实验六：

HuaDB数据库内核实验



本实验概览图

# HuaDB数据库内核实验手册

* 1. 实验介绍
     1. 关于本实验

本实验主要描述了数据库系统页面管理模块的缓存管理和页面组织，旨在通过实现记录的页面组织和经典的最近最少使用（LRU）缓存替换算法，以加深对数据库页面管理功能的理解。实验重点在于理解页面缓存管理以实现高效的磁盘页面访问和不同数据页面的格式化处理。其中，页面替换算法和高效的页面组织是实验的关键挑战，需要考虑不规则的页面访问模式和多种数据页面类型。实验的核心任务是完成变长形式记录数据的页面组织和LRU缓存替换算法。图1为数据库系统整体层次划分图，左侧为功能示意图，右侧为层次架构图。

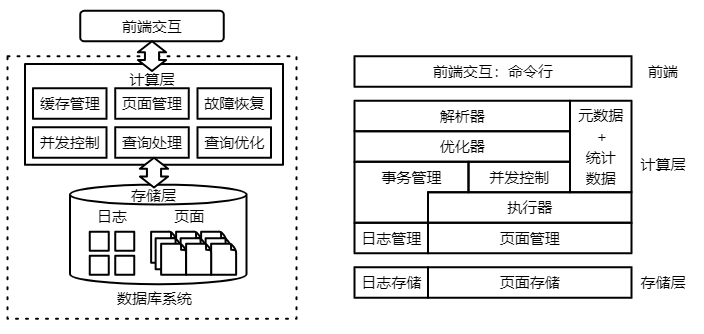


图1 数据库系统整体层次划分图

* + 1. 实验目的
* 理解数据库系统页面管理模块的核心功能；
* 学习如何有效地组织变长形式记录数据的页面；
* 了解如何管理页面缓存，以实现高效的磁盘页面访问。
  1. 实现页面组织与缓存管理功能概述

完成实验1页面组织与缓存管理需要实现两个基础功能，一是实现变长记录的页面组织，二是实现LRU缓存替换策略。在介绍1.4实验步骤前，简要介绍两个基础功能原理、理论知识以及实验设计思路。

* + 1. 实现变长记录的页面组织功能

本实验要求实现的第一个功能目标是实现变长记录的页面组织功能，这一功能对于理解数据库系统中的页面管理模块至关重要。页面管理是数据库系统中一个基本且复杂的组成部分，它负责内存和磁盘之间的数据交换，以及将不同类型的数据页面转换成特定的格式化数据。下面将简要概述变长记录的页面组织原理和实验设计思路。

1. **变长记录的页面组织原理**

页面管理负责组织加载页面，管理内存以及磁盘之间的数据移动。需要关注的两个关键问题，即如何在磁盘上存储和组织这些页面，以及如何将它们加载到内存中进行更快的访问。磁盘和内存中都是通过页来组织的，页包含很多记录：元组、系统表、索引、日志，通过record\_id或者tuple\_id来作为元组唯一标识，每一个页表由两部分组成:page\_header和data。Data存储真正的数据，包含元组，索引，日志，元数据等，常用面向元组方法或基于日志结构化存储表示数据。然而，使用面向元组方法存储数据存在一些不足，其中之一是无法处理变长数据记录，为了解决此问题，产生了slotted pages。

Slotted page是数据库中页的常见组织形式，其优点是可以存储可变长的元组，提高存储效率，同时支持记录的灵活管理（增删改查）。涉及到对不同大小记录的存储和定位。一种常见方法是使用指针数组或链表来管理这些记录，同时动态调整页面内部空间以优化存储。通过page\_id和slot\_id可以查询磁盘上任意一个元组的内容。当新增一条记录时，会在slot array中新增一条记录，记录修改的record\_address， slot array与data从page两端向中间生长，二者相遇时,代表该页已满，此时需要缓存替换策略淘汰部分元组以替换更新，工作原理图如图2所示，页面组织方式如图3所示，page\_header包含字段如表1所示。

