Buffer Manager 设计报告

3120101973 陆洲

一、模块概述

数据库操作中涉及大量的文件读写操作。直接的磁盘读写速度过慢，显然不是一个高效的方法。因此，我们需要使用一个更快速的内存模型。于是，在 MiniSQL 中，我们加入了 Block 和缓冲区这两个概念。Block 的定义在 Block 类中，对缓冲区的实现和操作则由 BufferManager 模块来体现。这个模块属于较底层模块，因此将涉及直接的文件读写、缓冲管理，也将提出一些优化措施。

为提高磁盘 I/O 操作的效率，缓冲区与文件系统交互的单位是块，块的大小应为文件系统与磁盘交互单位的整数，一般可定为 4KB 或 8KB，本 MiniSQL 系统设置 Block 大小为 4KB。

二、主要功能

Buffer Manager 负责缓冲区的管理，主要功能有：

1. 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文件；
2. 实现缓冲区的替换算法，当缓冲区满时选择合适的页进行替换；
3. 记录缓冲区中各页的状态，如是否被修改过等；
4. 提供缓冲区页的 pin 功能，及锁定缓冲区的页，不允许替换出去。

三、主要对外接口

1）Block 类（主要供 BufferManager 使用）

1.向 Buffer 中的 Block 添加数据

```
int addData(char* data, int data_size);
```

2.读取 Buffer 中的 Block 里所需的数据

```
void readinData(int index, char* temp);
```

3.删除 Buffer 中的 Block 里的指定数据

```
void removeData(int index);
```

4.将缓冲区中的数据写出到文件

```
void saveToDisk();
```

5.向磁盘请求数据到系统缓冲区

```
void getFromDisk(int block_position, CFile* file);
```

2）BufferManager 类

1.替换算法

int findReplaceBlock();

2.向下层的接口，请求磁盘数据

int readBlockFromDisk(int blockPosition);

3.面向 RecordManager 的接口

void addRecord(CString tableName, CString head, int size);

void removeRecord(Address add);

Address nextRecordPosition(Address currentAdd);

void findTableEntry(CString tableName, Address& returnAddress);

4.面向 CatalogManager 的接口

void addInfo(CString head, int size);

CString readInfo(CString tableName, int& size);

void removeInfo(CString tableName);

int addTableEntry(CString tableName, Address& returnAddress);

void removeTableEntry(CString tableName);

5.单个数据项的接口

int storeData(CString head, int size, Address& returnAddress);

CString readData(Address address, int& size);

void removeData(Address address);

四、设计思路

1.Block 类

在 Block 类中主要实现的是 Block 的基本定义及一些便于 BufferManager 管理的函数。为完整实现 Block 的功能，我们设置了三个标志位，分别是数据是否有效（脏数据）、数据是否被锁住、数据是否被删除。

1）在一个 Block 被读入缓冲区时，我们将 Block 上的有效位置位为 1，表示数据有效。

2）在 Buffer 的 Update 正在进行时，我们将修改完成标志复位为 0；此时将这个 Block 将不允许被替换出缓冲区。在修改完后将标志位置位，表示修改已完成，可以进行替换。

3）执行删除时，因为删除的代价较大，因此我们采用假删，即将删除标志置位为 1，在下一次为储存数据查找空位时可以寻找假删部分进行添加。

4）提供 Pin 功能，一般使用索引的 block 不会被替换。

类定义源码：

```
class Block{
public:
	char space[BLOCKSIZE];//set free space, using array to store data
	int block_position;//point out the address of block in the whole file
	CTime last_use;//update the last access time to preform LRU
	bool valid;//valid flag, opposite of dirty
	bool modified;//modified
	bool pinned;
	bool deleted;//flag to preform fake deletion
	CFile* file;//for each table, we create one file

public:
```

```
    //basic function
    ……

    //gets and sets, basically record and index operations
    ……

    //interface function
    int addData(char*, int);
    void readData(int, char*);
    void removeData(int);
    void saveToDisk();
    void getFromDisk(int, CFile*);
}
```

