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

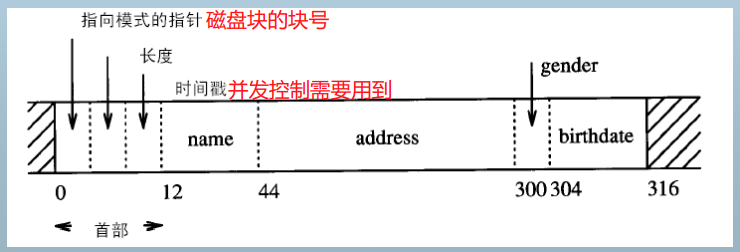
CPU：AMD Ryzen 5 3600X

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，，即每个page可以存储12个记录数据，创建新page的代码如下：

1. bFrame CreatePage(int num) {
2. bFrame frame;
3. int i = 0;
4. *// 数据块号*
5. auto ch = static\_cast<char\*>(static\_cast<void\*>(&num));
6. copy(ch, ch + sizeof(ch), frame.field);
7. i += sizeof(ch);
8. *// 时间戳*
9. unsigned int btimestamp = 0;
10. ch = static\_cast<char\*>(static\_cast<void\*>(&btimestamp));
11. copy(ch, ch + sizeof(ch), frame.field + i);
12. i += sizeof(ch);
13. *// 偏移量表*
14. unsigned short int offset[FRAMESIZE / RECORDSIZE];
15. int BLOCK\_HEAD = sizeof(btimestamp) + sizeof(offset);
16. for (int t = 0; t < FRAMESIZE / RECORDSIZE; t++) {
17. int toffset = BLOCK\_HEAD + t \* RECORDSIZE;
18. ch = static\_cast<char\*>(static\_cast<void\*>(&toffset));
19. copy(ch, ch + sizeof(ch), frame.field + i);
20. i += sizeof(ch);
21. };
22. for (int j = 0; j < FRAMESIZE / RECORDSIZE; ++j) {
23. *// 写入记录*
24. char\* p\_schema = new char[4];
25. copy(p\_schema, p\_schema + sizeof(p\_schema), frame.field + i);
26. i += sizeof(p\_schema);
27. delete p\_schema;
28. unsigned int timestamp = num;
29. ch = static\_cast<char\*>(static\_cast<void\*>(&timestamp));
30. copy(ch, ch + sizeof(ch), frame.field + i);
31. i += sizeof(ch);
32. unsigned int length = 316;
33. ch = static\_cast<char\*>(static\_cast<void\*>(&length));
34. copy(ch, ch + sizeof(ch), frame.field + i);
35. i += sizeof(ch);
36. char namePtr[32];
37. copy(namePtr, namePtr + sizeof(namePtr), frame.field + i);
38. i += sizeof(namePtr);
39. char addressPtr[256];
40. copy(addressPtr, addressPtr + sizeof(addressPtr), frame.field + i);
41. i += sizeof(addressPtr);
42. char genderPtr[4];
43. copy(genderPtr, genderPtr + sizeof(genderPtr), frame.field + i);
44. i += sizeof(genderPtr);
45. char birthdayPtr[12];
46. copy(birthdayPtr, birthdayPtr + sizeof(birthdayPtr), frame.field + i);
47. }
48. return frame;
49. }

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. for (int i = 0; i < DEFBUFSIZE; ++i) {
5. bcbList[i] = nullptr;
6. ftop[i] = -1;
7. }
8. *// 初始化 buf 缓存区*
9. buf = new bFrame[DEFBUFSIZE];
10. cout << "buffer大小： \t" << DEFBUFSIZE << endl;
11. *// 打开数据库文件*
12. dsmgr.OpenFile("data.dbf");
14. Time = hit = miss = Incount = Outcount = 0;
15. }

