实验二

SA20225085 朱志儒

实验目的

实现一个简易的数据库存储与缓存管理器，实现对磁盘进行字节流操作的读写函数，实现基于哈希+列表的 page 到 frame 的映射，使用 trace 文件 data-5w-50w-zipf.txt 验证项目，更改 DEFBUFSIZE 对比分析实验结果。

实验环境

操作系统：Windows 11

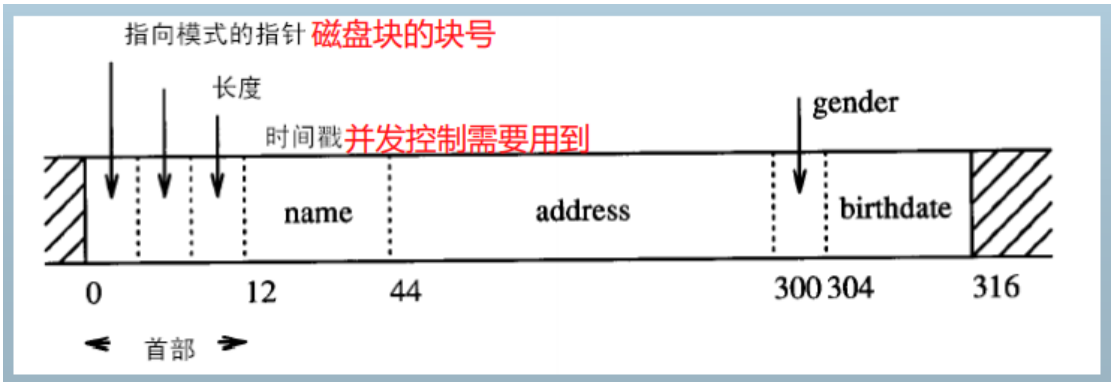
CPU：AMD Ryzen 5 3600X

RAM：32 GB

IDE：Visual Studio 2019，Windows SDK 10.0

实验内容

依据实验文档，创建数据库文件 data.dbf，文件头部存储 page 指针，指向各个 page，各个 page 只存储数据。page 头部存储块号、时间戳以及指向各个记录的指针，记录格式如下：



每条记录的长度为 316 字节，每个 page 的大小 (FRAMESIZE) 为 4096 字节，$4096 \div 316 \approx 12$，即每个 page 可以存储 12 个记录数据，创建新 page 的代码如下：

```
bFrame CreatePage(int num) {
 bFrame frame;
 int i = 0;

 // 数据块号
 auto ch = static_cast<char*>(static_cast<void*>(&num));
 copy(ch, ch + sizeof(ch), frame.field);
 i += sizeof(ch);

// 时间戳
unsigned int btimestamp = 0;
ch = static_cast<char*>(static_cast<void*>(&btimestamp));
copy(ch, ch + sizeof(ch), frame.field + i);
i += sizeof(ch);

// 偏移量表
unsigned short int offset[FRAMESIZE / RECORDSIZE];
int BLOCK_HEAD = sizeof(btimestamp) + sizeof(offset);
for (int t = 0; t < FRAMESIZE / RECORDSIZE; t++) {
 int toffset = BLOCK_HEAD + t * RECORDSIZE;
 ch = static_cast<char*>(static_cast<void*>(&toffset));
 copy(ch, ch + sizeof(ch), frame.field + i);
 i += sizeof(ch);
};

for (int j = 0; j < FRAMESIZE / RECORDSIZE; ++j) {

 // 写入记录
 char* p_schema = new char[4];
 copy(p_schema, p_schema + sizeof(p_schema), frame.field + i);
 i += sizeof(p_schema);
 delete p_schema;
 unsigned int timestamp = num;
 ch = static_cast<char*>(static_cast<void*>(&timestamp));
 copy(ch, ch + sizeof(ch), frame.field + i);
 i += sizeof(ch);
 unsigned int length = 316;
 ch = static_cast<char*>(static_cast<void*>(&length));
 copy(ch, ch + sizeof(ch), frame.field + i);
 i += sizeof(ch);
 char namePtr[32];
 copy(namePtr, namePtr + sizeof(namePtr), frame.field + i);
 i += sizeof(namePtr);
 char addressPtr[256];
```

```
45.  copy(addressPtr, addressPtr + sizeof(addressPtr), frame.field +
    i);
46.  i += sizeof(addressPtr);
47.  char genderPtr[4];
48.  copy(genderPtr, genderPtr + sizeof(genderPtr), frame.field + i)
   ;
49.  i += sizeof(genderPtr);
50.  char birthdayPtr[12];
51.  copy(birthdayPtr, birthdayPtr + sizeof(birthdayPtr), frame.fiel
   d + i);
52.
53. }
54. return frame;
55.}
```

