项目 2：缓冲区管理器（2019 年春）

指导老师邹兆年 [(znzou@hit.edu.cn)](mailto:(znzou@hit.edu.cn)

学生姓名 学生编号 年级

# 导言

BadgerDB 项目的目标是让你亲身体验 RDBMS 的关键组件。在这个项目中，你需要在提供的存储管理器之上实现一个缓冲管理器。

## BadgerDB I/O 层

BadgerDB 数据库系统的最底层是 I/O 层。该层允许系统上层创建/销毁文件、分配/重新分配文件中的页面以及读取/写入文件中的页面。该层由两个类组成：一个文件（类 File）类和一个页面（类 Page）类。这些类使用 C++ 异常处理任何意外事件的发生。我们将为您提供文件类、页面类和异常类的实现。要启动此项目，您可以将压缩文件夹 BufMgr.zip 下载到您的私人工作区，然后使用以下命令解压该文件夹： unzip BufMgr.zip

为帮助您理解代码的工作原理和功能，代码已作了充分的注释。请使用 Doxygen 为代码生成文档，如下所示。在 bufmgr 目录下运行以下命令生成文档文件。

> 制作文档

请注意，">"是 Linux 机器上的 shell 提示符，而不是命令的一部分。文档文件将在 docs 目录中生成。现在，你可以在浏览器中打开 docs/index.html 文件，查看类及其方法的说明，以便更好地理解它们的实现。

## BadgerDB 缓冲管理器

数据库*缓冲*池是一个固定大小的内存缓冲区阵列，称为*帧*，用于存放从磁盘读入内存的数据库*页*（也称为*磁盘块*）。页是磁盘和主内存缓冲池之间的传输单位。大多数现代 DBMS 的页面大小至少为 8,192 字节。另一个需要注意的重要问题是，内存中的数据库页面在首次读入时是磁盘上相应页面的精确副本。页面从磁盘读入缓冲池后，DBMS 软件会更新页面上存储的信息，导致缓冲池中的副本与磁盘上的副本不同。这种页面被称为 "*脏页*"。

由于磁盘上的数据库本身往往大于缓冲池可用的主内存容量，因此在任何时候，内存中都只能容纳数据库页面的一个子集。缓冲区管理器用于控制哪些页面常驻内存。每当缓冲区管理器收到对数据页的请求时，缓冲区管理器都会检查请求的页面是否已经在构成缓冲区池的某个帧中。如果是，缓冲区管理器会简单地返回一个指向该页面的指针。如果不是，缓冲区管理器就会释放一个框架（如果该页面很脏，可能会将其中的页面写入磁盘），然后将请求的页面从磁盘读入已释放的框架中。

在进一步阅读之前，你应该首先阅读介绍 BadgerDB I/O 层的文档，以便了解它的功能（见上一节）。简而言之，I/O 层

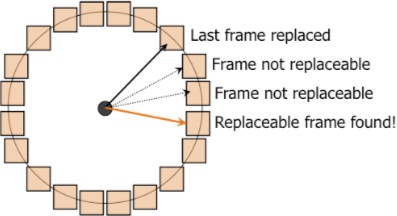


图 1：缓冲区管理器的结构

提供了一个面向对象的 Unix 文件接口，其中包含打开/关闭文件以及读取/写入文件页面的方法。目前，您需要知道的关键是打开文件（通过传递字符串名称）会返回一个 File 类型的对象。该类具有读取和写入文件页面的方法。您将使用这些方法在磁盘和缓冲池之间移动页面。