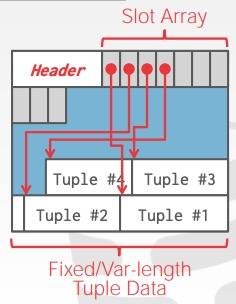


图2 变长记录元组数据工作原理图

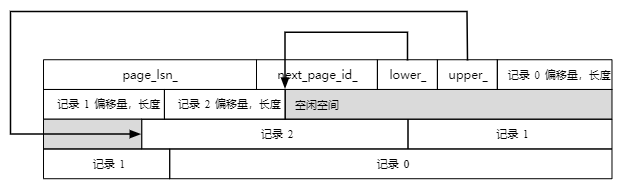


图3 页面组织方式

表1 page\_header字段表

| **变量名** | **变量类型** | **长度** | **作用** |
| --- | --- | --- | --- |
| deleted\_ | bool | 1 | 标注记录是否删除 |
| xmin\_ | xid | 4 | （实验 3）插入该数据的事务 id |
| xmax\_ | xid | 4 | （实验 3）删除该数据的事务 id |
| cid\_ | cid\_t | 4 | （实验 3）插入该数据的事务内部 command id |

1. **实验设计思路**

变长记录的页面组织需支持增删改查功能，即记录插入、记录读取、记录删除和系统表变更。本文将从这4个步骤简要概述任务1实现思路。

* + - 1. 记录插入
* 判断空闲空间是否足够容纳新记录，没有则新建页面
* 寻找合适的插入位置
  + 可通过页面槽slots\_数组确定空闲槽位
  + 先遍历槽数组，找到第一个空闲槽位
* 找到空闲槽位后，拷贝新记录数据到该槽位位置。同时更新空闲指针lower,upper，并标记该槽位已占用（不会如何更新offset)
* 更新页面状态，将page标记位dirty
  + - 1. 记录读取
* 通过 TableScan 的 GetNextRecord 函数，维护rid\_成员变量来追踪当前读取的记录位置
* 调用 TablePage 的 GetRecord 函数，根据 slot\_id 和表的结构从页面中反序列化得到记录
* 当表中所有数据读取结束时，返回空指针表示读取结束
  + - 1. 记录删除
* 在 table.cpp 和 table\_page.cpp 中实现 DeleteRecord 函数
* 使用 buffer\_pool\_ 获取要删除记录的页面，并调用 TablePage 的 DeleteRecord 函数
* 实现标记删除，即设置记录头部的 deleted\_ 变量
* 在 table\_scan.cpp 的 GetNextRecord 函数中，排除已标记为删除的记录
  + - 1. 系统表变更
* 更改编译选项，使数据库使用 SystemCatalog 代替 SimpleCatalog
  + 1. 实现LRU缓存替换策略功能

LRU策略是一种常用的缓存替换算法，用于管理缓存中的数据项。其核心思想是：

* 追踪每个数据项的访问时间：每当一个数据项被访问时，记录这个访问的时间或顺序。
* 淘汰最久未被访问的数据项：当缓存空间不足时，选择最久未被访问的数据项进行淘汰。
  + 1. 代码结构

实验框架主要包含以下几个模块，加粗部分为实验中涉及到的，需要你来补充的模块：

* binder: 语义解析模块
* catalog: 系统表模块
* common: 工具模块，包含字符串处理函数、异常相关类等
* database: 数据库引擎
* **executors: 查询执行模块**
* **log: 日志模块**
* operators: 查询计划树节点
* **optimizer: 优化器**
* planner: 查询计划生成模块
* **storage: 存储模块**
* **table: 表相关类及函数**
* **transaction: 事务模块**
  + 1. 相关代码模块

本次实验涉及到代码中如下的功能模块：

|  |
| --- |
| * log:日志相关类   + [log\_record](https://github.com/thu-db/huadb/tree/main/src/log/log_record.h)：各类日志的抽象类，用于衍生出各种具体日志，已经完成。   + log\_manager：日志管理器，负责日志的记录以及故障恢复的具体执行过程，需要补充恢复算法。   + logs\_records：各类具体的物理日志类，需要补充其中记录增删改日志的重做和撤销。 * table:数据表相关类   + table：实验 1 的基础上添加记录变更时的日志记录功能。 |