CreatePage 函数在 FixNewPage 函数中被调用，以创建新的 page 并复制到 buffer 中。

依据文档要求创建包含 50000 个 page 的 data.dbf 文件，代码如下：

```
1.  // 调用 FixNewPage 50000 次以建立 data.dbf 文件
2.  for (int i = 0; i < MAXPAGES; ++i) {
3.   bmgr.FixNewPage();
4.  }
5.  bmgr.WriteDirtys();
```

BCB 的设计如下：

```
1. struct BCB {
2.  int page_id;
3.  int frame_id;
4.  int latch;
5.  int count;
6.  int dirty;
7.  int next;
8. };
```

与文档的设计不同之处在于 next 属性，文档中的 next 用于指向下一个 BCB，而我设计的 next 用于记录数据库访问该 BCB 对应 page 的 Time。BMgr 类的私有属性 Time 表示数据库对 page 操作的时间（初始值为 0），即每当数据库读取或写入 page 一次，Time 都会加 1。

依据文档的要求，采用 LRU 算法作为替换策略，我在实现该策略时，构建一

个最小堆（容器是 vector），堆中的元素就是 buffer 中 page 所对应的 BCB。建堆与整堆（调用 algorithm 库中的 make_heap 与 push_heap）时元素根据 BCB 的 next 属性进行比较，也就是说，next 越小的 page 最近没有被访问，而 next 越大的 page 则最近经常访问。每当需要选择被替换的 page 时，选取堆顶的 BCB 所对应的 page 进行替换，对堆的修改操作（调用 algorithm 库中的 pop_heap）是，将堆顶元素 A 与最后一个元素 B 交换位置，再将 A 从堆中删除，最后再整堆。如果有新的 page 被读入 buffer，则需要将该 page 对应的 BCB 加入堆的末尾，再整堆。上述的建堆、整堆操作会在 BMgr 类的构造函数、FixPage 函数、FixNewPage 函数、UnfixPage 函数和 RemoveLRUEle 函数中被调用。

类 BMgr 的私有属性如下：

```
1. class BMgr {
2. private:
3.  int ftop[DEFBUFSIZE];
4.  map<int, vector<BCB*>> ptof;
5.  vector<BCB*> bcbList;
6.  vector<BCB*> LRUList;
7.  DSMgr dsmgr;
8.  int Time;
9.  bFrame* buf;
10.};
```

其中，ftop 表示 frame_id 到 page_id 的转换，ptof 表示 page_id 到 frame_id 的转换，其结构是 HashMap+Vector，key 由 Hash(int page_id)函数得到，对应的 value 是 vector，表示 key 相同的 BCB 组成的列表。bcbList 表示存储在 buffer 中的 page 所对应的 BCB 列表，LRUList 就是上面所述的最小堆。Time 就是上述的表示数据库对 page 操作的时间，buf 就是 buffer 缓冲区。

类 BMgr 的构造函数如下：

```
1. BMgr::BMgr():bcbList(DEFBUFSIZE) {
2.  // 将 LRUList 变成小根堆，将 Time 作为排序依据
3.  make_heap(LRUList.begin(), LRUList.end(), cmp);
4.
5.  for (int i = 0; i < DEFBUFSIZE; ++i) {
6.   bcbList[i] = nullptr;
7.   ftop[i] = -1;
8.  }
9.
10. // 初始化 buf 缓存区
11. buf = new bFrame[DEFBUFSIZE];
12. cout << "buffer 大小：  \t" << DEFBUFSIZE << endl;
13.
14. // 打开数据库文件
15. dsmgr.OpenFile("data.dbf");
16.
17. Time = hit = miss = Incount = Outcount = 0;
18.}
```