### 缓冲区替换政策和时钟算法

在需要空闲帧时，决定替换哪个页面的方法有很多。操作系统中常用的策略有 FIFO、MRU 和 LRU。尽管 LRU 是最常用的策略之一，但它的开销很大，在数据库系统中出现的一些常见情况下，它并不是最佳策略。相反，许多系统使用的*时钟算法*近似于 LRU 行为，而且速度更快。图 1 显示了缓冲池的概念布局。在图 1 中，每个方框对应缓冲池中的一个帧。假设缓冲池中包含编号为 0 至 numBufs *-* 1 的 numBufs 帧。从概念上讲，缓冲池中的所有帧都排列在一个循环列表中。与每个帧相关联的位称为 refbit。每次访问缓冲池中的页面时（通过对缓冲区管理器的 readPage() 调用），相应帧的 refbit 都会被设置为 true。在任何时间点，时钟指针（其值介于 0 和 numBufs *-* 1 之间的整数）都会按顺时针方向前进（使用模块化算术，因此不会超过 numBufs *-* 1）。时钟指针每经过一帧，就会检查并清除 refbit。如果该位被置位，则相应的帧 "最近 "被引用，不会被替换。另一方面，如果 refbit 为假，则会选择要替换的页面（假设该页面未被钉住，钉住页面将在下文讨论）。如果被选中的缓冲区帧是脏的（即被修改过），当前占用该帧的页面将被写回磁盘。否则，该帧将被清除，并从磁盘读入一个新页面到该位置。图 2 展示了

时钟算法。时钟算法详情如下。

### 缓冲区管理器的结构

BadgerDB 缓冲区管理器使用三个 C++ 类：BufMgr、BufDesc 和 BufHashTbl。 BufMgr 类只有一个实例。该类的一个关键组件是实际缓冲池，它由一个 numBufs 框架数组组成，每个框架相当于一个数据库页面的大小。除了这个数组，BufMgr 实例还包含一个 BufDesc 类的 numBufs 实例数组，用于描述缓冲池中每个帧的状态。哈希表用于跟踪当前驻留在缓冲池中的页面。这个哈希表由 BufHashTbl 类的一个实例实现。该实例是 BufMgr 类的私有数据成员。下文将详细介绍这些类。

BufHashTbl **类**BufHashTbl 类用于将文件和页码映射到缓冲池帧，并使用链式桶散列实现。我们提供了该类的实现方法供您使用。

struct hashBucket {

File file; // 指向文件对象的指针（下文将详细介绍 PageId pageNo; // 文件中的页码

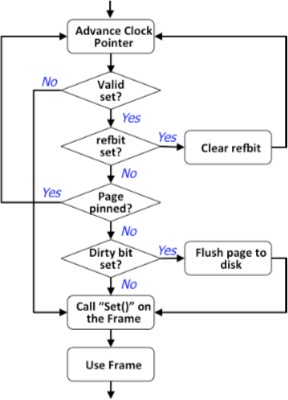


图 2：时钟替换算法

FrameId frameNo; // 缓冲池中页面的帧号 hashBucket next; // 链中的下一个桶

};

下面是哈希表的定义。

类 BufHashTbl

{

私有：

hashBucket\*\* ht; // 指向实际哈希表 int HTSIZE 的指针；

int hash(const File\* file, const PageId pageNo); //returns a value between 0 and HTSIZE-1 public：

BufHashTbl(const int htSize); // 构造函数

~BufHashTbl(); // 析构函数

// void insert(const File\* file, const int pageNo, const int frameNo)；

// 检查（文件、页码号）当前是否在缓冲池中（即在

// 哈希表中。bool lookup(const File\* file, const int pageNo, int& frameNo)；

// void remove(const File\* file, const int pageNo)；

};

BufDesc **类**BufDesc 类用于跟踪缓冲池中每个帧的状态。其定义如下。

首先要注意的是，BufDesc 类的所有属性都是私有的，而 BufMgr 类被定义为好友。虽然这看起来很奇怪，但这种方法限制了只有 BufMgr 类才能访问 BufDesc 的私有变量。而另一种方法（将所有内容都公开）则将访问权限限制得过宽。

BufDesc 类中大多数属性的用途应该是显而易见的。dirty 位（如果为 true）表示页面已脏（即已更新），因此必须在框架用于保存其他页面之前写入磁盘。pinCnt 表示页面被钉住的次数。refbit

refbit 用于时钟算法。有效位用于指示帧是否包含有效页面。无需在该类中实现任何方法。不过，如果您愿意，可以以任何方式对其进行扩充。

类 BufDesc

{

friend class BufMgr; private：

File\* file; // 文件对象指针 PageId pageNo; // 文件中的页面

FrameId frameNo; // 缓冲池帧号

int pinCnt; // 该页面被钉住的次数 bool dirty; // 如果是 dirty 则为 true；否则为 false

bool valid; // 如果页面有效则为 true

bool refbit; // 如果该缓冲区最近被引用过则为 true

void Clear(); // 初始化缓冲区帧

void Set(File\* filePtr, PageId pageNum); // 设置 BufDesc 成员变量值 void Print() // 打印成员变量值

