数据库设计文档

曹金坤

515260910022

目录

1 简介

- 项目名称：多功能单值数据库
- 主要数据结构：B+ 树
- 开发环境：Win10 Build 14393 & MacOS 10.12.5
- 测试环境：WIn10 Build 14393
- 基本功能：实现一对单值数据 <key, value> 的如下操作：
  - 添加一对 <key, value> 数据进入数据库
  - 删除数据库中一对 <key, value> 数据
  - 修改数据库中一对 <key, value> 数据的 value 信息
  - 通过给定 key, 查找数据库中对应的 value 信息
  - 顺次输出数据库中存储的所有 <key, value> 数据

2 数据库特点

2.1 二次包装

数据库的实现通过一个句柄类 DBHANDLE 传输给外界。使用者的对数据库的相关操作通过操作这个公开的句柄接口得以实现；

这样的设计，使得数据库用户逻辑更加清晰，同时也使得同时对多数据库的复杂操作拥有统一的逻辑和接口，使得在不更改用户接口的前提下，可完成对底层数据结构和中层数据库实现方式的改变。此外，经过测试，二次包装对数据库主要操作的性能没有明显负面影响。

2.2 不定长数据

数据库默认的 <key, value> 类型为 <string, string> 型，并且支持不定长的数据存储。这样的设置是为了代码的简洁与检验的方便，可通过模版编程轻易地扩展数据库支持的数据类型。但是，需要注意

的是，key 的数据类型需要支持"==","<",">" 操作，即是可比较大小的；

2.3 多模式

考虑到索引文件的大小，数据库提供的两种构造模式，默认模式（MODE A）会将所有的索引信息读取到内存中进行操作，并在数据库关闭时重写索引文件，这种方式适用于大多数情况（已利用 1000 万条数据测试），并且较为快速；但当索引文件过大时，可选取 MODE B，使数据库可同时处理文件和内存中的索引信息，而不需要读取全部索引信息进入缓存；

数据库的模式需要在构造数据库时指定，需要注意的是，为了尽力压缩默认模式下的索引文件大小，提高数据库性能，MODE A 和 MODE B 的索引文件并不通用，相关细节可见 3.6 模式 B 简介部分；

2.4 文件分离

数据库依赖索引文件 (.idx) 和数据文件 (.dat)，在构建数据库的过程中，相应的数据结构中不需要即时读取.dat 文件的信息，只有在需要对某个 key 关联的 value 进行操作时（查找，修改等），才需要进入.dat 文件进行操作；

尽管索引文件和数据文件的分离为数据库设计的一般规则，但是在该数据库中，这种设计使得不同模式下的数据库可以共用同一个数据文件，并且可以利用一份数据文件，构建出各自需要的特定格式的索引文件。

2.5 优良空间性能

数据库的构建过程中，充分考虑到了对计算机存储空间的优化，主要体现在两个方面：

2.5.1 空间重利用

数据库使用过程中，当删除了存储的数据时，会在.dat 文件中产生空闲的文本区间，此时，当需要向.dat 文件中写入新的数据时，优先利用这些空闲的中间位置会减小数据文件的大小，达到提升数据库空间性能的作用。需要注意的是，因为数据库支持不定长数据存储，所以空间重利用的尝试并不总会成功，详见 3.5 空间重利用部分；

2.5.2 压缩索引文件内容

数据库的设计中，尽量压缩了索引文件的大小，节约计算机存储空间，需要指出的是，数据库仅在关闭时更新索引文件，而索引文件中的索引信息总拥有最紧密的结构，已无法在不改变编码格式和数据库基本逻辑的前提下进一步优化；

2.6 优良时间性能

为了减小数据库操作的时间延迟，带来更好的使用体验，考虑到硬盘和内容的性能差异，数据库的操作应当尽量少地进行对文件的读写操作，为此数据库中设置了缓存类 buffer，用于记录最近对数据的查找、更改等操作，借此尽量利用对内存对象的读写替代对硬盘文件的读写，提高时间性能，细节详见 3.3. 自定义结构部分中的缓存（buffer）相关内容。

3 数据库实现

3.1 接口设置

- 创建

  DBHANDLE* db_open(char *pathname, OPEN_TYPE oflag, DB_MODE mode = A)

  - 创建一个新的数据库
  - pathname: 文件路径名
  - oflag: 创建形式；可选项：READ, WRITE, CREATE：
    * READ: 当目标路径下存在相应.idx 和.dat 文件时，利用其建立数据库，并且仅可读取数据库信息，不可执行更改操作
    * CREATE: 目标路径下不存在相应.idx 和.dat 文件时，创建空.idx 和.dat 文件，建立空数据库，拥有读写权限

    * WRITE: 当目标路径下不存在相应.idx 和.dat 文件时，创建空文件和空数据库；当相应文件存在时，利用其建立数据库，拥有读写权限

  - mode: 数据库模式。可选项：A, B:

    * A: 默认数据库模式，读取全部索引信息进入内存并操作

    * B: 支持超大索引文件的数据库模式，建立数据库时仅读取索引文件中的部分信息进入内存，索引文件中保留和数据结构，对数据结构的操作直接实现在文件中。

  - 返回一个数据库句柄类的指针

- 关闭

  void db_close(DBHANDLE* db)

  - 关闭数据库
  - db: 数据库句柄类的指针
  - 关闭数据库前，会强制将缓存中的信息写入数据文件，并且重写索引文件

- 存储

  int db_store(DBHANDLE* db, string key, string data, STORE_TYPE flag)

  - 存储数据进入数据库
  - db: 目标数据库的句柄的指针
  - key: 存储数据的键
  - data: 存储数据的内容
  - flag: 存储操作的类型。可选项：STORE, INSERT, REPLACE：

    * INSERT: 只支持 key 不存在时，向数据库中插入新的 <key, value>

    * REPLACE: 只支持 key 已存在时，更改数据库中 key 对应的 value 和相应信息

    * STORE: 若 key 已存在，则执行 REPLACE 操作，若 key 不存在，则执行 INSERT 操作

* 如果成功进行了一次存储操作，则返回 0, 否则返回-1

- 查找

  string db_fetch(DBHANDLE* db, string key)

  - 查找数据库中的数据
  - db: 目标数据库的句柄的指针
  - key: 需要找到的键的内容
  - 如果索引存在于数据库，则返回对应的数据信息，否则返回空

- 删除

  int db_delete(DBHANDLE* db, string key)

  - db: 目标数据库的句柄的指针
  - key: 需要删除的目标索引
  - 如果成功删除则返回 0，否则返回-1

- 重置读写头

  void db_rewind(DBHANDLE* db)

  - db: 目标数据库的句柄的指针
  - 在数据库中，有将所有索引串联的链表对象，该操作使得位于该链表上的读写头位置恢复到链表开端

- 顺序获取数据

  string db_nextrec(DBHANDLE* db, string key)

  - db: 目标数据库的句柄的指针
  - 借助于串联所有索引的链表对象和读写头，读取给定的 key 后一个索引的信息，返回相应数据，如果读写头已经到达链表尾部，说明已经遍历索引信息，返回空

3.2 代码模块

源代码提供了如下五个文件模块：

- **bplustree.h**

  数据结构定义文件，提供了节点类 Node，B+ 树类 Tree 的定义，以及与之相关的对该数据结构操作所需要的变量和函数的声明和实现；

- **database.h**

  出于对数据库“二次包装”的理念，该头文件提供了数据库句柄类 DBHANDLE 以及相关辅助类的定义，以及为了对一个基于 B+ 树的数据库操作所需要的变量和函数的声明和实现；

- **corr_test.cpp**

  提供了对数据库操作正确性方面进行测试所需要的代码，具体内容详见 4. 测试与分析部分；

- **time_test.cpp**

  提供了对数据库操作时间性能方面进行测试所需要的代码，具体内容详见 4. 测试与分析部分；

- **space_test.cpp**

  提供了对数据库操作空间性能方面进行测试所需要的代码，具体内容详见 4. 测试与分析部分。

3.3 自定义结构

3.3.1 buffer(缓存)

map<string, Token*> buffer

缓存 (buffer) 是一个 stl 库的 map 对象，利用索引信息 key 查找一个特定的 Token 对象，其中 Token 是一个自定义的类，其内部包含 **string key, string value, bool deleted, int offset** 四个成员变量，记录了对一个 <key, value> 的操作信息，其中，deleted 表示该键值对上一次接收到的的操作是否为有效的“删除”操作，offset 记录了在数据文件中该 value 数据的起始位置（偏移量）。

当对数据库中的 <key, value> 记录进行有效的删除、插入、修改和查找操作时，更新后的 <key, value> 信息会被首先记录在缓存中，而不是直接写入数据文件。当缓存容量达到预先设置的阈值或者即将关闭数据库时，会刷新缓存，将其中记录的信息逐一写入数据文件中。其中，由于使用了 map 类型，所以对 key 的查找效率远高于 vector 的线性查找，通常也明显优于对一个拥有巨大数据量的 b+ 树中叶结点信息的查找。另一方面，因为 map 支持键值对的直接映射和修改，所以可以保证缓存中的 <string, value> 信息始终处于最新状态，单次刷新中至多只会更新一个 key 对应的 value 一次，大大减少了读写硬盘中文本信息的频率，极大的提高了时间性能。