构造函数的功能是初始化 LRUList、bcbList、ftop、buf、Time，打开数据库文件 data.dbf。

FixPage 函数如下：

```
int BMgr::FixPage(int page_id, int prot) {
 int frame_id;
 int key = Hash(page_id);

 // 在 ptof 中查找 page_id 是否存在，即 page_id 是否事先载入内存
 if (ptof.find(key) != ptof.end()) {
  for (int i = 0; i < ptof[key].size(); ++i) {
   if (ptof[key][i]->page_id == page_id) {
    frame_id = ptof[key][i]->frame_id;
    bcbList[frame_id]->count++;
    bcbList[frame_id]->latch = 1;

    // 统计命中次数
    //cout << "命中 " << frame_id << "frame" << endl;
    hit++;

    for (int k = 0; k < LRUList.size(); ++k) {
     if (LRUList[k]->page_id == page_id) {

      // 更新 BCB 的 Time 元素
      LRUList[k]->next = Time++;

      // 调整 LRUList
      push_heap(LRUList.begin(), LRUList.end(), cmp);

      return frame_id;
     }
    }
   }
  }
 }
 if (NumFreeFrames() == 0) {
  // 选择被替换的 frame
  frame_id = SelectVictim();
 }
 else {
  for (int i = 0; i < DEFBUFSIZE; ++i) {
   // 查找空闲的 frame
   if (bcbList[i] == nullptr) {
    frame_id = i;
    break;
   }
```

```
  }
}

// 统计未命中次数
miss++;
Incount++;

// 将 page 载入内存的 frame
buf[frame_id] = dsmgr.ReadPage(page_id);

// 添加新的 BCB 到 BCB 列表
bcbList[frame_id] = new BCB(page_id, frame_id, 1, 1, 0, Time++);

// 将新的 BCB 加入 LRUList
LRUList.push_back(bcbList[frame_id]);
push_heap(LRUList.begin(), LRUList.end(), cmp);
// 更新 ftop
ftop[frame_id] = page_id;
// 更新 ptof
if (ptof.find(key) == ptof.end()) {
 vector<BCB*> list;
 list.push_back(bcbList[frame_id]);
 ptof[key] = list;
}
else {
 for (int i = 0; i < ptof[key].size(); ++i) {
  if (ptof[key][i]->page_id == page_id) {
   ptof[key][i] = bcbList[frame_id];
   return frame_id;
  }
 }
 ptof[key].push_back(bcbList[frame_id]);
}

return frame_id;
}
```

FixPage 函数的功能：在 ptof 中查找 page_id 是否存在，即 page_id 是否事先载入内存，若存在，则统计命中次数，更新 BCB 的 next 属性，调整 LRUList（整堆）；若不存在且 buffer 已满，则选择被替换的 page；若 buffer 未满，则查找空闲的 frame。接着统计未命中次数，将 page 载入内存的 buffer，将该 page 对应的 BCB 加入 BCB 列表（bcbList）和 LRUList（加入后需整堆），更新 ftop 和 ptof。

FixNewPage 函数如下：

```
1. NewPage BMgr::FixNewPage() {
2.  int frame_id, page_id;
3.
4.  for (int i = 0; i < MAXPAGES; ++i) {
5.   // 找到一个空闲的 page
6.   if (dsmgr.GetUse(i) == 0) {
7.    page_id = i;
8.    dsmgr.SetUse(i, 1);
9.    dsmgr.IncNumPages();
10.   break;
11.  }
12. }
13. if (NumFreeFrames() == 0) {
14.  // 选择被替换的 frame
15.  frame_id = SelectVictim();
16. }
17. else {
18.  for (int i = 0; i < DEFBUFSIZE; ++i) {
19.   // 查找空闲的 frame
20.   if (bcbList[i] == nullptr) {
21.    frame_id = i;
22.    break;
23.   }
24.  }
25. }
26.
27. // 生成一个新的 page
28. bFrame nframe = CreatePage(page_id);
29.
30. // 将新的 page 复制到 buffer
31. copy(nframe.field, nframe.field + FRAMESIZE, buf[frame_id].field
    );
32.
33. // 添加新的 BCB 到 BCB 列表
34. bcbList[frame_id] = new BCB(page_id, frame_id, 1, 1, 1, Time++);
35.
36. // 将新的 BCB 加入 LRUList
37. LRUList.push_back(bcbList[frame_id]);
38. push_heap(LRUList.begin(), LRUList.end(), cmp);
39.
40. // 更新 ftop
41. ftop[frame_id] = page_id;
```

```
42.
43. int key = Hash(page_id);
44.
45. // 更新 ptof
46. if (ptof.find(key) == ptof.end()) {
47.  vector<BCB*> list;
48.  list.push_back(bcbList[frame_id]);
49.  ptof[key] = list;
50. }
51. else {
52.  for (int i = 0; i < ptof[key].size(); ++i) {
53.   if (ptof[key][i]->page_id == page_id) {
54.    ptof[key][i] = bcbList[frame_id];
55.   }
56.  }
57.  ptof[key].push_back(bcbList[frame_id]);
58. }
59.
60. return NewPage(page_id, frame_id);
61.}
```