基础功能需要补充约 200 行代码，本次实验依赖于实验 1，请确保完成以上任务再开始本次实验。

相关功能模块的抽象示意图如图4：

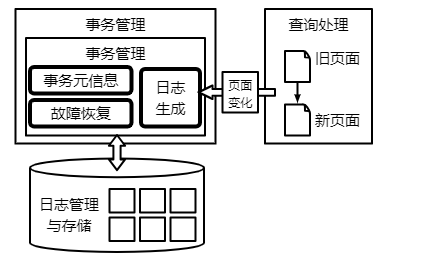


图4 相关功能模块的抽象示意图

* 1. 前置步骤

本实验均在HuaDB上进行，由C++语言编写，支持基础的页面存储、故障恢复、缓存管理、查询处理等功能。该实验提供HuaDB基础实验框架，在实验1中，需要在框架中填充相应的函数实现代码，使之通过该次实验的所有测例。实验前，需要先执行环境配置、克隆仓库、编译及测试等步骤。

### 环境配置

HuaDB 使用了 C++17 标准，开始实验前，请确保你的开发环境支持 C++17 标准。

目前 HuaDB 仅支持 macOS 和 Linux 操作系统，使用 Windows 的同学建议使用虚拟机进行实验。

HuaDB 代码下载与提交需要使用 [git](https://git-scm.com/) 工具，代码编译需要使用 [CMake](https://cmake.org/) 及 [Make](https://www.gnu.org/software/make/) 工具，且需要安装 [gcc](https://gcc.gnu.org/) 或 [clang](https://clang.llvm.org/) 编译器。此外，代码调试中可能会用到调试器 [gdb](https://www.sourceware.org/gdb/) 或 [lldb](https://lldb.llvm.org/)。开始实验前，请确保你的开发环境安装了这些工具并可以正常使用，如没有，请根据你使用的操作系统选择对应的命令进行环境配置。

Ubuntu

|  |
| --- |
| sudo apt install git g++ make cmake gdb |

### 克隆仓库

从GitHub克隆代码，命令如下：

|  |
| --- |
| git clone https://github.com/thu-db/huadb.git |

克隆成功后，进入huadb目录下：

|  |
| --- |
| cd huadb |

### 编译安装

修改环境变量，命令如下：

|  |
| --- |
| vim CMakeLists.txt  set(CMAKE\_CXX\_FLAGS "-pthread")  CMAKE\_BUILD\_PARALLEL\_LEVEL=$(nproc) make |

通过以上指令可以编译出 debug 版本的程序，为便于调试，不建议编译 release 版本。

编译生成的文件位于 build/debug 目录，实验过程中只需要关注 build/debug/bin 目录中的可执行程序即可，具体包括：

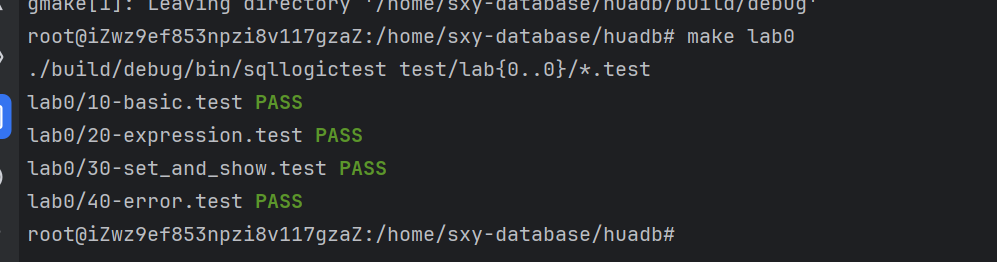
1. shell: 数据库程序，运行后可以与数据库进行交互。
2. sqllogictest: 测试程序，用于批量测试。

### 验证测试

运行如下命令来验证你编译出的数据库程序和测试程序可以正常运行：

|  |
| --- |
| make lab0 |

如果产生如下输出，表示程序正常运行。



|  |
| --- |
| Test: 1/3  lab0/10-basic.test PASS  lab0/20-expression.test PASS  lab0/30-set\_and\_show.test PASS  lab0/40-error.test PASS  Test: 2/3  lab0/10-basic.test PASS  lab0/20-expression.test PASS  lab0/30-set\_and\_show.test PASS  lab0/40-error.test PASS  Test: 3/3  lab0/10-basic.test PASS  lab0/20-expression.test PASS  lab0/30-set\_and\_show.test PASS  lab0/40-error.test PASS |

