

8.2.3 Buffer容量对数据库读写数据时间的影响

插入不同数据时，不同Buffer大小用时

Data \ SIZE	1	1000	2000	3000	4000	5000	6000
0	199	79	52	96	43	34	35
1000	39119	19318	18788	19027	19858	16051	17885
2000	73031	43064	47016	40490	40320	35466	31482
3000	99998	66024	68380	72212	61575	56632	45580
4000	128559	91694	97613	91225	97142	81048	59409
5000	166634	120283	115580	109791	116535	110177	73510
6000	204007	145776	139014	142309	137117	124432	111918
7000	242555	170557	153049	156264	153642	138364	132340
8000	280176	195887	173745	169836	179060	153693	153342
9000	316783	217794	188893	192309	193267	167047	174865
10000	350028	235762	212530	206137	206844	201079	197548

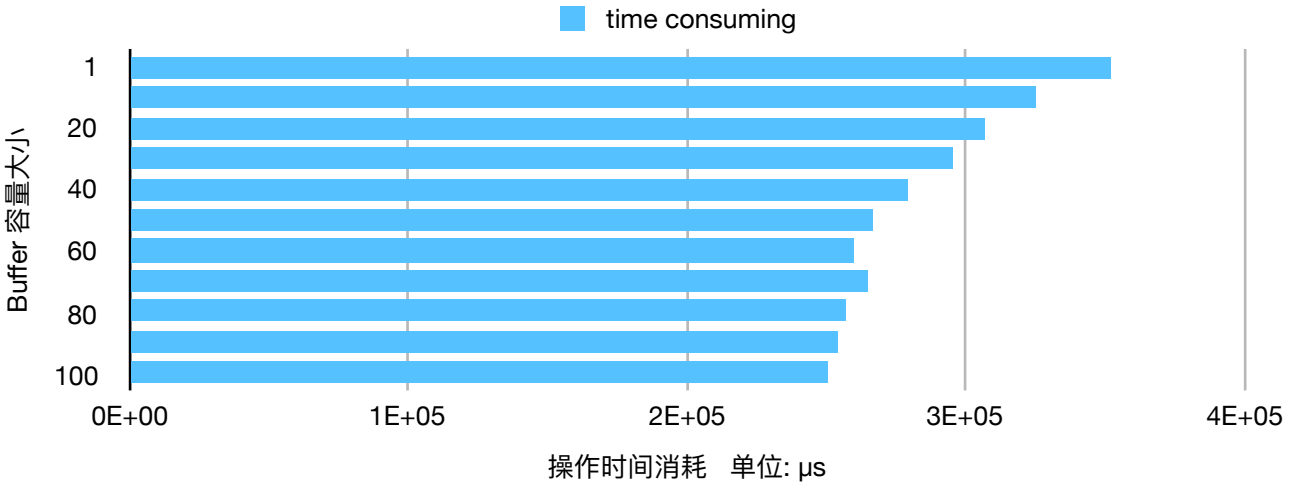


由以上图可以看出：

当Buffer容量增大时，确实对数据库插入程序的时间进行了优化。然而当Buffer容量达到2000 (20%nrec)时，优化效果变得不显著。因为

Buffer的刷新机制，导致容量在5000以上时，10000次数据插入Buffer只能flush一次 (容量为6000时，显著可见在数据为6000时有额外时间开销)。

所以进一步决定针对固定的数据总量10000，Buffer容量由1递增至100，观察时间优化效益。



SIZE	1	10	20	30	40	50	60	70	80	90	100
Time	351744	325450	307067	295152	279415	266812	260215	265241	256945	254034	250755

