Database 开发文档

515030910474　吕旭晖

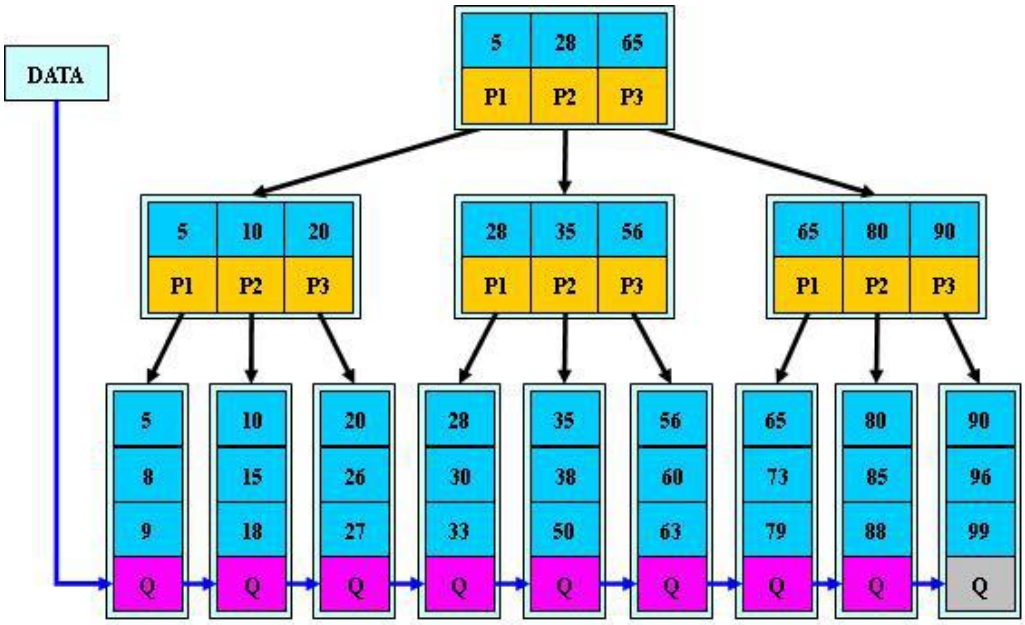
一、概述

这是使用 B+树这一数据结构开发而出的数据库，实现了数据库基本的增删改查的功能，数据库包含两个文件，一个是索引文件，用于储存 B+树作为数据库的索引，另一个文件是数据文件，用于保存数据。对这两个文件的操作分别属于两个类。该数据库还自带了测试功能，可以测试数据库在处理大量数据时的性能。除此之外，该程序也设计了一个简单的用户交互界面以方便操作。

二、数据结构

该数据库采用 B+树作为索引，B+树以节点为单位存储于索引文件中，当对数据库进行各种操作时，都要先在索引文件中找到对应的数据才能进一步操作。

B+树的结构如下图所示：



三、对索引文件的操作

索引文件中的一个 B+树节点由一下几个部分组成：

leaf	parent	left	right	key1	pos1	key2	pos2	.....

每个节点长度都是固定的，若采用的 B+树为 n 叉树，则该节点的大小为

1*1 + 4*3 + 8*(n + 1) = (21 + 8n)byte

节点的第一个位置用一个 char 字符来表示该节点是否是叶子节点，char=='1'时，该节点就是叶子节点，反之不是。

除了第一个位置以外，之后的每个位置存储的都是 4-byte 形式的 int 型对象，parent 处储存该节点的父节点在索引文件中的位置，left 储存与该节点相邻的左边节点在索引文件中的位置，right 则储存右边节点的位置，之后储存的是 n + 1 个（key，pos）对（这些对按递增次序排列，这是由每次取出该节点操作时操作有序而决定的），用于存储其子节点的 key 以及子节点在索引文件中的位置。若该节点的父节点不存在则 parent = -1，同理当其余节点不存在时（包括未被占用的子节点位置），他们所保存的值也是 -1。

对索引文件的操作，也即对数据库对应的 B+树的操作，对索引文件中的 B+树的操作并不是一次性将索引全部读出来，而是一个节点一个节点地读，通过如下函数，可以通过一个节点在索引文件中的位置找到该节点，并将其读出，实例化成一个 Node 对象，并返回一个指针。

- 获取节点：

```
Node* BplusTree::getNode(int pos) { ... }
```

—— pos: 节点在索引文件中的位置。

结构体 Node 是 B+树节点在内存中的存在形式，保存有该节点是否是叶子节点、父节点位置、左右节点位置等信息，还用一个 vector 来储存其子节点的信息，为保证该 vector 的递增次序，每次插入新子节点信息前都要先找到新插入的 key 应该处在的位置。