如果程序报错，可以根据报错信息对照[测试说明](https://thu-db.github.io/huadb-doc/overview)进行排查。

此外，你可以通过如下命令进入数据库交互界面：

|  |
| --- |
| make shell |

你可以在交互界面中运行一些基础的 DDL 命令，如：

|  |
| --- |
| Welcome to HuaDB. Type "\?" or "\h" for help  huadb> \?  \? or \h show help message  \c [database\_name] change database  \d show tables  \d [table\_name] describe table  \l show databases  \q quit  huadb> \l  +---------------+  | database\_name |  +---------------+  | tmp |  +---------------+  (1 row)  huadb> create table test(id int, info varchar(10));  CREATE TABLE  huadb> \d  +------------+  | table\_name |  +------------+  | test |  +------------+  (1 row)  huadb> \d test  +------+---------+------+  | name | type | size |  +------+---------+------+  | id | int | 4 |  | info | varchar | 10 |  +------+---------+------+  (2 rows)  huadb> drop table test;  DROP TABLE  huadb> \d  +------------+  | table\_name |  +------------+  (0 rows)  huadb> \q |

* 1. 实验步骤
     1. 变长记录页面组织

**步骤1：记录插入**

实现记录插入功能位于代码结构table类table.cpp，观察此结构代码可发现函数调用关系如图 所示。用户输入插入语句，数据库系统会将插入操作通过查询处理器的SQL引擎进行词法语法树转换，又通过执行引擎计算表达式。在这一步，记录插入的上层调用位于执行层insert\_executor.cpp，调用了Table类的InsertRecord函数，而该函数底层调用的是TablePage类的InsertRecord函数实现该功能。在记录插入部分，需要完成Table类InsertRecord和TablePage类InsertRecord两个函数实现。根据函数调用关系图，我们可以采用自上而下的形式编写逻辑，方便我们梳理逻辑结构。

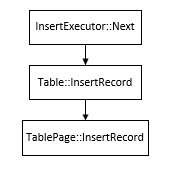


图 记录插入函数调用关系图

在Table类InsertRecord函数中，实现思路首先需要通过buffer\_pool\_获取该页面，判断比较该页面大小和插入记录的大小，如果空间不足，则需调用BufferPool的NewPage函数新建页面，并在新页面中插入记录。如果页面空闲空间足够放下要插入的记录，则进一步调用页面TablePage的InsertRecord函数插入记录。实现流程如下：

1. 获取页面

数据存储是通过表存储在磁盘中，每一张表是通过唯一标识符page\_id识别，页面之间是通过链表实现，获取页面传入的page\_id则是Table定义的第一个页面的页面号first\_page\_id， 通过first\_page\_id与其他页面进行链接。

|  |
| --- |
| pageid\_t page\_id=first\_page\_id\_;  auto page=buffer\_pool\_.GetPage(db\_oid\_,oid\_,first\_page\_id\_); |

获取的页面是BufferPool类，需转换为TablePage类以进行记录插入操作。

|  |
| --- |
| auto table\_page=std::make\_unique<TablePage>(page); |

1. 遍历页表，判断页面空间是否足够插入记录,并进行插入操作调用。

|  |
| --- |
| while(table\_page->GetFreeSpaceSize()<record->GetSize()&& table\_page->GetNextPageId()!=NULL\_PAGE\_ID){  page\_id++;  page = buffer\_pool\_.GetPage(db\_oid\_,oid\_,page\_id);  table\_page=std::make\_unique<TablePage>(page);  }  if(table\_page->GetFreeSpaceSize()<record->GetSize() && table\_page->GetNextPageId()==NULL\_PAGE\_ID){  page\_id++;  // 创建新页面时需设置当前页面的 next\_page\_id，并将新页面初始化  table\_page->SetNextPageId(page\_id);  table\_page=std::make\_unique<TablePage>(buffer\_pool\_.NewPage(db\_oid\_,oid\_,page\_id));  table\_page->Init();  }  // 找到空闲空间，通过 TablePage 插入记录  slotid\_t slot\_id=table\_page->InsertRecord(record,xid,cid);  return {page\_id, slot\_id}; |