3.3.2 mrySpace(内存空间)

mrySpace 是一个自定义的类，记录了一块内存空间的信息，并且映射到数据文件中一块特定位置的定长区间，其具有以下成员变量：

- **int len**: 对应数据文件中空间的长度；
- **int offset**: 对于数据文件中空间的位置；
- **preSpace***: 该块内存空间链接的前一块空间的指针；
- **nextSpace***: 该块内存空间链接的后一块空间的指针；

在一个数据库句柄中分别有两条 mrySpace 的链表，分别记录了正在存储有效数据的空间和已经被删除数据对应的空间，这种设置为数据库插入和修改信息时进行的空间重利用提供了方便。

3.4 接口功能实现

- 打开

  1. 如目标文件不存在，而打开的方式为 READ, 则打开失败，终止操作；
  2. 如目标文件存在，而打开的方式为 CREATE，则打开失败，终止操作；
  3. 其余情况下，如文件存在则读取索引文件中信息构建 b+ 树结构，否则创建空树和空数据库句柄；

4. 初始化正在使用的内存空间和空闲内存空间两条链表，初始化缓存空间；

5. 返回该数据库句柄的指针;

- 关闭

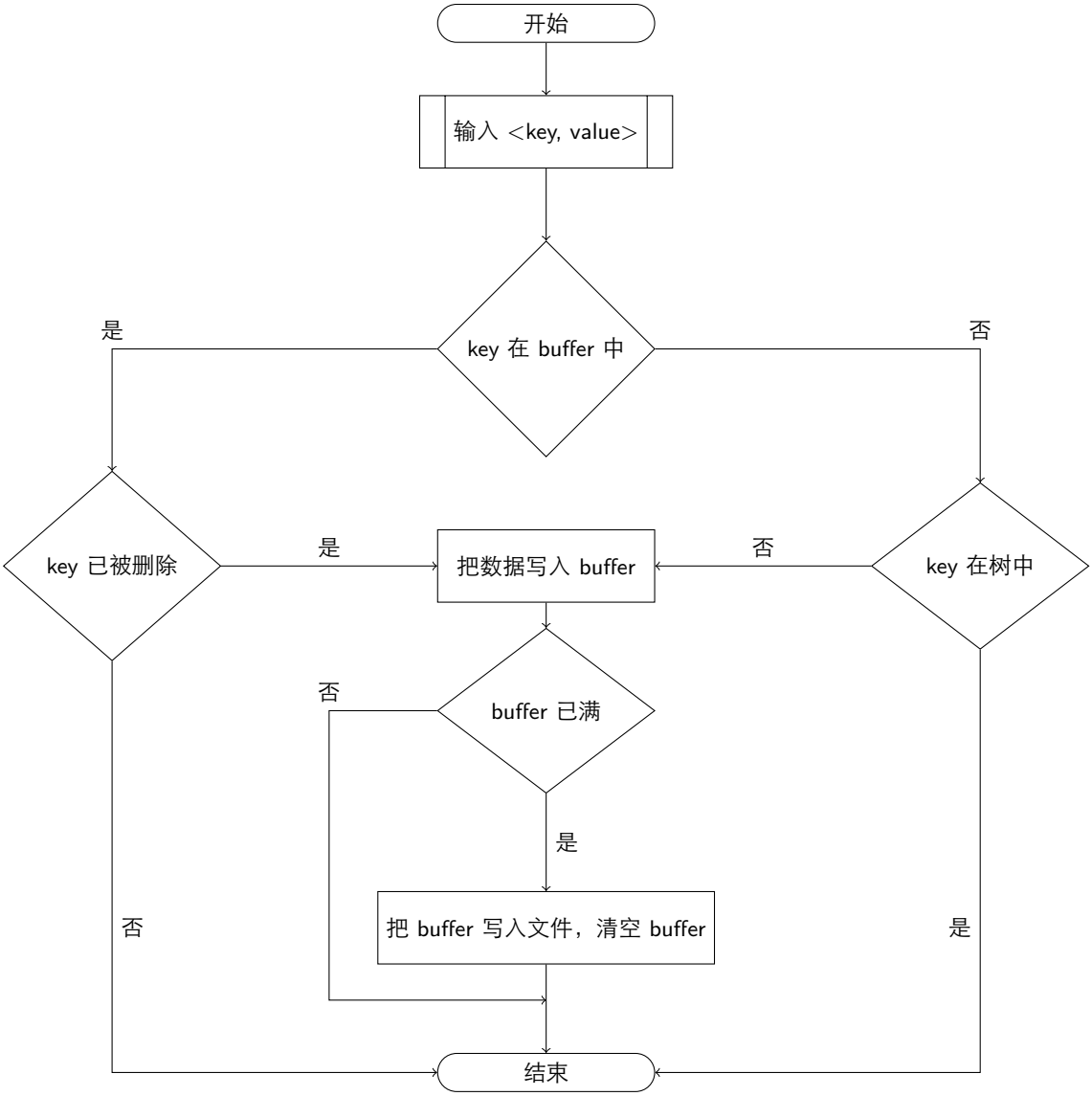
1. 将缓存中的数据写入数据文件；

2. 输出内存中的叶结点信息，重写索引文件；

3. 关闭索引文件和数据文件流;

- 插入

插入操作的流程图



1. 检查缓存中是否已存在 key：
   - 若存在于缓存，且并不处于被删除的状态，则该 key 存在于数据库，终止操作；
   - 若存在于缓存，但处于被删除的状态，则把插入信息写入缓存，若缓存满，则刷新，之后终止操作；