2.BufferManager 类

定义了 Block 大小 4KB，缓冲区最大可以存放 100 个 Block,使用 set 来寻址，因为数据量较小，所以将每一个 Block 当做一个页。

上层的 Manager（RecordManager 和 IndexManager）向 BufferManager 申请所需要的数据，该模块将先查看数据是否已在缓冲区中存在，若存在，则将直接返回，否则将从磁盘读取块进入缓冲区再返回。

替换算法选择 LRU 算法，使用链表来实现。每次将最近使用的 Block 置于链表头，在 Buffer 充满时将末位的 Block 替换出去。

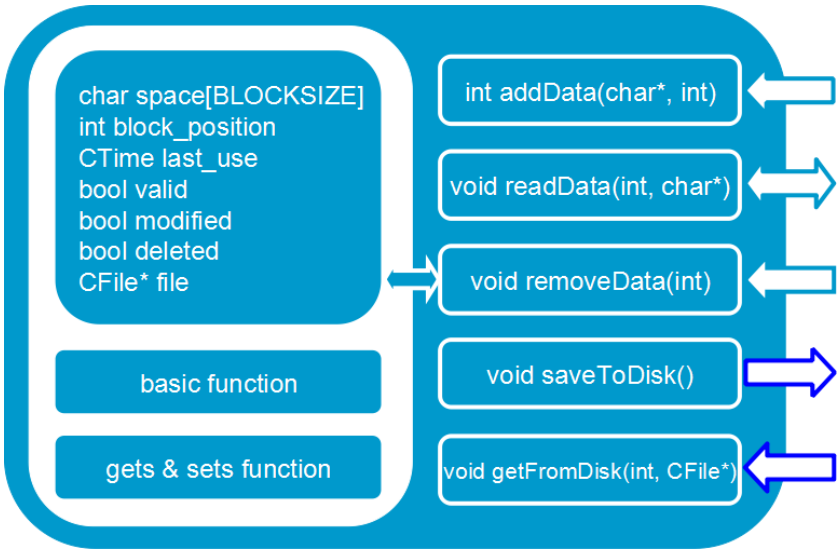
类定义源码：

```
class BufferManager
{
public:
    DBFile m_file;
    Block buffer[100];
    std::set<int> bufferContainer;

public:
    //basic buffer function
    ……
    int findReplaceBlock();

    //interface function
    int storeData(String head, int size, Address& returnAddress);
    String readData(Address position, int& size);
    void removeData(Address position);
    void addRecord(CString tableName, String head, int size);
    void removeRecord(Address add);
    Address nextRecordPosition(Address currentAdd);
    void addInfo(String head, int size);
    String readInfo(CString tableName, int& size);
```

```
    void removeInfo(CString tableName);
    int addTableEntry(CString tableName, Address& returnAddress);
    void findTableEntry(CString tableName, Address& returnAddress);
    void removeTableEntry(CString tableName);
};
```