在TablePage类InsertRecord函数中，实现关键在于理解页面槽，如何维护lower和upper指针。首先需要获取记录数量作为槽大小，lower是页面空闲空间的起始位置，upper是页面空闲空间终止位置。其次，维护指针则需要lower往后移动一个槽大小，upper往前移动一个槽大小。最后，将记录序列化写入data，并标记为dirty，返回槽号slot\_id。

|  |
| --- |
| slotid\_t slot\_id=GetRecordCount();  //lower需要往后移动一个槽的大小;upper需要往前移动一个记录的大小  Slot\* slot= reinterpret\_cast<Slot \*>(page\_data\_+\*lower\_);  slot->size\_=record->GetSize();  \*lower\_+=sizeof(Slot);  \*upper\_-=slot->size\_;  slot->offset\_=\*upper\_;  char \* record\_begin=reinterpret\_cast<char \*>(page\_data\_+\*upper\_);  // 将 record 写入 page data  record->SerializeTo(record\_begin);  // 将 page 标记为 dirty  page\_->SetDirty();  return slot\_id; |

**步骤2：记录读取**

记录读取需要补全TableScan::GetNextRecord和TablePage 的 GetRecord两个函数。在GetNextRecord函数中，需要维护 rid\_ 成员变量，表示目前读取到的记录的 rid，这是因为上层会多次调用 GetNextRecord 函数，每次返回一条记录，需要用 rid\_ 成员变量记录目前读取到了哪一条记录，下次读取时返回下一条记录。维护 rid\_变量之后，调用 TablePage 的 GetRecord 函数获取记录。当表中所有数据读取结束时，返回空指针，表示结束读取。在GetRecord函数中，根据 slot\_id 和表的结构 column\_list，从页面中反序列化得到记录。

TableScan::GetNextRecord函数实现步骤如下：

* 检查当前页是否为空：如果rid\_.page\_id\_是NULL\_PAGE\_ID，表示当前页为空，此时扫描结束，函数返回空指针。
* 获取并处理当前页：如果当前页不为空，通过buffer\_pool\_获取当前页(rid\_.page\_id\_)对应的Page对象。然后，使用此Page对象创建一个TablePage对象。
* 读取记录：使用TablePage对象和当前的槽位ID (rid\_.slot\_id\_)来获取当前页的记录，并将这个记录封装成一个Record对象。
* 更新rid\_变量：为了避免重复读取同一条记录，更新rid\_变量，指向下一条待读取的记录。首先检查下一个槽位ID是否超过当前页的记录总数：
  + 如果没有超过，将rid\_.slot\_id\_递增，指向下一个槽位。
  + 如果超过，意味着需要读取下一页，因此更新rid\_.page\_id\_为下一页的ID，并将槽位ID重置为0。
* 返回记录：最后，返回当前读取的记录。

|  |
| --- |
| std::shared\_ptr<Record> TableScan::GetNextRecord(xid\_t xid, IsolationLevel isolation\_level, cid\_t cid,const std::unordered\_set<xid\_t> &active\_xids) {  // LAB 1 BEGIN  if(rid\_.page\_id\_ == NULL\_PAGE\_ID){  return nullptr;  }  std::shared\_ptr<Page>page= buffer\_pool\_.GetPage(table\_->GetDbOid(),table\_->GetOid(),rid\_.page\_id\_);  auto table\_page = std::make\_unique<TablePage>(page);  std::shared\_ptr<Record> record = table\_page->GetRecord(rid\_.slot\_id\_, table\_->GetColumnList());  record->SetRid({rid\_.page\_id\_,rid\_.slot\_id\_});  if (rid\_.slot\_id\_+1<table\_page->GetRecordCount()){  rid\_.slot\_id\_ ++;  }else{  rid\_.page\_id\_ = table\_page->GetNextPageId();  rid\_.slot\_id\_=0;  }  return record;  } |