- 放回节点：

```
void BplusTree::putNode(Node* cur) { ... }
```

—— cur: 需要放回的节点。

由于这个操作没有涉及到文件位置，放回节点之前需要手动先将文件指针放到该节点应当处于的位置，这是不难做到的，放回节点后，原来文件中的信息将被覆盖，从而实现了信息的更新，保证了一致性，做完放回操作后，cur 占用的内存会被释放。

- 查找节点：

```
Node* BplusTree::searchNode(int key) { ... }
```

—— key: 需要找到的 key

找到该 key 所在的叶子节点，并通过之前的 getNode()函数将该节点实例化成一个 Node

对象，从而可进行进一步操作。若所要查找的对象不存在，则会返回一个空指针。

- 插入节点：

```
bool BplusTree::insert(int key) { ... }
```

—— key：需要插入的 key。

该函数会从索引文件中的根节点开始，逐个节点向下查找直到到达该 key 应处的叶子节点，之后，再在叶子节点之中插入这个 key，如果在查找过程中找到了相同的已经存在的 key，则会返回 false。在完成插入操作之后，还要考虑新插入的 key 是否会对父节点造成影响以及考虑节点的上溢问题。

- 解决上溢：

```
void BplusTree::solveOverflow(Node* cur) { ... }
```

—— cur ：之前完成插入操作的节点。

在每次完成插入操作之后都会调用这个函数，若达到上溢条件，则会发生上溢。上溢主要分两种情况，若不是根节点发生上溢，则该节点一分为二并在其父节点中插入一个新键值，若是根节点，则这个根节点一分为二之后会还会生成一个新的根节点，从而全树高度加一。与此同时，解决掉一个节点地上溢问题，可能会引起其父节点的上溢，也需要依次解决。

- 删除操作：

```
bool BplusTree::remove(int key) { ... }
```

—— key：需要删除的 key。

先调用 searchNode()函数，若没有找到 key，则返回 false，若找到，则删除之，与插入操作类似，删除之后也要考虑对父节点的影响，同时要考虑下溢。

- 解决下溢：

```
void BplusTree::solveUnderflow(Node* cur) { ... }
```

—— cur：之前完成删除操作的节点。

完成删除操作之后调用这个函数，若满足下溢条件，则发生下溢。解决下溢的手段也主要是两种，第一种是向该节点左边或者右边的节点借 key，若左边及右边的子节点树都已经是最小值，则采用另一种方法，即合并，将该节点合并到左边节点或右边节点。解决下溢之后可能引起父节点的下溢，也需要解决。当根节点发生下溢时，全树高度会减一。

- 替换操作：

```
bool BplusTree::replace(int key, int newKey) { ... }
```

—— key：需要替换的键值;
—— newKey：所要换成的新值。

先用 searchNode()函数找到 key 所在的位置，若不存在则返回 false，否则即可进行替换，只更换 key 的值，并不变动其所对应的 pos 的值。

- 其他操作：

```
void BplusTree::save() { ... }
```

将 B+树的根节点位置以及 count 信息保存进索引文件后关闭索引文件。

```
void BplusTree::load() { ... }
```

读取索引文件。

```
void BplusTree::clear() { ... }
```

清空索引文件，放入一个空的根节点。

三、对数据文件的操作

数据文件中并不保存数据的 key 的信息，只保存 key 所对应的信息，本数据库所储存的信息和其 key 一样，也是 int 型的对象，在数据文件中以 4-byte 形式存储。

由于存储信息较少，对数据文件的操作相比索引文件来说也要简单的多，下列是对其进行操作的一些函数。

```
void Database::open() { ... }
```

//打开数据文件，若不存在则新建一个数据文件，同时读取索引文件。

```
void Database::close() { ... }
```

//关闭数据文件同时保存索引。

```
bool Database::store(int key, int data) { ... }
```

//保存一条信息。若 key 在 B+树中以存在，则返回 false。

```
int Database::fetch(int key) { ... }
```