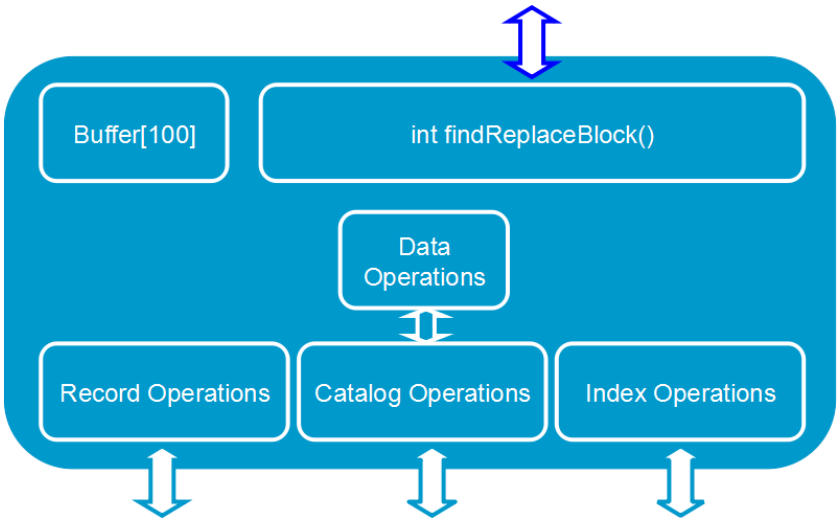
五、整体架构

1）Block 类



该类拥有一些内部函数，主要是用于判断、修改标志值以及提供信息给接口函数使用。另外，在接口函数中，上三个函数用于对上层使用，下两个函数用于沟通下层。

2）BufferManager 类



该类在主要实现对下层数据的管理和对上层 Manager 的支持。最核心的部分其实是对 Block 的调度。兼有一系列针对上层 Manager 的接口。

六、关键函数及伪码

1.Block 类 sets & gets 函数设计（简便起见，仅列出 set）

```
void Block::setBlockPosition(int position)
{
    //设 block 的位置指向
    //……
}

void Block::setValid(bool flag)
{
    //设置 valid 标志
    //……
}

void Block::setPinned(bool flag)
{
    //设置 pin 标志
    //……
}


void Block::setRecordNum(int num)
{
    //设置 record 的 id
    //……
}

void Block::setFreeSpaceEnd(int end)
{
    //设置空地址的尾部
    //……
}

void Block::setDataSize(int index, int size)
{
    //设置不同的数据的长度，便于储存 metadata
    //每个数据有一个内部 id，从 id 可以知道数据长度等信息
    //……
}

void Block::setDataPoint(int index, int position)
```

```
{
    //设置指向数据的指针
    //……
}

void Block::setRecordNextLink(int index, Address next)
{
    //设置下一条记录，并且将该 block 修改标志置位
    //……
    //同时获得读取它的时间，为替换算法服务
    last_use = CTime::GetCurrentTime();
    modified = 1;
}

void Block::setRecordFrontLink(int index, Address front)
{
    //设置 record 的前一条记录，以用于 sequencial file
    //……
    //同时获得读取它的时间，为替换算法服务
    last_use = CTime::GetCurrentTime();
    modified = 1;
}

void Block::setIndexNext(int index, Address next)
{
    //向后扫描 index
    //……
}
```

Set 和 get 函数主要是为了避免对成员变量的直接修改而设。因此整个函数会比较短小，但是在修改标志位、进行指针位移等微操作中会发挥很大作用。

2.添加数据

```
int Block::addData(char* data, int data_size)
{
    if(haveEnoughSize(data_size) == 0)
      //如果没有足够空间来添加，将返回错误

    ……
    for(i = 0; i < total; i++)//检查中间是否有空
        if(getDataSize(i) == 0)
            break;
    if(i != total){//如果中间的空位就足够，先存在中间的空位里
        ……
        //set flag;
        last_use = CTime::GetCurrentTime();
```

```
            modified = 1;
            return index;
        }
        else{//空位不够的话，从 free space 的末尾开始存
            ……
            int FreeSpaceEnd = getFreeSpaceEnd();//compute the space
            ……
            for(i = 0; i < data_size; i++)//store the data into Block
                space[FreeSpaceEnd + 1 + i] = data[i];
            last_use = CTime::GetCurrentTime();
            modified = 1;
            return total;
        }
}
```

该函数进行添加数据操作。

3.读取数据

```
void Block::readData(int index, char* temp)
{
        int size = getDataSize(index);

        int length = 0;
        int p = getDataPoint(index);
        int end = p + getDataSize(index) - 1;
        for(; p <= end; p++, length++)//create a temp to get the data and return
            temp[length] = space[p];
        last_use = CTime::GetCurrentTime();
}
```