FixNewPage 函数功能：找到一个空闲的 page 以获取 page_id，若 buffer 已满，则选择被替换的 page；若 buffer 未满，则查找空闲的 frame。接着构造一个新的 page，将新的 page 复制到 buffer，将新 page 对应的 BCB 加入 BCB 列表（bcbList）和 LRUList（加入后需整堆），最后更新 ftop 和 ptof。

UnfixPage 函数如下：

```
int BMgr::UnfixPage(int page_id) {

 // 根据 page_id 查找对应的 BCB
 BCB* bcb = ptof[Hash(page_id)][0];
 for (int i = 0; i < ptof[Hash(page_id)].size(); ++i) {
  if (ptof[Hash(page_id)][i]->page_id == page_id) {
   bcb = ptof[Hash(page_id)][i];
   break;
  }
 }

 bcb->count--;
 if (bcb->count == 0) {
  bcb->latch = 0;
 }

 // 更新 BCB 的 Time
 bcb->next = Time++;

 // 调整 LRUList
 push_heap(LRUList.begin(), LRUList.end(), cmp);

 return bcb->frame_id;
}
```

UnfixPage 函数功能：依据 page_id 在 ptof 中查找对应的 BCB，更新 BCB 中的 count、latch 和 next 属性，调整 LRUList（整堆）。

NumFreeFrames 函数如下：

```
int BMgr::NumFreeFrames() {
 return DEFBUFSIZE - LRUList.size();
}
```

NumFreeFrames 函数功能：获取 buffer 中空闲的 frame 数量。

SelectVictim 函数如下：

```
int BMgr::SelectVictim() {

  // 选择 LRUList 中第一个 BCB，即最近没有访问过的 frame
  BCB* ptr = LRUList.front();

  // 删除 LRUList 中第一个 BCB
  RemoveLRUEle(ptr->frame_id);

  // 如果为 dirty 则需写回磁盘
  if (bcbList[ptr->frame_id]->dirty) {
   Outcount++;
   dsmgr.WritePage(ptr->page_id, buf[ptr->frame_id]);
  }

  // 删除 BCB 列表中对应的 BCB
  RemoveBCB(ptr, ptr->page_id);

  //cout << "选择 " << ptr->frame_id << "frame 换出" << endl;
  int frame_id = ptr->frame_id;
  delete ptr;
  return frame_id;
}
```

SelectVictim 函数功能：选择 LRUList 中堆顶的 BCB，即最近没有访问过的 page，删除该 BCB 并整堆，如果该 page 为 dirty，则需写回磁盘，删除 BCB 列表（bcbList）、ftop、ptof 中对应的记录。

RemoveLRUEle 函数如下：

```
void BMgr::RemoveLRUEle(int frame_id) {
  // 删除 LRUList 中对应的 BCB
  pop_heap(LRUList.begin(), LRUList.end(), cmp);
  LRUList.pop_back();
}
```

RemoveLRUEle 函数功能：将 LRUList 堆顶的 BCB 与最后一个 BCB 交换位置，删除后再整堆。

SetDirty 函数如下：

```
1. void BMgr::SetDirty(int frame_id) {
2.  bcbList[frame_id]->dirty = 1;
3. }
```

SetDirty 函数功能：将 frame_id 对应的 BCB 中的 dirty 属性置 1。

UnsetDirty 函数如下：

```
1. void BMgr::UnsetDirty(int frame_id) {
2.  bcbList[frame_id]->dirty = 0;
3. }
```

UnsetDirty 函数功能：将 frame_id 对应的 BCB 中的 dirty 属性置 0。

WriteDirtys 函数如下：

```
1. void BMgr::WriteDirtys() {
2.  for (int i = 0; i < DEFBUFSIZE; ++i) {
3.   // 将所有 dirty 的 frame 写回磁盘
4.   if (bcbList[i] != nullptr && bcbList[i]->dirty == 1) {
5.    Outcount++;
6.    dsmgr.WritePage(ftop[i], buf[i]);
7.   }}}
```