BufDesc(); // 构造函数

};

BufMgr **类**BufMgr 类是缓冲区管理器的核心。你应该在这里编写本项目的新代码。

类 BufMgr

{

私人

FrameId clockHand; // 用于时钟算法的时钟指针

BufHashTbl \*hashTable; // 哈希表映射（文件、页面）到帧号 BufDesc \*bufDescTable; // BufDesc 对象，每帧一个

std::uint32\_t numBufs; // 缓冲区池中的帧数 BufStats bufStats; // 缓冲区池使用情况的统计数据

void allocBuf(FrameId & frame); // 使用时钟算法分配空闲帧 void advanceClock(); // 将时钟提前到缓冲池中的下一帧

公众：

Page \*bufPool; // 实际缓冲池

BufMgr(std::uint32\_t bufs); // 构造函数

~BufMgr(); // 析构函数

void readPage(File\* file, const PageId PageNo, Page\*& page)；

void unPinPage(File\* file, const PageId PageNo, const bool dirty); void allocPage(File\* file, PageId& PageNo, Page\*& page)；

void disposePage(File\* file, const PageId pageNo); void flushFile(const File\* file)；

};

该类的定义如下

* BufMgr(const int bufs)

这是类的构造函数。它将为缓冲池分配一个数组，其中包含 bufs 页面框架和相应的 BufDesc 表。根据设置，在分配缓冲池时，所有帧都将处于清空状态。哈希表一开始也将处于空状态。我们提供了构造函数。

* ~BufMgr()

清空所有脏页面，并停用缓冲池和 BufDesc 表。

* void advanceClock()

将时钟前进到缓冲池中的下一帧。

* void allocBuf(FrameId& frame)

使用时钟算法分配空闲帧；必要时，将脏页写回磁盘。如果所有缓冲区框架都被钉住，则抛出 BufferExceededException。该私有方法将被下面描述的 readPage() 和 allocPage() 方法调用。请确保如果分配的缓冲帧中有有效页面，则从哈希表中删除相应的条目。

* void readPage(File\* file, const PageId PageNo, Page\*& page)

首先，通过调用哈希表上的 lookup() 方法（当页面不在缓冲池中时，该方法可能会抛出 HashNotFoundException）检查页面是否已在缓冲池中，以获取帧号。根据 lookup() 调用的结果，有两种情况需要处理：

* + 情况 1：页面不在缓冲池中。调用 allocBuf() 分配缓冲区框架，然后调用 file->readPage() 方法将页面从磁盘读入缓冲池框架。然后，将页面插入哈希表。最后，在框架上调用 Set() 方法对其进行正确设置。Set() 会将页面的 pinCnt 设置为 1。通过页面参数返回包含页面的框架指针。
  + 情况 2：页面在缓冲池中。在这种情况下，设置相应的 refbit，递增 pinCnt

页面的指针，然后通过页面参数返回一个指向包含该页面的框架的指针。

* void unPinPage(File\* file, const PageId PageNo, const bool dirty)

减少包含（file, PageNo）的帧的针脚数，如果 dirty == true，则设置 dirty 位。如果针脚数已为 0，则抛出 PAGENOTPINNED。 如果在哈希表中未找到页面，则不做任何操作。

* void allocPage(File\* file, PageId& PageNo, Page\*& page)

该方法的第一步是通过调用 file->allocatePage() 方法在指定文件中分配一个空页面。该方法将返回一个新分配的页面。然后调用 allocBuf() 获得一个缓冲池框架。然后，在哈希表中插入一个条目，并调用 Set() 方法对框架进行正确设置。该方法通过 pageNo 参数向调用者返回新分配页面的页码，并通过 page 参数返回为该页面分配的缓冲区框架指针。