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：

1. int BMgr::FixPage(int page\_id, int prot) {
2. int frame\_id;
3. int key = Hash(page\_id);
4. *// 在 ptof 中查找 page\_id 是否存在，即 page\_id 是否事先载入内存*
5. if (ptof.find(key) != ptof.end()) {
6. for (int i = 0; i < ptof[key].size(); ++i) {
7. if (ptof[key][i]->page\_id == page\_id) {
8. frame\_id = ptof[key][i]->frame\_id;
9. bcbList[frame\_id]->count++;
10. bcbList[frame\_id]->latch = 1;
11. *// 统计命中次数*
12. *//cout << "命中 " << frame\_id << "frame" << endl;*
13. hit++;
14. for (int k = 0; k < LRUList.size(); ++k) {
15. if (LRUList[k]->page\_id == page\_id) {
16. *// 更新 BCB 的 Time 元素*
17. LRUList[k]->next = Time++;
18. *// 调整 LRUList*
19. push\_heap(LRUList.begin(), LRUList.end(), cmp);
20. return frame\_id;
21. }
22. }
23. }
24. }
25. }
26. if (NumFreeFrames() == 0) {
27. *// 选择被替换的 frame*
28. frame\_id = SelectVictim();
29. }
30. else {
31. for (int i = 0; i < DEFBUFSIZE; ++i) {
32. *// 查找空闲的 frame*
33. if (bcbList[i] == nullptr) {
34. frame\_id = i;
35. break;
36. }
37. }
38. }
39. *// 统计未命中次数*
40. miss++;
41. Incount++;
42. *// 将 page 载入内存的 frame*
43. buf[frame\_id] = dsmgr.ReadPage(page\_id);
44. *// 添加新的 BCB 到 BCB 列表*
45. bcbList[frame\_id] = new BCB(page\_id, frame\_id, 1, 1, 0, Time++);
46. *// 将新的 BCB 加入 LRUList*
47. LRUList.push\_back(bcbList[frame\_id]);
48. push\_heap(LRUList.begin(), LRUList.end(), cmp);
49. *// 更新 ftop*
50. ftop[frame\_id] = page\_id;
51. *// 更新 ptof*
52. if (ptof.find(key) == ptof.end()) {
53. vector<BCB\*> list;
54. list.push\_back(bcbList[frame\_id]);
55. ptof[key] = list;
56. }
57. else {
58. for (int i = 0; i < ptof[key].size(); ++i) {
59. if (ptof[key][i]->page\_id == page\_id) {
60. ptof[key][i] = bcbList[frame\_id];
61. return frame\_id;
62. }
63. }
64. ptof[key].push\_back(bcbList[frame\_id]);
65. }
66. return frame\_id;
67. }

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. for (int i = 0; i < MAXPAGES; ++i) {
4. *// 找到一个空闲的 page*
5. if (dsmgr.GetUse(i) == 0) {
6. page\_id = i;
7. dsmgr.SetUse(i, 1);
8. dsmgr.IncNumPages();
9. break;
10. }
11. }
12. if (NumFreeFrames() == 0) {
13. *// 选择被替换的 frame*
14. frame\_id = SelectVictim();
15. }
16. else {
17. for (int i = 0; i < DEFBUFSIZE; ++i) {
18. *// 查找空闲的 frame*
19. if (bcbList[i] == nullptr) {
20. frame\_id = i;
21. break;
22. }
23. }
24. }
25. *// 生成一个新的 page*
26. bFrame nframe = CreatePage(page\_id);
27. *// 将新的 page 复制到 buffer*
28. copy(nframe.field, nframe.field + FRAMESIZE, buf[frame\_id].field);
29. *// 添加新的 BCB 到 BCB 列表*
30. bcbList[frame\_id] = new BCB(page\_id, frame\_id, 1, 1, 1, Time++);
31. *// 将新的 BCB 加入 LRUList*
32. LRUList.push\_back(bcbList[frame\_id]);
33. push\_heap(LRUList.begin(), LRUList.end(), cmp);
34. *// 更新 ftop*
35. ftop[frame\_id] = page\_id;
36. int key = Hash(page\_id);
37. *// 更新 ptof*
38. if (ptof.find(key) == ptof.end()) {
39. vector<BCB\*> list;
40. list.push\_back(bcbList[frame\_id]);
41. ptof[key] = list;
42. }
43. else {
44. for (int i = 0; i < ptof[key].size(); ++i) {
45. if (ptof[key][i]->page\_id == page\_id) {
46. ptof[key][i] = bcbList[frame\_id];
47. }
48. }
49. ptof[key].push\_back(bcbList[frame\_id]);
50. }
51. return NewPage(page\_id, frame\_id);
52. }