WriteDirtys 函数功能：将所有 dirty 的 page 写回磁盘。

PrintFrame 函数如下：

```
1. void BMgr::PrintFrame(int frame_id) {
2.  cout << "BCB: \n"
3.   << "\tpage_id: " << bcbList[frame_id]->page_id << endl
4.   << "\tframe_id: " << bcbList[frame_id]->frame_id << endl
5.   << "\tlatch: " << bcbList[frame_id]->latch << endl
6.   << "\tcount: " << bcbList[frame_id]->count << endl
7.   << "\tdirty: " << bcbList[frame_id]->dirty << endl
8.   << "\ttime: " << bcbList[frame_id]->next << endl;}
```

PrintFrame 函数功能：打印 frame_id 对应 BCB 的所有信息。

DSMgr 类的 OpenFile 函数如下：

```
1. int DSMgr::OpenFile(string filename) {
2.  currFile = fopen(filename.c_str(), "r+");
3.  return 0;
4. }
```

OpenFile 函数功能：以读写模式打开 filename 对应的文件。

CloseFile 函数如下：

```
1. int DSMgr::CloseFile() {
2.  fclose(currFile);
3.  return 0;
4. }
```

CloseFile 函数功能：关闭已打开的文件

ReadPage 函数如下：

```
1. bFrame DSMgr::ReadPage(int page_id) {
2.  fseek(currFile, page_id * FRAMESIZE, SEEK_SET);
3.  bFrame tmp;
4.  fread(&tmp, sizeof(char), FRAMESIZE, currFile);
5.  return tmp;
6. }
```

ReadPage 函数功能：根据 page_id，从数据库文件中读取对应的 page 并将其返回。

WritePage 函数如下：

```
1. int DSMgr::WritePage(int frame_id, bFrame frm) {
2.  fseek(currFile, frame_id * FRAMESIZE, SEEK_SET);
3.  fwrite(&frm, sizeof(char), FRAMESIZE, currFile);
4.  return 0;
5. }
```

WritePage 函数功能：将 dirty page 写回到数据库文件中对应的位置。

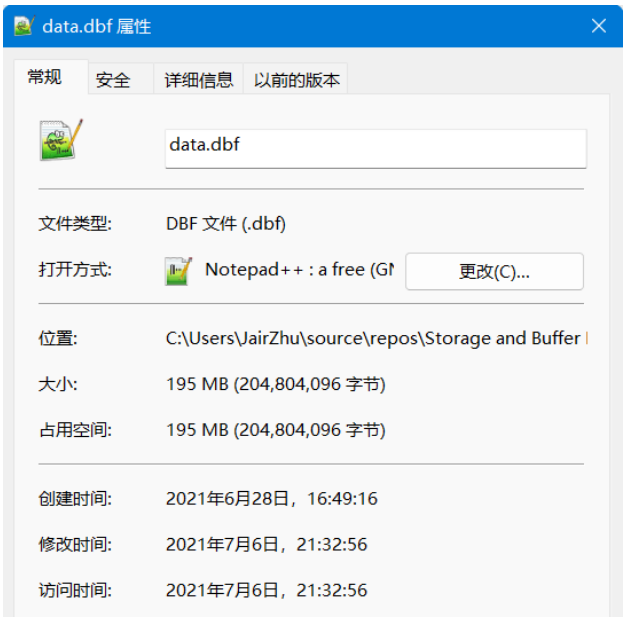
主程序如下：

```
int main() {
  BMgr bmgr;

  // 调用 FixNewPage 50000 次以建立 data.dbf 文件
  //for (int i = 0; i < MAXPAGES; ++i) {
  // bmgr.FixNewPage();}
  //bmgr.WriteDirtys();

  fstream file;
  file.open("data-5w-50w-zipf.txt");
  vector<pair<int, int>> opList;

  // 读取 trace 文件生成操作列表
  while (file.good()) {
   pair<int, int> op;
   char t;
   file >> op.first;
   file >> t;
   file >> op.second;
   opList.push_back(op);
  }

  // 依据操作列表模拟数据库访问磁盘
  for (int i = 0; i < opList.size(); ++i) {
   int page_id = opList[i].second;
   int dirty = opList[i].first;
   int frame_id = bmgr.FixPage(page_id, 0);
   if (dirty) {
    bmgr.SetDirty(frame_id);}
   bmgr.UnfixPage(page_id);
  }

  // 数据库关闭前将 dirty frame 写入磁盘
  bmgr.WriteDirtys();

  cout << "命中次数：\t" << bmgr.getHit() << endl
   << "命中率：  \t" << 1.0 * bmgr.getHit() / opList.size() << endl
   << "未命中次数：\t" << bmgr.getMiss() << endl
   << "读取磁盘次数：\t" << bmgr.getIncount() << endl
   << "写入磁盘次数：\t" << bmgr.getOutcount() << endl;
  return 0;}
```