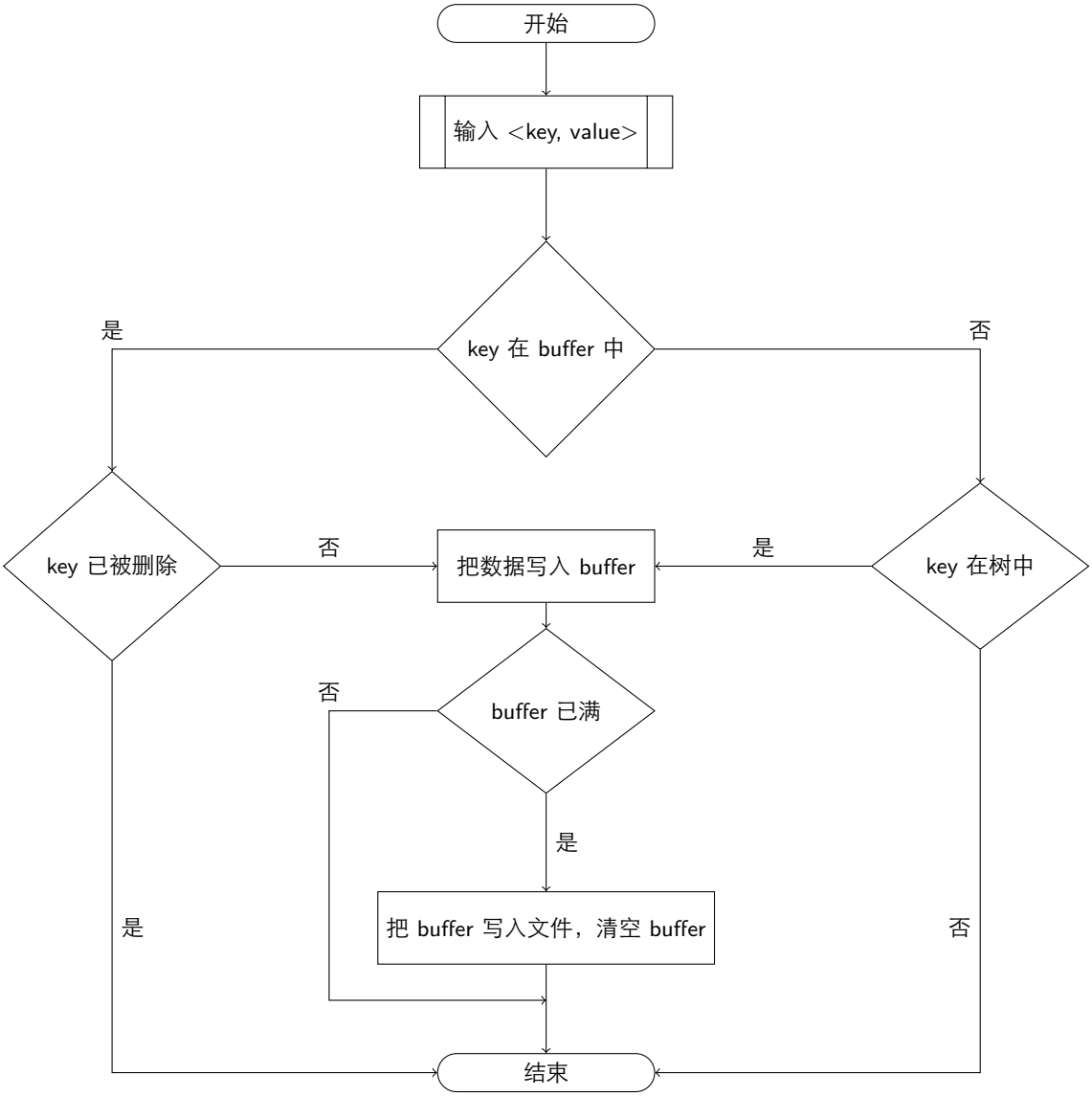
- 若不存在于缓存中，进入树结构中搜索该 key；

2. 检查树中是否已存在 key：

   - 若不存在，则向缓存中写入该条信息，终止操作；
   - 若已存在，则终止操作；

- 修改

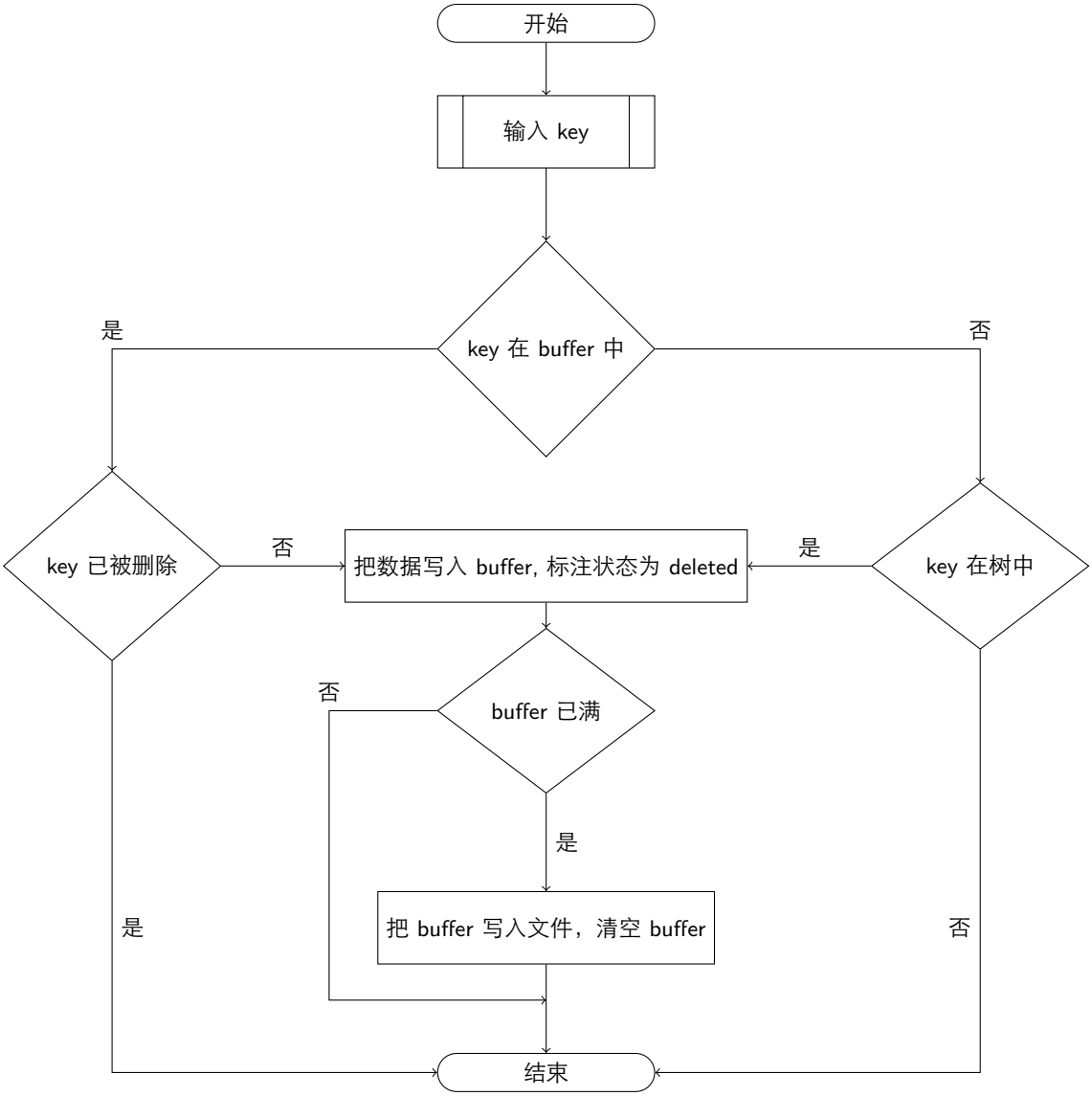
修改操作的流程图



1. 检查缓存中是否已存在 key：
   - 若已经存在于缓存中，且未被删除，则将新数据写入缓存，若缓存已满，则刷新缓存，终止操作；
   - 若已经存在于缓存中，但已被删除，则终止操作；
   - 若不存在于缓存中，尝试在树结构中搜索 key；
2. 检查树中是否已存在 key：
   - 若 key 不存在于树中，则终止操作；
   - 若 key 存在于树中，则向缓存中写入更新后的信息，若缓存已满，刷新缓存，终止操作；

- 删除

删除操作的流程图



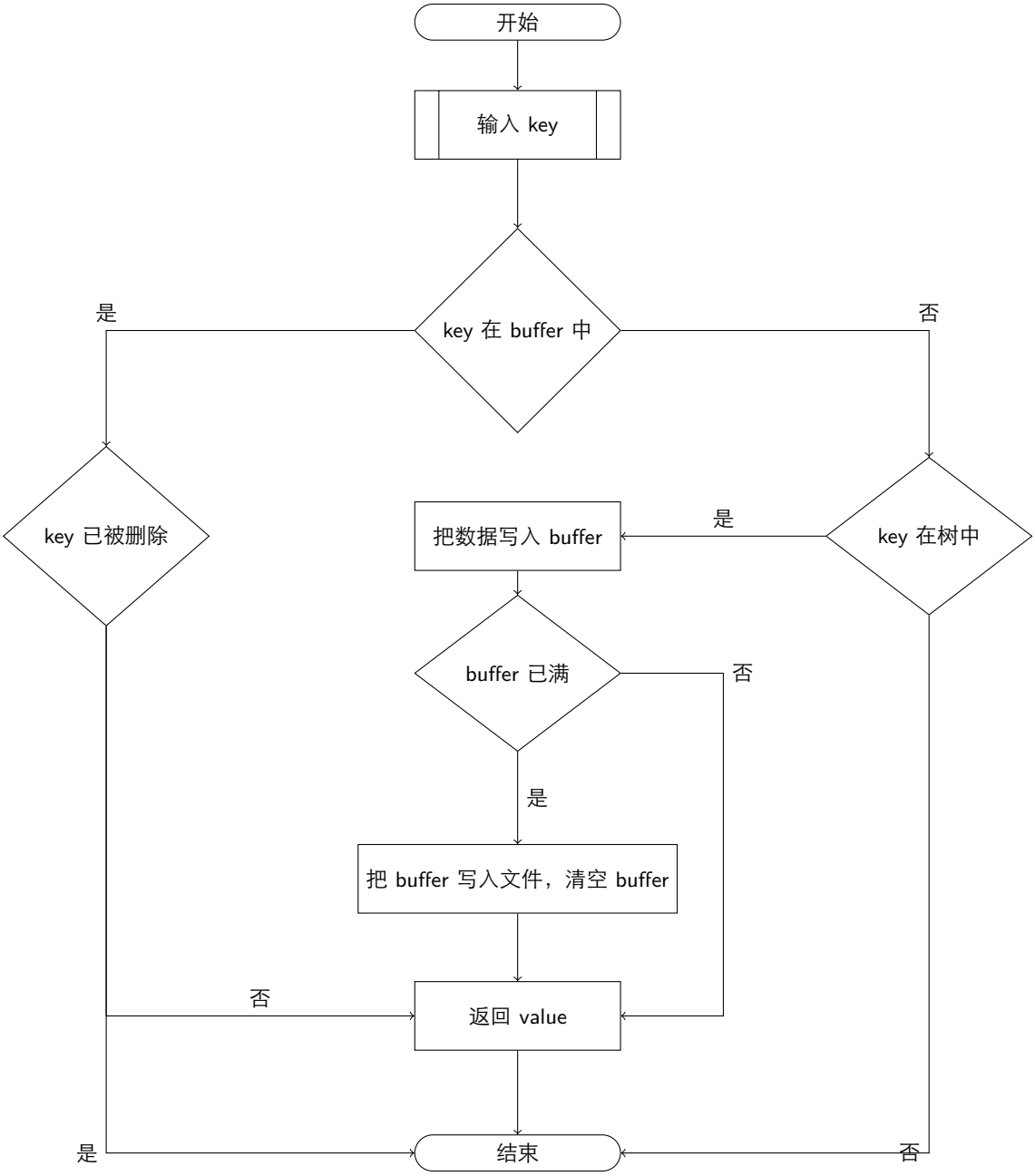
1. 在缓存中搜索 key:
   - 缓存中存在 key，且已处于被删除状态，终止操作；
   - 缓存中存在 key，但不处于被删除状态，将其置于被删除状态，终止操作；
   - 缓存中不存在 key，尝试在树结构中搜索 key；

2. 在树结构中搜索 key:

   - 树结构中没有 key，终止操作；
   - 树结构中有 key，将该条信息写入缓存，并设置为“deleted”状态，若缓存已满，刷新缓存，终止操作；

- 查找

查找操作的流程图



1. 在缓存中搜索 key：

   - 若缓存中存在 key，且不处于被删除状态，则得到对应 value，返回 value，终止操作；

- 若缓存中存在 key，但处于被删除状态，返回空，终止操作；
- 若缓存中不存在 key, 尝试进入树结构搜索；

2. 在树结构中搜索 key:

- 若 key 不存在于树中，返回空，终止操作；
- 若 key 存在于树中，获取对应子节点的索引信息，进而在数据文件中读取对应 value，返回 value, 并将该条信息写入缓存，若缓存已满，刷新缓存，终止操作；

3.5 空间重利用

当有数据记录从数据库中被清除时，相应的内存空间便闲置下来，对这些空间的重新利用，有利于压缩生成文件的大小，节约空间，而且在某些情况也有利于提高修改和插入操作的性能。该数据库的实现中，对空间重利用主要借助于以下两个定义在 DBHANDLE 类中的链表对象：

3.5.1 空闲空间链表 (emptySpace)

当一条数据库中的数据被成功删除时，原先这条数据指向的数据文件中的空间即被闲置，此时，从“非闲空间链表”中取出对应的 mrySpace 指针（存储了空闲文本空间的位置、长度等信息），将其加入 emptySpace 中，需要注意的是，应该尽量将拥有最大长度的空闲空间始终置于该链表的头部，从而可以在尝试重利用时很快得知是否有足够容纳新数据的空闲空间存在；

3.5.2 非闲空间链表 (filledSpace)

当一条数据从缓存中被刷新而写入数据文件时，他会在数据文件中占据一个特定的位置，并生成一个 mrySpace 对象记录他的位置和长度等信息，并加入 filledSpace 之中，当该数据被删除，对应空间闲置时，将该对象从 filledSpace 链表中取出，加入 emptySpace 链表中。需要注意的是，应该始终保持具有最大偏移量的对象位于 filledSpace 链表的头部，从而当需要在数据文件尾部添加数据信息时，可以很快找到正确的添加位置。

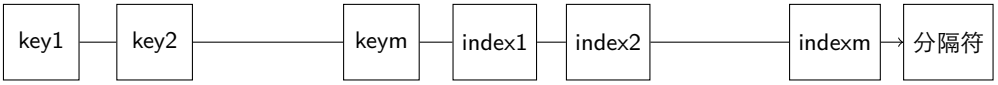
3.6 模式 B 简介

在默认的模式 A 中，索引文件中仅保留了构建 B+ 树时所需要的叶结点信息，即只保留了索引信息，而不记录树结构。但在模式 B 中需要记录 B+ 树的具体结构和节点关系，从而可以在不完全读取索引文件的情况下，对数据所在的树结构进行操作。

具体来说，模式 A 中，一条典型的索引信息是一个独立的索引数据，组成部分为：

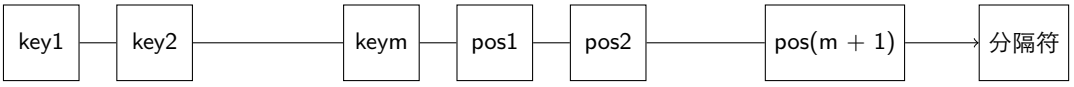
而在模式 B 中，索引文件的第一条信息为根节点的信息，一条典型的信息单元即是一个节点的所有信息，由 B+ 树节点的特点所决定，索引文件中存储的节点信息也分为两类。具体包括如下：

叶子节点



其中，一条 index 信息即是一对 <offset, len> 的组合数据；

内部结点



pos(i) 即是该节点第 i 个子节点在此索引文件中的位置，一般可由相对于文件头的偏移量来表示；

由上述的简介可以看出，模式 B 为了支持不常见的超大索引文件的操作，而大大损失了性能和通用性，由于在实际的测试中，模式 A 已经可以满足绝大多数甚至是刻意设置的极端情况的需求，同时也碍于时间的限制。所以仅在此提供模式 B 的实现简介。

另一点需要强调的是，在 B+ 树的操作中会涉及大量的节点上溢和下溢的操作，这往往需要递归地操作大量的节点信息，碍于模式 B 的设计限制，必然会导致大量的文件读写操作，反而会导致数据量较大时，难以忍受的时间延迟，无法满足日常需求。

4 测试与分析

注：在该节内容中，所有的测试均基于 $< string, string >$ 型的数据。同时为了保证测试结果的可复现，有序数据的选取方式一般由有序的整数型转化而来，细节可见相关代码文件中的测试模块。

4.1 正确性测试

4.1.1 大量数据测试

注：为了简化试验，该项测试中统一使用节点大小为 64。测试过程由文件 *corr_test.cpp* 中的函数 $big\_data\_test(DBHANDLE * db, int \quad num)$ 实现。

本次测试利用大量数据的操作验证数据库操作结果的正确性，操作流程如下：

1. 随机存储 num 条数据（key 和 value 服从对应关系 F_1）；
2. 读取全部 num 条数据，比对是否与插入结果相同；
3. 修改全部 num 条数据（修改后 value 服从对应关系 F_2）；
4. 读取全部 num 条数据，比对是否与修改值相同；
5. 随机删除 num/5 条数据，随机插入 num/5 条数据（key 和 value 服从对应关系 F_2）；
6. 随机读取 num/2 条数据，比对是否满足对应关系 F_2；
7. 记录上述过程中比对错误的次数，返回结果。

在试验过程中，分别选取了 num 取 10000,100000,1000000 的情况进行测试，测试结果全部正确。

4.1.2 复杂操作正确性测试

注：为了简化试验，该项测试中统一使用节点大小为 64。测试过程由文件 *corr_test.cpp* 中的函数 $complicated\_corr\_test(DBHANDLE * db, int \quad num)$ 实现。

本次测试模拟了用户可能进行的复杂操作，验证过程中数据库操作结果的正确性，操作流程如下：

1. 向数据库写 num 条记录（key 和 value 满足对应关系 F_1）；

2. 通过关键字读回 num 条记录；

3. 执行下面的循环 num*5 次：

   (a) 随机读一条记录，比对结果是否满足对应关系 F_1；

   (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，比对结果是否满足对应关系；

   (d) 每循环 17 次，随机替换一条记录为新纪录，但替换内容也满足对应关系 F_1；

在试验过程中，分别选取了 num 取 100,1000,10000 的情况进行测试，测试结果全部正确。

4.2 时间性能测试

在数据库的运行中，影响其各种操作所需要的运行时间的主要因素有数据容量、B+ 树节点大小、数据分散程度、数据类型等方面。为了检验和分析数据库的时间性能，设计了以下测试项目：

4.2.1 复杂操作测试

注：为了简化试验，该项测试中统一使用节点大小为 64。测试过程由文件 *time_test.cpp* 中的函数 $complicated\_time\_test(DBHANDLE * db, int nrec)$ 实现。

如同老师提供课程设计的讲解 ppt 中的“性能测试”过程一样，“复杂操作测试”流程为：

1. 向数据库写 nrec 条记录

2. 通过关键字读回 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纪录，在连续两次替换中，一次用同样大小的记录替换，一次用比以前更长的记录替换。

4. 将此进程写的所有记录删除，每删除一条记录，随机地寻找 10 条记录；

5. 记录全过程总共用时 *Time*；

结果与分析

分别取 nrec 为 1000, 2000,5000,10000, 100000 进行测试，得到如下结果：

nrec 数值	1000	2000	5000	10000	100000
Time/s	0.167	0.414	4.571	60.660	973.212

表 1: 复杂操作用时

由以上结果可以看出，随着数据量的增长，操作用时产生了超线性的增加，这是因为多方面原因造成的：

- 总数据量的增加，使得树的高度增加，之后操作的预期自树顶至叶结点的下降次数增加；
- 总数据量的增加，使得在插入和删除操作之后，b+ 树递归上溢和下溢的预期次数增加；
- 总数据量的增加，使得替换数据时，数据库查找新数据应该放置的数据文件偏移量所需时间增加；
- 总操作次数的增加，使得数据库尝试空间重利用时，寻找对应存储空间（mrySpace）的预期时间增加；

除了上述列举原因外，还有很多数据结构和操作系统方面的原因导致了测试时间与操作次数的超线性关系。而这也符合设计数据库之前的预期和数据库的理论特性。由于测试流程中涉及多项操作类型，不同类型受数据量的影响可能不同，详细的单项测试和比对测试见后文中的测试项目内容。

4.2.2 时间性能与数据量关系测试

注：为了简化试验，该项测试中统一使用节点大小为 64。测试过程由文件 *time_test.cpp* 中的函数 $volume\_nodesize\_time\_test(DBHANDLE * db, intnrec)$ 实现。

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 nrec 条记录，记录写入所需总时间 $Time_1$；
2. 通过关键字读取 nrec 条记录，记录读取所需总时间 $Time_2$；
3. 修改数据库中全部 nrec 条记录，记录修改所需总时间 $Time_3$；
4. 随机删除数据库中 nrec/5 条记录，记录删除所需总时间 $TIme_4$；

结果与分析

分别取 nrec 为 10k, 100k, 500k, 1000k 进行测试，得到如下结果：

nrec 取值	10k	100k	500k	1000k
$Time_1/s$	0.238	2.196	11.142	22.522
$Time_2/s$	0.024	0.306	1.627	3.427
$Time_3/s$	0.158	12.870	453.803	2058.04
$Time_4/s$	0.050	3.295	62.681	141.642
$Time_{total}/s$	0.470	18.667	529.253	2225.613