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：

1. int BMgr::UnfixPage(int page\_id) {
2. *// 根据 page\_id 查找对应的 BCB*
3. BCB\* bcb = ptof[Hash(page\_id)][0];
4. for (int i = 0; i < ptof[Hash(page\_id)].size(); ++i) {
5. if (ptof[Hash(page\_id)][i]->page\_id == page\_id) {
6. bcb = ptof[Hash(page\_id)][i];
7. break;
8. }
9. }
10. bcb->count--;
11. if (bcb->count == 0) {
12. bcb->latch = 0;
13. }
14. *// 更新 BCB 的 Time*
15. bcb->next = Time++;
16. *// 调整 LRUList*
17. push\_heap(LRUList.begin(), LRUList.end(), cmp);
18. return bcb->frame\_id;
19. }

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：

1. int BMgr::NumFreeFrames() {
2. return DEFBUFSIZE - LRUList.size();
3. }

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

SelectVictim函数如下：

1. int BMgr::SelectVictim() {
2. *// 选择 LRUList 中第一个 BCB，即最近没有访问过的 frame*
3. BCB\* ptr = LRUList.front();
4. *// 删除 LRUList 中第一个 BCB*
5. RemoveLRUEle(ptr->frame\_id);
6. *// 如果为 dirty 则需写回磁盘*
7. if (bcbList[ptr->frame\_id]->dirty) {
8. Outcount++;
9. dsmgr.WritePage(ptr->page\_id, buf[ptr->frame\_id]);
10. }
11. *// 删除 BCB 列表中对应的 BCB*
12. RemoveBCB(ptr, ptr->page\_id);
13. *//cout << "选择 " << ptr->frame\_id << "frame 换出" << endl;*
14. int frame\_id = ptr->frame\_id;
15. delete ptr;
16. return frame\_id;
17. }

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：

1. void BMgr::RemoveLRUEle(int frame\_id) {
2. *// 删除 LRUList 中对应的 BCB*
3. pop\_heap(LRUList.begin(), LRUList.end(), cmp);
4. LRUList.pop\_back();
5. }

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。

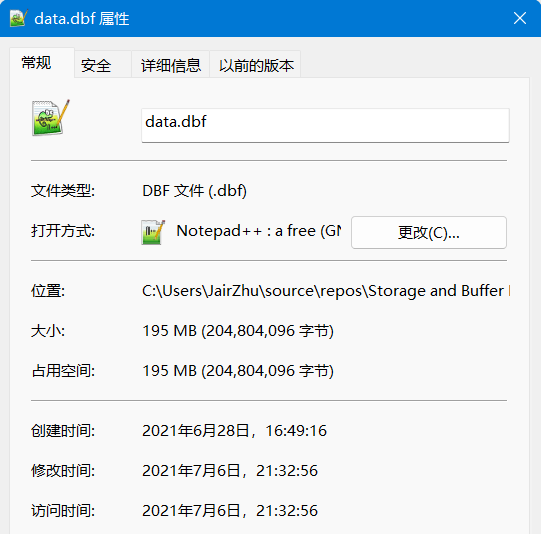
主程序如下：

1. int main() {
2. BMgr bmgr;
3. *// 调用 FixNewPage 50000次以建立 data.dbf 文件*
4. *//for (int i = 0; i < MAXPAGES; ++i) {*
5. *// bmgr.FixNewPage();}*
6. *//bmgr.WriteDirtys();*
7. fstream file;
8. file.open("data-5w-50w-zipf.txt");
9. vector<pair<int, int>> opList;
10. *// 读取 trace 文件生成操作列表*
11. while (file.good()) {
12. pair<int, int> op;
13. char t;
14. file >> op.first;
15. file >> t;
16. file >> op.second;
17. opList.push\_back(op);
18. }
19. *// 依据操作列表模拟数据库访问磁盘*
20. for (int i = 0; i < opList.size(); ++i) {
21. int page\_id = opList[i].second;
22. int dirty = opList[i].first;
23. int frame\_id = bmgr.FixPage(page\_id, 0);
24. if (dirty) {
25. bmgr.SetDirty(frame\_id);}
26. bmgr.UnfixPage(page\_id);
27. }
28. *// 数据库关闭前将 dirty frame 写入磁盘*
29. bmgr.WriteDirtys();
30. cout << "命中次数：\t" << bmgr.getHit() << endl
31. << "命中率： \t" << 1.0 \* bmgr.getHit() / opList.size() << endl
32. << "未命中次数：\t" << bmgr.getMiss() << endl
33. << "读取磁盘次数：\t" << bmgr.getIncount() << endl
34. << "写入磁盘次数：\t" << bmgr.getOutcount() << endl;
35. 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。