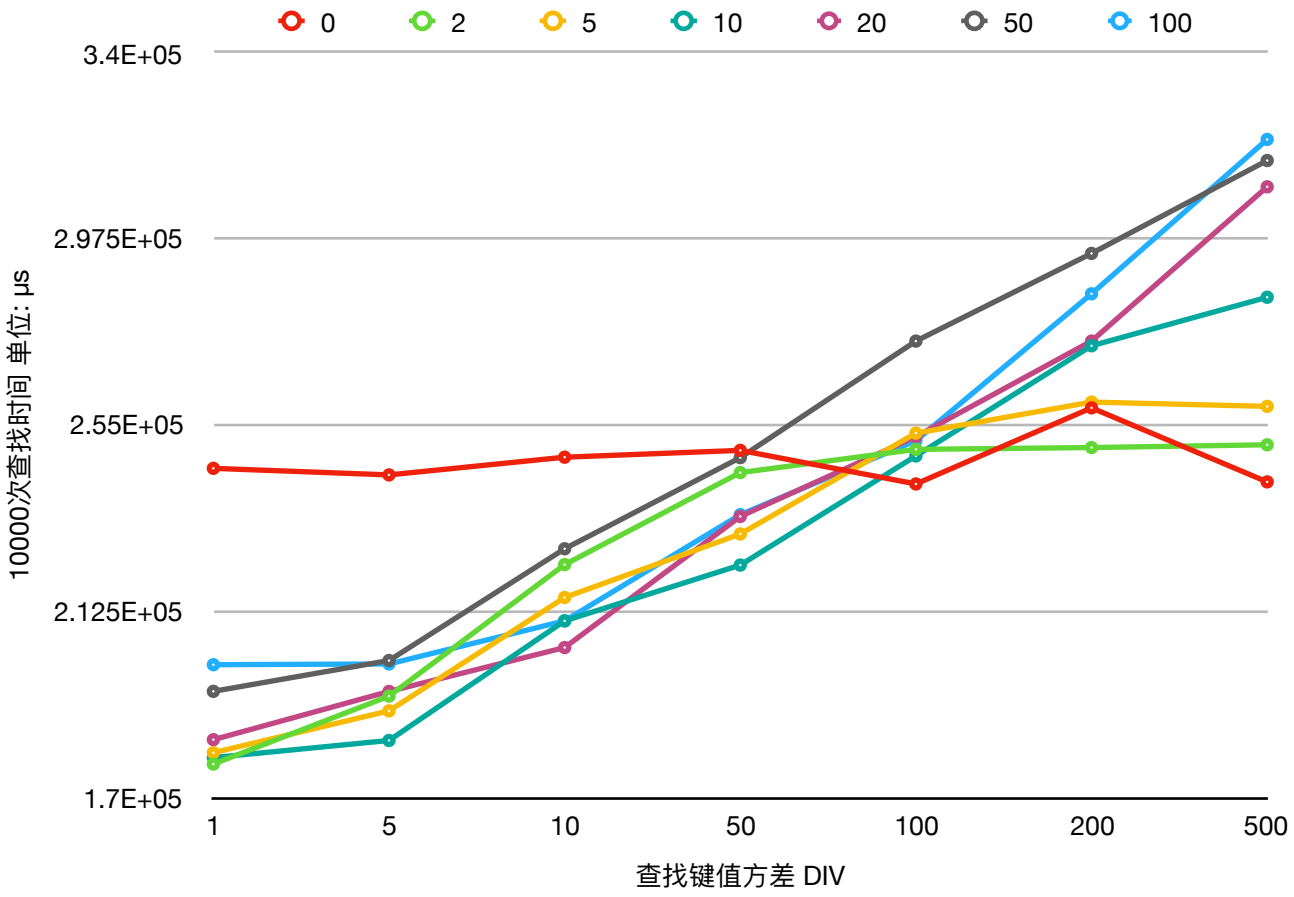
对于Buffer类来说，当插入的数据总为10000时，当其最大优化力度时，Buffer的容量为10000。然而此时数据库的所有数据都会被寄存在Buffer中，虽然时间效率提升了，带来了巨大的内存开销，不适用于大量数据的存储，背离了数据库的设计目的。

以10000条数据规模为例，在Buffer容量达到100时，插入时间从不加Buffer的351ms优化到250ms，优化力度达到28%；当Buffer容量为1000时，优化力度为32%；当Buffer容量达到10000时，达到极限优化力度42%。可以确定，为了达到更好的优化效果，为Buffer付出的内存资源需要更多。由此用户可以根据资源情况和优化要求合理设置Buffer的大小。

8.2.4 Cache对不同分布的热度数据查找时间的影响

测试数据控总量为10 0000条，键值为1-10 000的连续整数。程序给出一个随机变量服从均值为50000，方差为DIV的正态分布。以随机变量为键值查询数据库。不同方差，不同SIZE情况下，耗时如下：

DIV \ SIZE	0	2	5	10	20	50	100
1	245100	177653	180244	179125	183211	194238	200315
5	243602	193141	189788	183034	194212	201267	200482
10	247643	223145	215646	210314	204224	226754	210311
50	249201	244128	230127	223041	234125	247544	234534
100	241541	249423	253124	247931	252341	274124	251412
200	258899	249846	260211	273052	274125	294123	284912
500	242006	250471	259231	284124	309312	315312	320123