该函数进行读取数据操作。

4.

```
int compare(const void*, const void*);
int compare(const void * a, const void * b)//set the function pointer for qsort()
{
        return ((item *)a)->point > ((item *)b)->point;
}
void Block::removeData(int index)
{
        int total = getRecordNum();
        //排除要删除的地方不存在的错误
        if(index >= total)
            return;
        if(buffer[index].deleted)
            return;

        //采用 sequencial file order
```

```
    //因为在添加的时候虽然指针指向是正确的，但是线性读取的话并不是完全有序的
    //为使删除更简便，我们统一在删除时先进行排序
    ……
    int temp;
    if (size != 0)
    {
        qsort((void *)array, size, sizeof(item), compare);//快排
        for(int i = size - 1; i >= 0; i--)
        {
            if(i == size - 1)//最后一条记录，指针不再下移
            {
                temp = current + oldSize - array[i].size;
                for(int j = 1; j <= array[i].size; j++)
                    //假删标记
                    space[current+oldSize-j].deleted = 1;
                      ……
            }
            else//其他记录
            {
                for(int j = 1; j <= array[i].size; j++)
                    //假删标记
                    space[temp - j].deleted = 1;
                ……
                    //指针下移
            }
        }
    }
    //delete redundant space and set the pointer
    ……
    last_use = CTime::GetCurrentTime();
    modified = 1;
}
```

删除操作只做假删标记，并且为提高效率进行了排序。

5.与 Disk 的接口

```
//access disk derictly
void Block::saveToDisk()
{
    if(file == NULL)
        return;
    if(valid && modified)
    {
        if(block_position != -1)
        {
            //seek & write
```

```
                ……
            }
        }
}

void Block::getFromDisk(int block_position, CFile* file)
{
    if((unsigned int)block_position >= file->GetLength()/BLOCKSIZE)    //testing whether
there is an overflow occured
    {
        this->block_position = -1;
        return;
    }
    ……
    //对于一个刚添加入 Buffer 的块，将标志位按以下方法初始化
    last_use = CTime::GetCurrentTime();
    valid = 1;
    modified = 0;
}
```

6.替换算法

```
int BufferManager::findReplaceBlock()
{
    int replacePosition = 0;
    CTime earliestTime = buffer[0].getUseTime();
    int i;
    for(i = 0; i < BUFFERSIZE; i++)
    {
        if(!buffer[i].IsValid())
            break;
        if(earliestTime > buffer[i].getUseTime()&&(buffer[i].pinned == 1))
        {//按时间循环比较，如果有最近最早使用的块，就选出来，直到找到最早
        //排除正在被 pin 的块
            replacePosition = i;
            earliestTime = buffer[i].getUseTime();
        }
    }
    if(i != BUFFERSIZE)
        return i;
    else
        return replacePosition;
}
```

7.面向 CatalogManager 的接口

```
int BufferManager::addTableEntry(CString tableName, Address &returnAddress)
{
```

```
	//存入 TableName 等信息，并将开头的 6 个空间空出留待其他用途
	……

	int j = storeData(tableEntryHead, size, returnAddress);
	//设置指针等
	……
	return j;
}

void BufferManager::removeTableEntry(CString tableName)
{
	Address add, oldAdd, nextAdd;
	oldAdd.blockAddress = 0;
	oldAdd.indexAddress = 0;
	int i = readBlockFromDisk(0);
	buffer[i].getNextLink(0,add);
	CString name;
	while(add.blockAddress != -1 && add.indexAddress != -1)
	{
		//转换成容易比较的形式
		if(name == tableName)
		{
			//根据 TableName 查找并且读到它的其他指针信息等
			break;
		}
		delete [] temp;
		oldAdd = add;
		add = nextAdd;
	}
}

void BufferManager::addInfo(CString head, int size)
{
	//根据地址找到该表添加 Info
	//设置指针指向下一空的地址
	……
}

CString BufferManager::readInfo(CString tableName, int& size)
{
	int num;
	CString a;
	Address addr;
	CString info;
```

```
	……

	while (addr.indexAddress != -1)//如果 indexAddress 存在，存在这个表
	{
		//一条一条向下读
		……
	}

	if (addr.indexAddress == -1)//不存在这个表
	{
		size = -1;
		info.Empty();
	}
	return info;
}

void BufferManager::removeInfo(CString tableName)
{
	int i = readBlockFromDisk(0);
	Address add;
	buffer[i].getNextLink(1,add);

	Address oldAdd, nextAdd;
	CString name;
	int size;
	oldAdd.blockAddress = 0;
	oldAdd.indexAddress = 1;
	int j;
	while(add.blockAddress != -1 && add.indexAddress != -1)
	{
		……
		if(name == tableName)
		{
			……
			//移动指针的位置，并删除
			break;
		}
		delete [] TableInfo;
		//设置新的指针
		oldAdd.blockAddress = add.blockAddress;
		oldAdd.indexAddress = add.indexAddress;
		add.blockAddress = nextAdd.blockAddress;
		add.indexAddress = nextAdd.indexAddress;
```

```
    }
}
```