表 2: 操作时间与数据量关系

分析上述表格展示数据，可得到以下的结果：

- 数据库的插入新数据和查找数据操作，所需时间和数据库中的数据总量近似表现为线性关系；
- 数据库的替换数据和删除数据操作，所需要的时间和数据库中的数据总量表现为超线性关系；
- 对于删除和替换操作，分别计算该相对值:

$$单次操作用时变化率\ P_i = 操作用时变化量/数据总量变化量 = \frac{T_{i+1}}{T_i} * \frac{N_i}{N_{i+1}}$$

$$其中，T_i\ 和\ N_i\ 分别为第\ i\ 次测试中的操作用时和数据总量\ nrec$$

可得到如下计算结果：

	P_1	P_2	P_3
删除操作	6.59	3.81	1.13
替换操作	8.15	7.05	2.27

表 3: 时间变化与数据量变化的比值

由于 $i = 1, 2, 3, 4$ 分别对应 *nrec* 为 10k, 100k, 500k, 1000k 时的测试，所以由该结果可以推测，当数据库中的数据较少时，数据库的删除和替换操作用时与数据库中的数据量呈明显的超线性关系，但当数据库中的数据逐渐增多，非线性逐渐减弱。在此次试验中，当数据库的总数据量到达 100 万以上时，删除操作用时与数据量变化已表现为近似的线性关系，而替换操作用时与数据量的关系也逐渐接近线性。

结合上述的结果，我们可以分析：

- 向空数据库中插入数据和查找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；
  - 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，其中，空间重利用的查找用时会随着数据量的增加而增加，而树在删除数据后的递归下溢操作也会随着树高而剧烈增加；

- 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，更详细的结果见下一小节 4.2.3 时间性能与节点大小关系测试部分。

4.2.3 时间性能与节点大小关系测试

注：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 *nodesize*。测试过程由 *time_test.cpp* 中的函数 $volume\_nodesize\_time\_test(DBHANDLE * db, int \quad num)$ 实现。

结果与分析

分别取 nodesize 为 32, 64, 128, 256, 512 进行测试，得到如下结果：

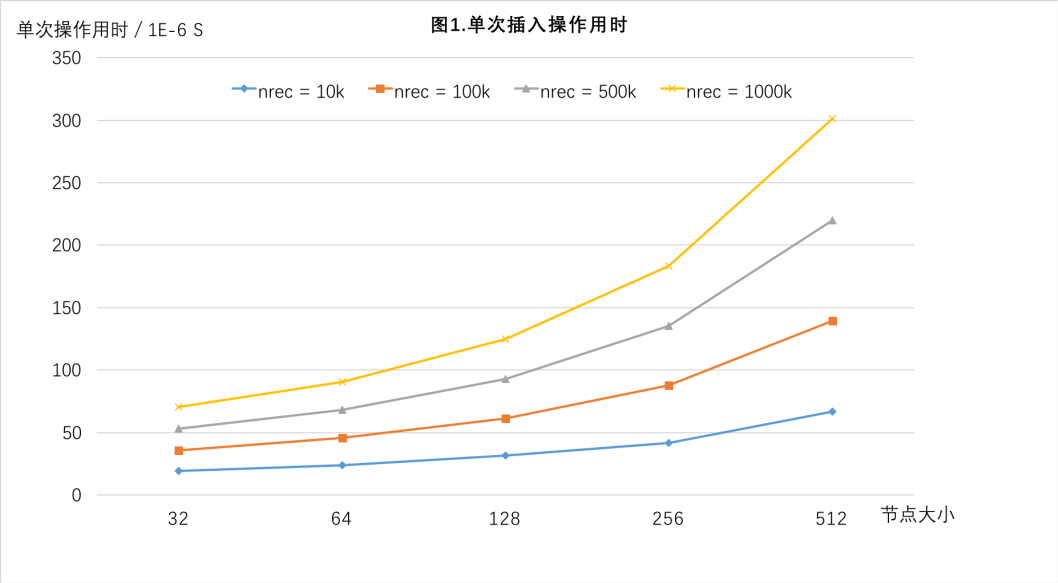
1. 插入数据用时与节点大小关系

nrec \ nodesize	32	64	128	256	512
10k	0.191	0.238	0.314	0.416	0.669
100 k	1.678	2.196	3.004	4.635	7.260
500k	8.628	11.142	15.604	23.657	40.074
1000k	17.512	22.522	32.118	48.027	81.629

表 4: 插入操作总用时 (s)

nrec \ nodesize	32	64	128	256	512
10k	19.1	23.8	31.4	41.6	66.9
100 k	16.78	21.96	30.04	46.35	72.60
500k	17.25	22.28	31.33	47.31	80.15
1000k	17.51	22.52	32.12	48.03	81.63

表 5: 单次插入操作用时 (μs)



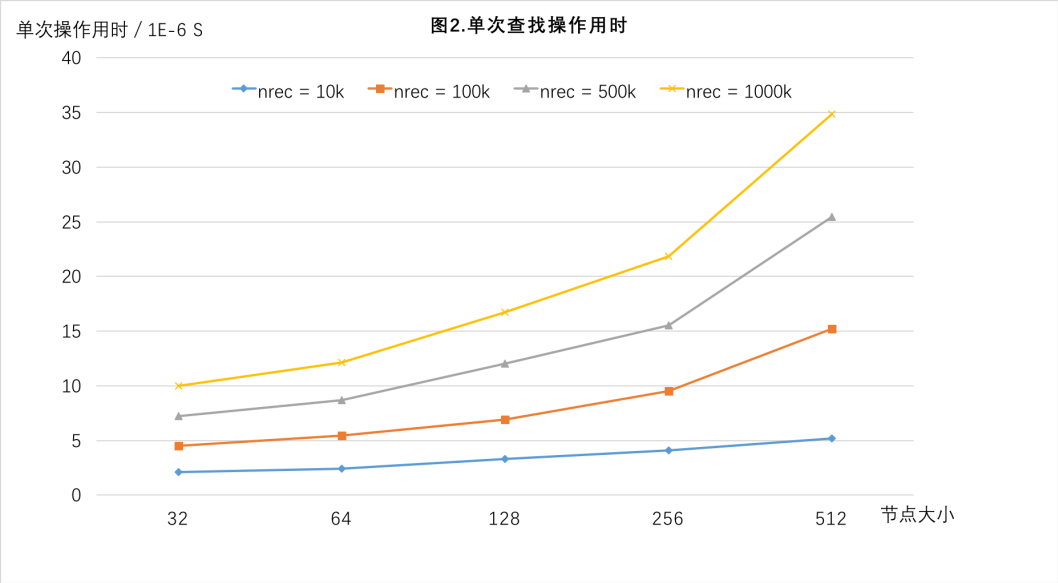
2. 查找数据用时与节点大小关系

nrec \ nodesize	32	64	128	256	512
10k	0.021	0.024	0.033	0.041	0.052
100 k	0.240	0.306	0.360	0.545	1.030
500k	1.370	1.627	2.564	2.989	5.703
1000k	2.765	3.427	4.685	6.320	9.442

表 6: 查找操作总用时 (s)

nrec \ nodesize	32	64	128	256	512
10k	2.1	2.4	3.3	4.1	5.2
100 k	2.40	3.06	3.60	5.45	10.03
500k	2.74	3.25	5.13	5.98	10.21
1000k	2.77	3.43	4.69	6.32	9.42

表 7: 单次查找操作用时 (μs)



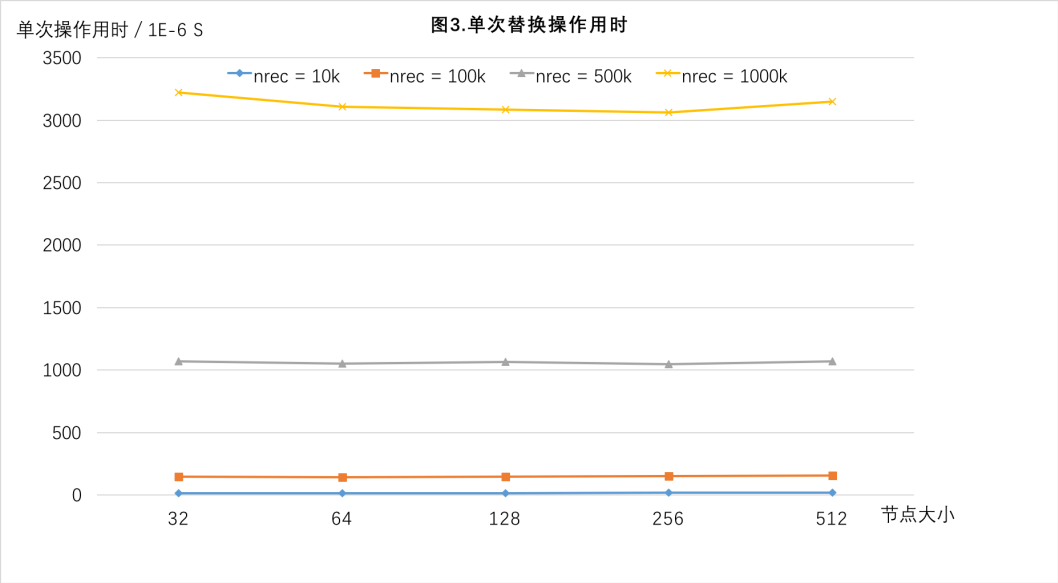
3. 替换数据用时与节点大小关系

nrec \ nodesize	32	64	128	256	512
10k	0.154	0.158	0.162	0.175	0.187
100 k	13.012	12.870	12.973	13.126	13.611
500k	462.945	453.803	459.579	445.744	456.547
1000k	2150.110	2058.040	2018.410	2018.440	2082.630