主程序功能：读取 trace 文件以生成操作列表，依据操作列表模拟数据库操作，数据库关闭前将 dirty page 全部写回磁盘，最后打印统计信息。

实验结果

1. 生成数据库文件 data.dbf 的结果如下：

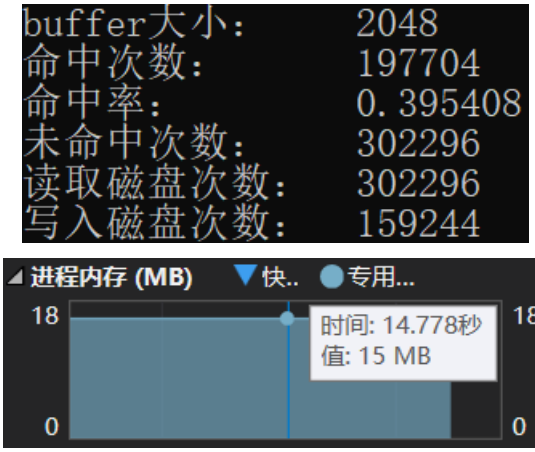


文件大小$204804096 = 4096 \times 50000 + 4096$，文件中第一块 page 用于存储指向其他 page 的指针，不存记录数据。

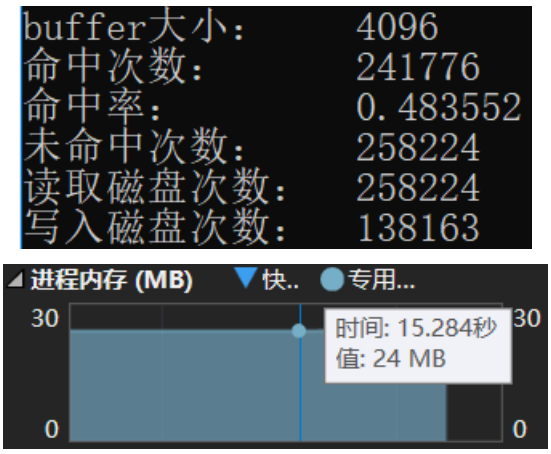
2. 比较不同的 DEFBUFSIZE 对 IO 次数的影响：

（1） 当 DEFBUFSIZE = 1024 时，统计结果如下：

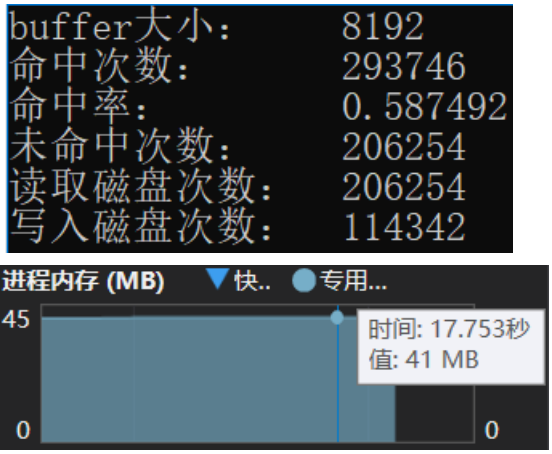
（2） 当 DEFBUFSIZE = 2048 时，统计结果如下：



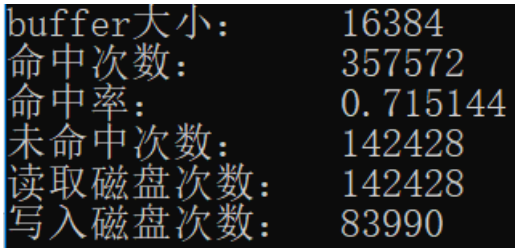
（3） 当 DEFBUFSIZE = 4096 时，统计结果如下：

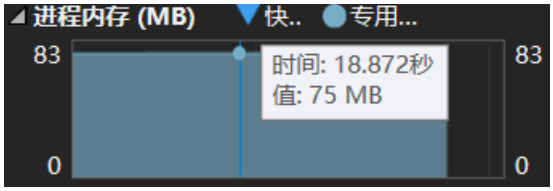


（4） 当 DEFBUFSIZE = 8192 时，统计结果如下：

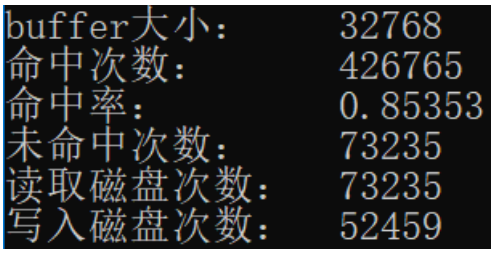


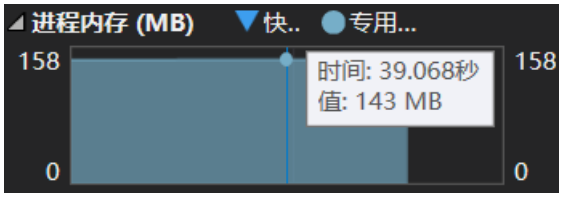
（5） 当 DEFBUFSIZE = 16384 时，统计结果如下：



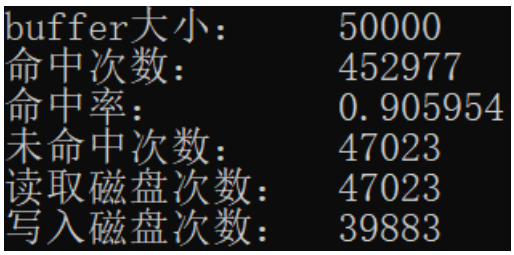


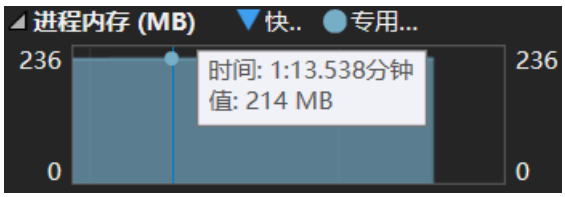
（6） 当 DEFBUFSIZE = 32768 时，统计结果如下：