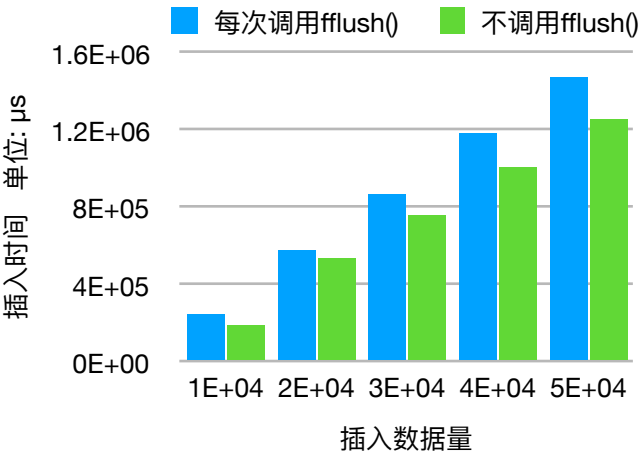
在5附加结构-5.1Cache中已经阐明了Cache的核心数据结构为一双向链表。在立案表内的查找速度由于B+树查找和数据文件读取的前提是查找距离够短。为了保证查找距离短，首先要保证Cache持有目标数据。否则查找过程中则会先遍历Cache，增加额外的时间开销。

图中可见，由双向链表作为Cache的核心数据结构的适用范围并不算广泛。在Cache容量小于10且方差较小时，有一定的优化作用。对于容量过大的Cache，无法包含全部热点数据，搜索过程也是一笔不小的开销。Cache适用范围小的根本原因还是其搜索过程的时间复杂度O(n)偏高。

8.2.5 引索文件流在系统缓存的存在性验证测试

测试将不断插入数据到空白数据库，记录时间。唯一变量为index文件读写时是否调用fflush();

时间/μs	每次调用fflush()	不调用fflush()
10000	245722	194060
20000	570773	541264
30000	868660	755353
40000	1183809	1005019
50000	1471623	1250066



由图表可知，每次读写index文件后，都自动调用一次fflush()让文件流从系统缓存上流入磁盘，反而增加了时间开销。因为系统缓存的存在和index文件本身较小的空间大小(与对应的data文件大小之比约为1 : 10)，index文件的读写过程可能没有直接访问

磁盘。但每次都进行flush()之后，反复的读写文件反而加大了时间开销。可以推出：对于较小的index文件(10万条数据1.2mb)，其内容可能被储存在系统缓存中。

8.3 用于空间性能分析的测试程序

8.3.1 大量数据增删时，Holder对data文件大小的影响

测试程序在有Holder和无Holder的情况下分别交替进行插入数据和删除数据，观察data文件在磁盘上的实际大小。

向数据库内连续交替增删10 0000条数据，每条数据为长度100的string，data文件在无Holder的情况下大小为11.2mb；在有Holder的情况下大小仅为112字节(包括了一些数据信息)。

说明在类似的连续增删情况下，Holder可以使得data文件空间效率达到最优。

8.3.2 大量数据增删时，Holder对index文件大小的影响

测试程序同8.3.1中程序。

index文件在磁盘实际大小由1.6mb变为2kb。

证明了Holder对data文件和index文件同样具有空间优化效力。

9. 优化方向

9.1. Cache的核心数据结构效率不高，适用范围窄，有待改进；

9.2. Holder没有做到真正删除文件，只做了数据的修改。换言之，data文件和index文件大小只增不减；

9.3. Pair类实现可变长value造成了一定量的空间浪费；可一加入记录长度的值，用偏移量寻址；

9.4. 大规模数据的index文件依赖系统缓存不稳定，可为其编写缓存工具；

10. 参考及致谢

1. www.wikipedia.com/database/ —Database
2. https://www.cs.usfca.edu/ B+tree visualization
3. 邓俊辉 《数据结构(C++语言版)》
4. W.Richard Stevens / Stephen A.Rago
   《UNIX环境高级编程apue》(第二十章 数据库函数库)
5. 《Programming Abstractions in C++》Eric S. Roberts

感谢戚老师详实，全面地讲授了实现数据库核心数据结构的知识；

感谢董助教，张助教对我提出的问题不厌其烦地解答；

感谢一起交流的同学们，在相互讨论中，我对项目有了更深的理解。