8.面向 RecordManager 的接口

```
void BufferManager::findTableEntry(CString tableName, Address& returnAddress)
{
    ……
    CString tableEntry;
    while(add.blockAddress != -1 && add.indexAddress != -1)
    {//将地址中储存的 TableEntry 做一个转化，转化成上层需要的形式
        ……
    }
    returnAddress.blockAddress = add.blockAddress;
    returnAddress.indexAddress = add.indexAddress;
}

void BufferManager::addRecord(CString tableName, CString head, int size)
{
    Address add, dataAdd, nextAdd;
    findTableEntry(tableName, add);
    if(add.blockAddress == -1 && add.indexAddress == -1)
    {
        //这条记录是这个 block 的最后一条
        ……
    }
    else
    {
        int i = readBlockFromDisk(add.blockAddress);
        buffer[i].getRecordNextLink(add.indexAddress, nextAdd);
        if(nextAdd.blockAddress != -1 && nextAdd.indexAddress != -1)
        {
            //最普通的情况
            ……
        }
        else
        {
            //该条记录是最开始的一条
            ……
        }
    }
}

void BufferManager::removeRecord(Address add)
{
    ……
```

```
    if(frontAdd.blockAddress == nextAdd.blockAddress &&
        frontAdd.indexAddress == nextAdd.indexAddress)
    {
        //这个记录不是跨块删除，指针重新布置
        ……
    }
    else
    {
        //跨块删除，指针重新布置
        ……
    }
}
```

9.面向数据的接口

```
int BufferManager::storeData(CString head, int size, Address& returnAddress)
{
    int replacePosition = 0;
    CTime earliestTime = buffer[0].getUseTime();
    int i;
    for(i = 0; i < 100; i++)
    {//若该块 block 因各种理由不能使用，返回错误
        if(!buffer[i].IsValid() ||
            (buffer[i].haveEnoughSize(size) && buffer[i].getBlockAddress() != 0))
            break;
        if(earliestTime > buffer[i].getUseTime())
        {
            replacePosition = i;
            earliestTime = buffer[i].getUseTime();
        }
    }
    if(i != 100)
    {
        if(buffer[i].IsValid())
        {
            returnAddress.blockAddress = buffer[i].getBlockAddress();
            returnAddress.indexAddress = buffer[i].addData(head,size);
        }
        else
        {
            int free_block = 0;
            std::set<int>::const_iterator cit;
            while(1)
            {//寻找有空的 block，直到找到为止
                free_block = m_file.findEnoughSpace(free_block + 1, size);
                cit = bufferContainer.find(free_block);
```

```
                    if(cit == bufferContainer.end())
                        break;
                }
                //返回有空的 Block 的地址
                ……
            }
            return i;
        }
        Else//有需要替换的情况，就替换
        {
            int free_block = 0;
            std::set<int>::const_iterator cit;
            while(1)//寻找空 block
            {
                free_block = m_file.findEnoughSpace(free_block + 1, size);
                cit = bufferContainer.find(free_block);
                if(cit == bufferContainer.end())
                    break;
            }
            //调用替换算法
            ……
        }
}

CString BufferManager::readData(Address address, int& size)
{
    int i = readBlockFromDisk(address.blockAddress);
    CString head;
    size = buffer[i].getDataSize(address.indexAddress);
    head = new char[size];
    buffer[i].readinData(address.indexAddress, head);
    return head;
}

void BufferManager::removeData(Address address)
{
    //排除要删除的数据不存在的情况后做假删
    ……
}
```