表 8: 替换操作总用时 (s)

nrec \ nodesize	32	64	128	256	512
10k	15.4	15.8	16.2	17.5	18.7
100 k	130.1	128.7	129.7	131.2	136.1
500k	925.9	907.6	919.2	891.5	913.1
1000k	2150.1	2058.0	2018.4	2018.4	2082.6

表 9: 单次替换操作用时 (μs)



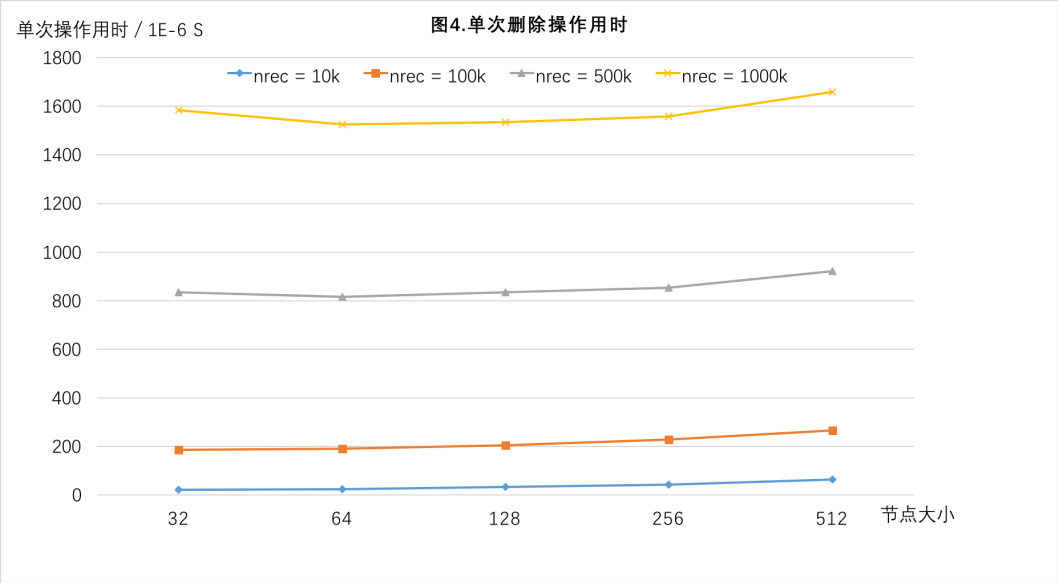
4. 删除数据用时与节点大小关系

nrec \ nodesize	32	64	128	256	512
10k	0.041	0.050	0.066	0.087	0.130
100 k	3.259	3.295	3.432	.3.704	4.019
500k	64.752	62.681	63.110	62.466	65.512
1000k	149.777	141.642	139.82	141.026	147.812

表 10: 删除操作总用时 (s)

nrec \ nodesize	32	64	128	256	512
10k	20.5	25.0	33.0	43.5	65.0
100 k	166.30	164.75	171.60	185.20	200.95
500k	647.52	626.81	631.10	624.66	655.12
1000k	748.90	708.20	699.10	705.15	739.05

表 11: 单次删除操作用时 (μs)



由上述的测试结果可以得出以下结论：

- 随着 B+ 树节点大小的增大，单次插入和查找操作的用时都出现了单调增加的趋势，因为插入和查找操作设计的磁盘和内容互动较少，所需用时基本都由 B+ 树中的操作产生，结合 B+ 树的特性，我认为这种单调的增长是由于以下 B+ 特性决定：
  - 随着数据总量的增加，树的高度呈对数级别的增加；
  - 随着数据总量的增加，节点未满时，查找节点中键的用时呈线形增加；

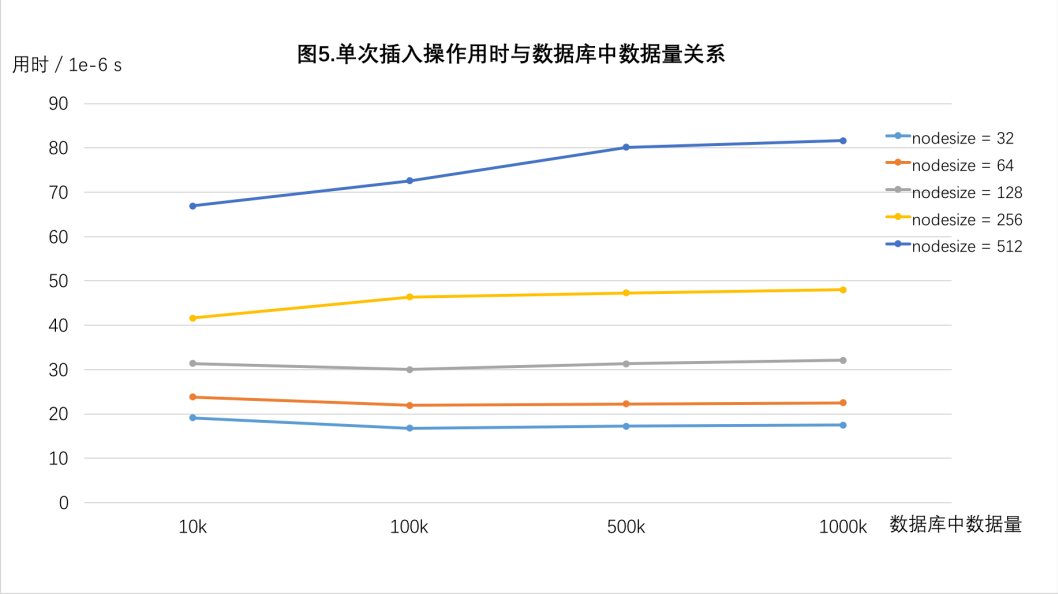
  因此，在节点大小增加的时候，后者所需要的线性查找时间明显增加了，而前者需要的对数查找时间尽管有一定的减少（相同数据量，节点越大，预期树高越低），但是总用时仍体现出增加的趋势；
- 随着 B+ 树节点大小的增大，单次替换和删除操作的用时并未体现出明显的变化趋势，由于替换和删除操作均涉及复杂的内存和文件操作，同时考虑到 B+ 树的性质，我认为该实验结果由以下特性决定：
  - 随着节点大小的增大，查找对应子节点的位置所需时间有了增加（理由同插入和查找用时的变化趋势）；

- 考虑到删除操作中的“下溢”过程，树高的降低体现出了更大的效益（预期需要的递归次数下降）；
- 相较于尝试空间重利用所需要的时间（查找 mrySpace 类的指针链表），上述发生在树结构中的操作用时比重很低；
- 相较于替换过程中经常发生的更新缓存写入文件所需要的时间，上述发生在内存中的操作用时的比重很低；

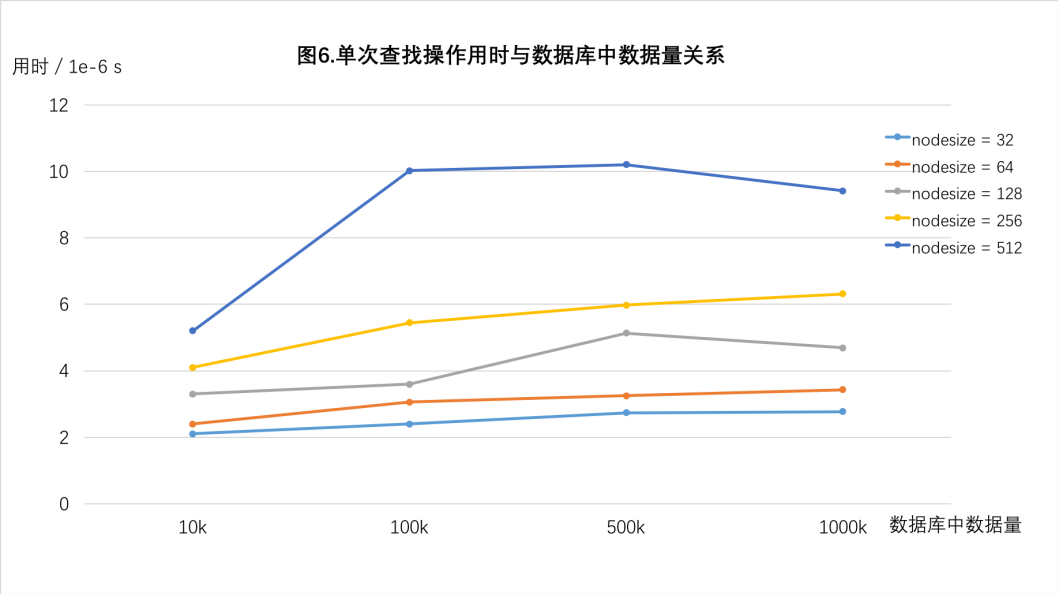
综上所述，对于替换和删除操作，单次操作的用时更多地取决于发生在保持内存磁盘一致性和空间重利用两点中的用时，而这些都与树的节点大小无关，所以随着树节点的增大，单次删除和替换操作的用时并未体现出明显的相关性。