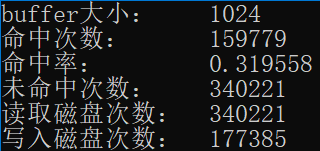
## 实验结果

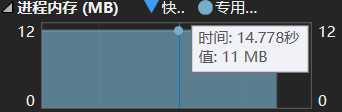
1. 生成数据库文件data.dbf的结果如下：



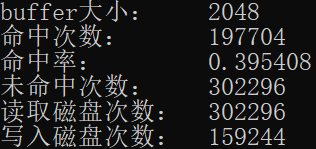
文件大小，文件中第一块page用于存储指向其他page的指针，不存记录数据。

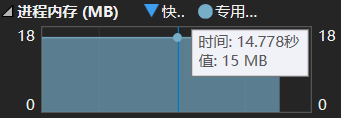
1. 比较不同的DEFBUFSIZE对IO次数的影响：
2. 当DEFBUFSIZE = 1024时，统计结果如下：



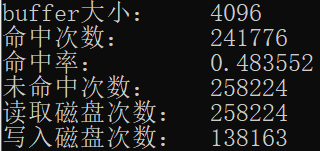


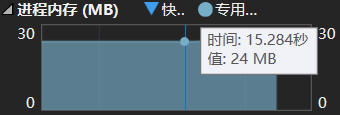
1. 当DEFBUFSIZE = 2048时，统计结果如下：



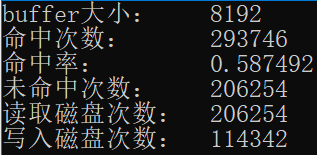


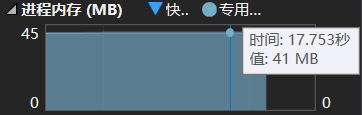
1. 当DEFBUFSIZE = 4096时，统计结果如下：



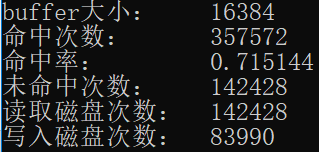


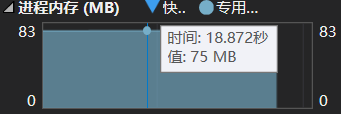
1. 当DEFBUFSIZE = 8192时，统计结果如下：



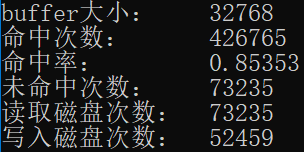


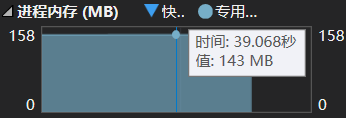
1. 当DEFBUFSIZE = 16384时，统计结果如下：



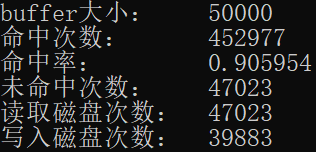


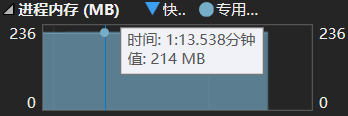
1. 当DEFBUFSIZE = 32768时，统计结果如下：



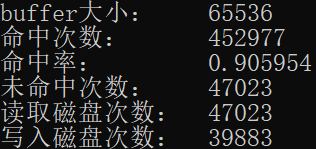


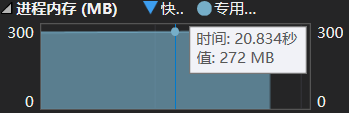
1. 当DEFBUFSIZE = 50000时，统计结果如下：





1. 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。