//根据键值查找信息，若未找到则返回-1。

```
bool Database::remove(int key) { ... }
```

//在数据文件中删除 key 所对应的数据，这里的删除不是真的删除，只是把之前保存的

该数据位置信息抹除掉，从而无法再从数据文件中找到该信息。

```
void Database::clear() { ... }
```

//清空数据库，也会清空掉所有闲置的信息。

四、性能测试

通过调用数据库的性能测试函数来进行性能测试，数据库有两个测试函数：

- 正确性测试：

```
void Database::test(int nerc) { ... }
```

调用该函数可进行数据库的正确性测试，测试流程如下：

步骤：

(1) 向数据库写*nrec*条记录。
(2) 通过键值读回*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，添加一条新记录并读回这条记录。
  (d) 每循环17次，随机替换一条记录为新记录。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) 将此子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。

在向数据库进行读写操作的同时，有一个参照用的 map，也同样对其进行相应的读写操作，每次从数据库中读取记录时也从 map 中读取之，二者进行比较即可验证数据库的正确性。

测试结果如下图所示：

```
Input the data size:1000000
choose the mode:1.correctness  2.performance
1
write test...
pass: put in 631968 datas.
read test...
pass:631968 datas get.
compound test...
pass.
delete test...
pass.
Total time: 1297
```

在数据量为 1000000 量级时测试全部通过，可见该数据库的正确性是有一定的保证的。

- 性能测试：

性能测试的步骤和正确性步骤类似，区别是性能测试中默认正确性已经保证，故没有 map 与之比较，注重测试性能。

测试结果如下：

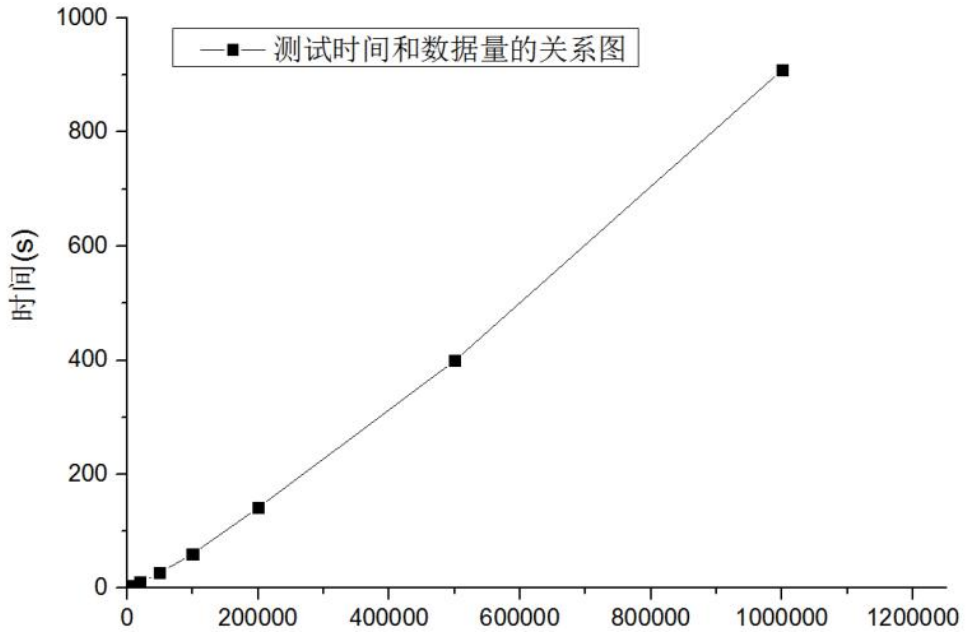
```
Input the data size:1000000
choose the mode:1.correctness  2.performance
2
Total time: 909

Input the the order:
```

在数据量为 1000000 量级时，跑完全部测试花了整整 15 分钟，可见该数据库在性能方面做的并不是太好仍有改进的余地。

- 在不同数据量下的性能测试结果：

数据量	测试时间（s）
10000	4
20000	10
50000	27
100000	60
200000	141
500000	399
1000000	909

绘制折线图：



由折线图可知，测试数据量与总时间大致呈线性关系，符合预期。

五、总结

该数据库虽然实现了一个数据库最基本的增删改查操作，正确性基本有所保证，但还有许多值得改进的地方，最主要的实在性能上面，每次读取 B+树的节点之后都要花一段时间来重建一个 Node 对象，可见这种将 B+树写成文件的方式并不是很理想，可以考虑使用硬盘树的方式加以改进。在一些细节方面该程序也有所欠缺，比如若对程序进行操作之后不执行 close 命令就直接关闭命令行的话，回导致树的根节点信息无法保存，从而再次打开时数据错乱，虽然这样的错误是人为引起的，但其实可以对程序加以改进以消除这样的错误发生的可能性。