* void disposePage(File\* file, const PageId pageNo)

该方法从文件中删除特定页面。在从文件中删除页面之前，如果要删除的页面在缓冲池中分配了一个帧，则要确保释放该帧，并相应地删除哈希表中的条目。

* void flushFile(File\* file)

应扫描 bufTable 以查找属于文件的页面。对于遇到的每个页面，它应该(a) 如果页面是脏的，则调用 file->writePage() 将页面刷新到磁盘，然后将页面的脏位设置为 false；(b) 从 hashtable 中删除页面（无论页面是干净的还是脏的）；(c) 为页面帧调用 BufDesc 的 Clear() 方法。

如果文件的某个页面被钉住，则抛出 PagePinnedException。如果遇到属于文件的无效页面，则抛出 BadBufferException。

# 入门

解压 BufMgr.zip 文件后，您将看到一个名为 bufmgr 的目录。 在该目录中，您将找到以下文件：

* Makefile 文件：一个 make 文件。您可以在 shell 中输入 make 来制作项目。
* main.cpp：驱动程序文件。展示如何使用文件和页面类。还包含缓冲区管理器的简单测试用例。您必须使用更严格的测试套件来增强这些测试。
* buffer.h：缓冲区管理器的类定义
* buffer.cpp：方法的骨架实现。请在此处提供您的实际实现。
* bufHash.h：缓冲池哈希表类的类定义。请勿更改。
* bufHash.cpp：缓冲池哈希表类的实现。请勿更改。
* file.h：文件类的类定义。不应更改此文件。
* file.cpp：文件类的实现。不应更改此文件。
* 文件 iterator.h：文件中页面的迭代器实现。请勿更改。
* page.h：页面类的类定义。请勿更改。
* page.cpp：页面类的实现。请勿更改。
* page iterator.h：页面记录迭代器的实现。
* 异常目录：所有异常类的实现。如有需要，可在此添加更多文件。

# 编码和测试

我们制定本项目的目的是让您了解并充分享受使用 C++ 进行面向对象编程的好处。您的编码风格应延续这种风格，拥有定义明确的类和简洁的接口。不建议恢复 C（低级过程）编程风格，否则会受到惩罚。代码应使用 Doxygen 风格的注释进行详细记录。每个文件都应以姓名和学生 ID 开头，并说明文件的目的。每个函数前都应有几行注释，说明该函数，并解释输入和输出参数及返回值。

# 上交

您需要将所有必要材料压缩在一个文件夹中提交（使用 GZip 或 WinZip）。您的文件夹应命名为"<学生编号> <学生姓名> Proj2"，并只包含源代码文件（无二进制文件）。我们将编译您的缓冲区管理器实现，将其与我们的测试驱动程序链接并运行测试。由于我们应该能够使用任何有效的驱动程序测试您的代码，因此必须严格遵守此处指定的接口定义。如果您修改了这些接口，导致代码无法编译，您将受到处罚。

# 物流

BadgerDB 以 C++ 编写，可在 Linux 机器上运行。这里有几个后勤要点：

* **警告：**C++ 的优势之一是在编译时进行代码检查（从而减少运行时的错误）。请尽可能多地打开编译器警告，以利用这一优势。我们将提供的 Makefile 默认开启 -Wall。
* **辅助工具：**时刻留意能简化工作的工具。例如：用于编译和构建项目的 make、用于自动生成依赖关系的 makedepend、用于编写测试脚本的 perl 或 python 或 bash、用于跟踪内存错误的 valgrind、用于调试的 gdb 和用于版本控制的 git。
* **软件工程：**像这样的大型项目需要大量的软件设计工作。在开始编写任何代码之前，请花一些时间考虑您的整体方法。

# 检查点

* 收到数据页请求时，是否能返回正确的结果：请求的页在pool中，返回指针；不在，是否正确更新并返回
* Buﬀer Manager的结构是否正确
* 是否正确访问buffer pool中的页面
* 时钟算法的过程是否正确：时钟指针的检查过程是否正确，页面替换是否正确，是否正确将数据写回磁盘
* 良好的编程风格：面向对象，Doxygen风格的注释