此外，为支持 4.2.2 时间性能与数据量关系测试部分中得出的初步结论，根据上述试验数据，还绘制了不同操作类型单次用时与数据库中数据量的关系，展示如下：

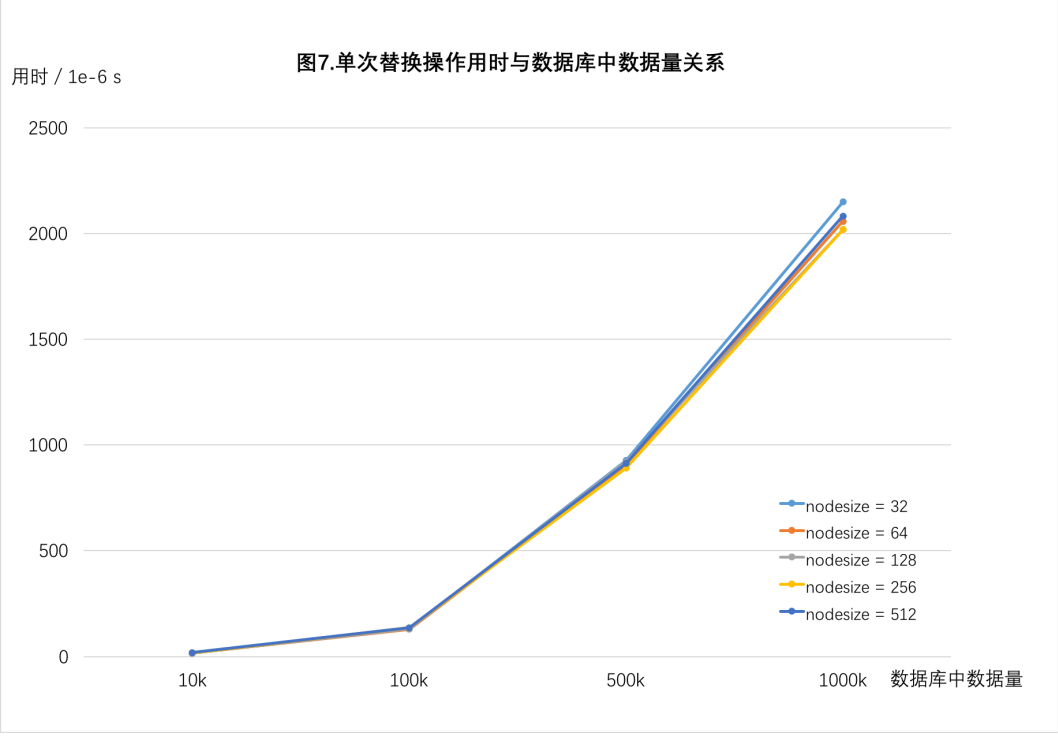
- 单次插入操作用时与数据库数据量关系



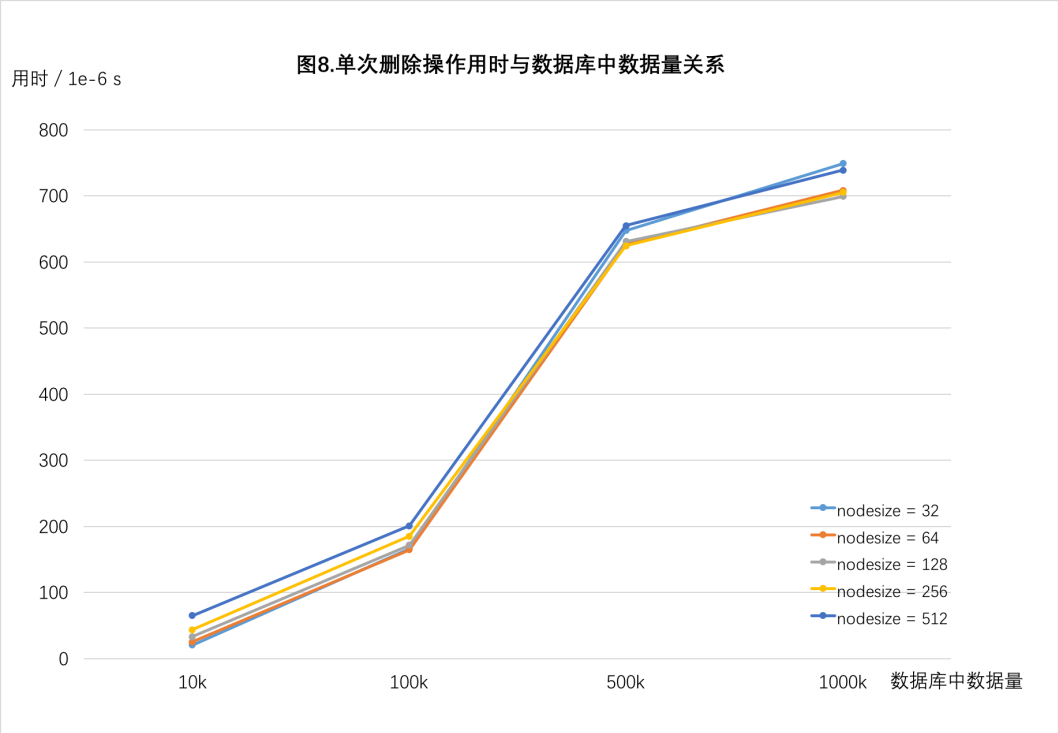
- 单次查找操作用时与数据库数据量关系



- 单次替换操作用时与数据库数据量关系



- 单次删除操作用时与数据库数据量关系



上述四幅图表展示的结果，验证了 4.2.2 时间性能与数据量关系测试部分中的结论，并得到了进一步的结论，总结如下：

- 单次插入和查找操作的用时更多的取决于树节点的大小，而对数据库中总数据量的变化不敏感，未表现出明显的相关性；
- 单次替换和删除操作的用时更多的取决于数据库中数据总量，随着数据量的增加，单次替换和删除操作用时都表现出了明显的增加；
- 随着数据量的增加，单次删除操作的用时逐渐稳定，而单次查找操作的用时稳定趋势不明显；

上述结果的可能原因已在 4.2.2 时间性能与数据量关系测试和 4.2.3 时间性能与节点大小关系测试进行了推理和分析，在此不再赘述。

4.2.4 缓存对时间性能的影响

注：为了简化试验，该项测试中统一使用节点大小为 64。测试过程由 *time_test.cpp* 文件中的函数 $buffer\_time\_test(DBHANDLE * db)$ 实现。其中，对缓存容量的修改需要在 *database.h* 文件中修改全

局变量 bufferSize，若不需要缓存，则修改其等于 1 或者在功能代码中修改即可（经过重构，这一修改后的代码也被提供在了 *database.h* 中，但处于被注释的状态，有兴趣可以研究细节）。

本次测试仿真用户操作，执行如下操作循环 10000 次：

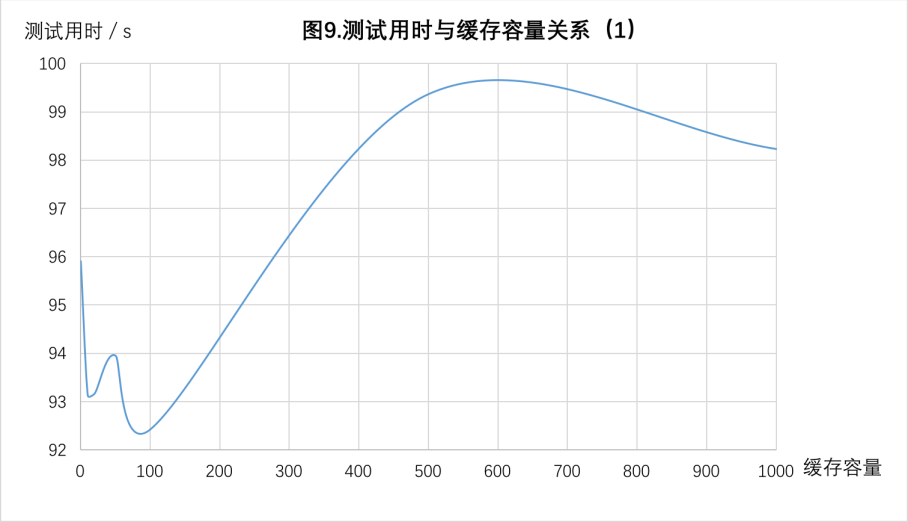
1. 随机插入 50 条记录；
2. 随机修改 20 条记录；
3. 随机查找 20 条记录；
4. 随机删除 20 条记录；
5. 随机查找 20 条记录。

结果与分析

测试过程中，分别取缓存容量为 0（不使用缓存）10，20，50，100，500，1000 进行测试，得到如下结果：

缓存容量	0	10	20	50	100	500	1000
测试用时/s	95.907	93.123	93.164	93.948	92.420	99.371	98.234