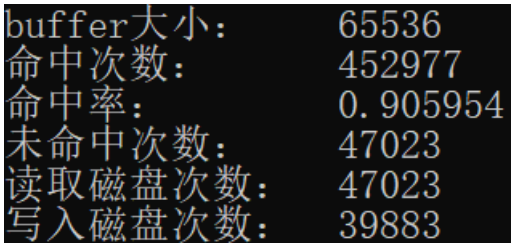


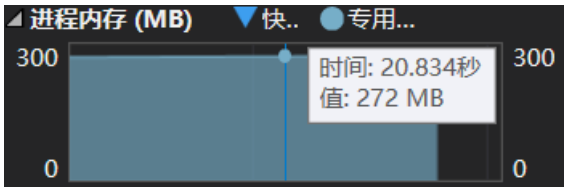
（7） 当 DEFBUFSIZE = 50000 时，统计结果如下：





（8） 当 DEFBUFSIZE = 65536 时，统计结果如下：





上述数据整理如下：

DEFBUFSIZE	命中次数	命中率	读取磁盘次数	写入磁盘次数	占用内存
1024	159779	31.96%	340221	177385	11MB
2048	197704	39.54%	302296	159244	15MB
4096	241776	48.36%	258224	138163	24MB
8192	293746	58.75%	206254	114342	41MB
16384	357572	71.51%	142428	83990	75MB
32768	426765	85.35%	73235	52459	143MB
50000	452977	90.60%	47023	39883	214MB
65536	452977	90.60%	47023	39883	272MB

由上表可知，随着 DEFBUFSIZE 的增加，命中次数也逐渐增大，命中率也逐渐提高，占用的内容也逐渐增大，但读取和写入磁盘的次数逐渐减少。

同时也可以看到，当 DEFBUFSIZE 增加到一定数量时，命中次数、命中率、读取和写入磁盘的次数均减小到稳定值。

虽然最后两次测试的 DEFBUFSIZE ≥ 50000，但其命中率没有达到 100%，原因是数据库启动时，buffer 为空，每个 page 第一次载入 buffer 都会出现一次未命中，换句话说，未命中次数就是 trace 文件中 page_id 的种类数 47023，显然，trace 文件中 page_id 没有覆盖 data.dbf 中所有 page。