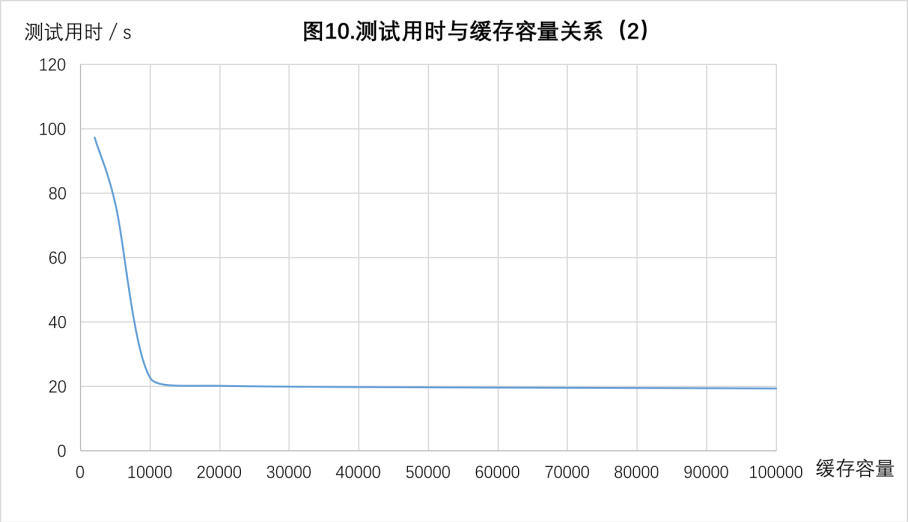
表 12: 测试用时与缓存大小关系 (1)



此外，为了说明 buffer 的存在可以提高时间性能的根本原因，实验中还设置了缓存容量为 2000，5000，10000，20000，100000 的非正常情况，得到的结果如下：

缓存容量	2000	5000	10000	20000	100000
测试用时/s	97.271	76.469	22.585	20.166	19.344

表 13: 测试用时与缓存大小关系 (2)



从上述展示结果可以发现，相较于没有缓存的情况，选取适当大小的缓存可以明显提高数据库的时间性能，特别地，当缓存容量增大到可以与总操作次数的量级相比较时，数据库的大多数操作都会直接和内容中已经存在的缓存数据互动，而不需要读写文件，此时的时间性能甚至可以提高 5 倍。但是在实际使用中，因为内存空间的限制和数据安全方面的考虑，使用过大的缓存是不实际的，但测试结果也已经充分说明了对于数据读写的功能方面，内存的性能是远远优于硬盘的。总而言之，选取适当大小（该试验中，该最优大小应在 100 左右），可以明显提高数据库的时间性能，并且保持对内存空间的合理使用。

4.3 空间性能测试

在数据库中的运行中，影响其生成的文件大小的主要因素有数据容量、空间重利用效率、编码形式、数据类型等方面。为了检验和分析数据库的空间性能，设计了以下的测试流程：

4.3.1 空间重利用效率测试

注，以下试验过程中，统一使用节点大小为 64，测试过程由文件 *space_test.cpp* 中的测试函数 *reuse_space_test*$(DBHANDLE*db)$ 实现，但请注意，打开和关闭重利用功能需要在 *database.h* 文件中分别删除第 336 行和第 337 行代码，这是因为尽管可以在调用数据库的参数中添加这一开关，但是作为默认必须的功能，仅为了一次测试而提供这样的参数开关是不合理的，在权衡之后，经过代码重构，仅需要修改头文件中一行代码即可开启和关闭这一功能。

测试流程如下：

1. 随机存储 500k 条数据；
2. 随机删除 200k 条数据；
3. 随机存储 500k 条数据；

结果与分析

	无空间重利用	有空间重利用	差占重利用大小百分比
数据 (.dat) 文件大小/kb	3253	2826	15.12%
索引 (.idx) 文件大小/kb	8545	8314	1.45%

表 14: 文件大小与空间重利用的关系

由上述结果可以看出，通过空间重利用，数据文件的大小被显著地压缩了，这是符合我们的预期的，这说明了空间重利用对优化数据库空间性能的显著作用，此外，有空间重利用和无空间重利用两次测试所用时间基本持平。因为数据库重利用类中利用了链表结构，可能减慢了重利用的效率，如果可以进行适当优化，空间重利用机制的加入甚至会明显提高数据库的时间性能。

此外，由于我们采用了关闭数据库后重写索引文件的方式，所以索引文件始终都处于最紧密的书写方式，不受空间重利用机制的影响，除非修改索引的组织形式、更换编码方式或者进行二次压缩，否则，索引文件的大小已经无法进行优化。

不难说明，空间重利用的效率还与节点大小、删除数据大小均匀度以及单个数据长度相关，具体的来说，在节点越大、删除量越多、单个数据越长的情况下，空间重利用会更好地提高数据库的空间性能。

4.3.2 空间性能与数据量关系测试

注，以下试验过程中，统一使用节点大小为 64，测试过程由文件 *space_test.cpp* 中的测试函数 $volume\_space\_test(DBHANDLE * db, int \quad num)$ 实现。

测试流程如下：

1. 不重复地存储 num 条数据；
2. 随机删除 num/2 条数据；
3. 不重复地存储 num 条数据；

结果与分析

测试过程中，分别取 num 为 1k,10k,100k,1000k 得到如下结果：

num 取值	1000	10k	100k	1000k
索引文件 (.idx) 大小/kb	12	134	1547	17332
数据文件 (.dat) 大小/kb	4	50	662	6466

表 15: 文件大小与数据量的关系

由上述试验结果可知，经过特定次数的插入数据、删除数据、插入数据操作，形成的索引文件和数据文件大小由以下特点：

- 索引文件和数据文件大小比例基本不变（试验中约为 3:1）；
- 索引文件和数据文件大小关于操作的次数呈近似的线性增长的关系；

需要强调的是，为了保证删除操作的成功，被删除键的大小处于插入键的范围内，而如果没有这一限制，那么数据文件和索引文件的大小会更加明显地受空间重利用和删除键分布的均匀程度等因素的影响。

5 待改进之处

5.1 优化空间利用

尽管在数据库的设计过程中，已经利用空间重利用机制对空间利用进行了优化，但仍有可供改进之处：

- **优化数据布局**：当修改数据并重利用空间后，若新数据长度比该空间块长度短，则可以将之后数据块前移，以减少数据文件所需要的空间，但是该过程会改变数据对应的空间位置，所以只可在关闭数据库之后，添加一步重写数据文件和索引文件的过程，对数据文件进行压缩；
- **优化编码方式**：在设计的数据库中，为了提高数据库的可移植性和可调试性，使用了可读字符的数据操作方式，但若使用其他编码方式，可进一步优化数据文件的空间大小；
- **压缩数据文件**：在数据库关闭之后，默认的数据文件会以可读文本的形式直接存储，这会需要较大的存储空间，如果在数据库关闭之后，对数据文件进行压缩处理（如使用 haffman 压缩等方式），可进一步压缩数据文件大小，但是会需要下次使用该文件时先进行解压缩操作。

5.2 优化时间性能

尽管在数据库的设计过程中，已经在许多方面考虑到了对时间性能的优化，但仍有可供改进之处：

- **更换数据结构**：该数据库默认使用了一般的 B+ 树作为底层的数据结构，但因为数据库的调用接口和内部实现的接口都不依赖于具体底层数据结构的性质，所以若更换更高效的数据结构，可能会进一步优化数据库的时间性能；
- **优化自定义结构**：数据库的设计中定义了缓存 (buffer)，缓存单元 (Token)，内存空间 (mrySpace)

和空间链表 (emptySpace 和 filledSpace) 等自定义结构，其中包装了 Map 和链表等数据结构，优化这些自定义结构体的实现（如优化链表的递归查找过程），可以进一步优化数据库的时间性能；

- **优化数据结构操作**：在实现 B+ 树的过程中，许多操作的时间性能都可以进一步优化，比如在删除内部结点时，有“多次旋转”和“融合子节点”等不同的方式，此外，处理“上溢”和“下溢”时，一般的递归过程可能也可以进一步优化，在不同情况下，不同方式可能会优劣不同，优化这些操作也很重要；

5.3 更通用的设计

在数据库的设计中，碍于时间和能力，数据库并不能应付所有可能的情况，数据库的通用性还可以进一步优化：

- **支持更多键值类型**：碍于时间限制，本数据设计默认支持了 <string, string> 类型的键值数据，但可利用模版编程使数据库支持更广泛的键值类型，但需要注意的是，键的数据类型应该可以支持大小的比对操作；
- **支持超大文件异步处理**：数据库本来设置了模式 B，使得所有的数据结构操作都直接在文件中实现，但是测试后发现性能实在难以满足需要，也碍于时间限制，只好将数据库对超大文件的支持作为之后的进一步工作；
- **跨平台移植**：尽管数据库文件编写中并未使用任何的平台依赖和编译器依赖的代码，但是或许是因为 ide 的特性限制，当代码在其他平台测试时，偶尔会出现 warning 的情况，尽管不影响功能实现，但是说明文件代码的通用性和规范性需要进一步改善。

5.4 软件工程优化

在数据库的开发过程中，因为软件工程素养方面的不足，出现了大量低效率开发的现象，说明在软件工程优化方面，还需要改进，具体为：

- **优化单元测试**：在开发过程中，本人利用 visual studio 自带单元测试功能进行了少量初步情况的监测，但代码覆盖率和检测状况覆盖率仍需要进一步提升和优化；

- **优化代码质量**：因为代码文件经历了多次大规模的重写，也碍于时间限制和本人的能力水平，在代码的重构方面仍有许多可改进之处，从而进一步改善代码文件的组织架构、可读性和健壮性等；
- **优化模块组织**：本数据库文件中，基础数据结构和数据库的实现都简单包装在了两个头文件中，之后由测试代码直接调用，导致两个头文件过大，且不满足“定义与实现分离”的一般需求，另一方面，数据库中的自定义结构也可以单独构成模块文件，这些处理都可以进一步改善文件的模块组织。

综上所述，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。