TablePage::GetRecord函数需要实现插入新纪录的逻辑，主要步骤如下：

* 确定槽位ID：通过调用GetRecordCount()获取当前页中已有记录的数量，该数量也作为新记录的槽位ID（slot\_id）。
* 更新lower和upper指针：在页中，lower指向槽的开始位置，而upper指向记录数据的末尾。为了添加新记录，需要移动这两个指针：
  + lower指针向前移动一个Slot的大小，以腾出空间放置新槽的信息。
  + upper指针向后移动新记录的大小，为新记录数据腾出空间。
* 设置新槽的信息：新槽的信息包括记录的大小和在页中的偏移量。这些信息被写入到lower指针指向的位置。
* 写入记录数据：将记录的数据序列化（SerializeTo）并写入到页中upper指针指向的位置。
* 标记页为脏：由于页的内容发生了变化，所以将其标记为脏（SetDirty），这意味着它需要在某个时刻被写回到磁盘。
* 返回槽位ID：函数返回新插入记录的槽位ID。

|  |
| --- |
| slotid\_t TablePage::InsertRecord(std::shared\_ptr<Record> record, xid\_t xid, cid\_t cid) {  slotid\_t slot\_id=GetRecordCount();  Slot\* slot= reinterpret\_cast<Slot \*>(page\_data\_+\*lower\_);  slot->size\_=record->GetSize();  \*lower\_+=sizeof(Slot);  \*upper\_-=slot->size\_;  slot->offset\_=\*upper\_;  char \* record\_begin=reinterpret\_cast<char \*>(page\_data\_+\*upper\_);  record->SerializeTo(record\_begin);  page\_->SetDirty();  return slot\_id;  } |

**步骤3：记录删除**

在记录删除这步中，需要实现三个函数：table.cpp的DeleteRecord函数、table\_page.cpp的DeleteRecord函数和table\_scan.cpp的GetNextRecord函数。table.cpp 的 DeleteRecord 函数的实现思路与记录插入类似，只需通过 buffer\_pool\_ 获取需要删除的记录对应的页面，调用页面 TablePage 的 DeleteRecord 函数即可。table\_page.cpp的DeleteRecord 函数实现了页面中记录的删除，在实验的基础功能中，仅要求实现标记删除即可。只需设置记录头部的 deleted\* 变量即可。完成上述步骤后，还需要修改 table\_scan.cpp 的 GetNextRecord 函数，读取记录时判断记录是否已经被标记为删除，不再返回已经删除的数据。

Table::DeleteRecord

|  |
| --- |
| void Table::DeleteRecord(const Rid &rid, xid\_t xid, bool write\_log) {  auto table\_page = std::make\_unique<TablePage> (buffer\_pool\_.GetPage (this->db\_oid\_,this->oid\_,rid.page\_id\_) );  table\_page->DeleteRecord(rid.slot\_id\_, xid);  } |

TablePage::DeleteRecord

|  |
| --- |
| void TablePage::DeleteRecord(slotid\_t slot\_id, xid\_t xid) {  Slot\* slot = slots\_ + slot\_id;  char \* record\_data = reinterpret\_cast <char \*> ( page\_data\_ + slot -> offset\_);  std::unique\_ptr<Record> record = std::make\_unique<Record>();  record->DeserializeHeaderFrom(record\_data);  record->SetDeleted(true);  record->SerializeHeaderTo(record\_data);  } |

在table\_scan.cpp的GetNextRecord函数增加删除部分

|  |
| --- |
| if(record->IsDeleted()){  return GetNextRecord(xid, isolation\_level, cid,active\_xids);  } |

**步骤4：系统表变更**

此部分不需要添加任何代码，仅需更改编译选项即可。

在完成步骤 1-3 之前，由于数据库不具备基础的增删改查功能，存储元信息的系统表无法正常工作，我们使用了 SimpleCatalog 临时替代了系统表功能。但是 SimpleCatalog 没有数据库的增加/删除/切换功能，导致我们无法通过 40-database.test 测例。因此，我们需要改变编译选项，让我们的数据库使用系统表 SystemCatalog。

首先清理此前生成的编译文件及数据库文件：

|  |
| --- |
| make destroy && make clean |

之后编译使用 SystemCatalog 的数据库：

|  |
| --- |
| make debug |

编译完成后，再次对 lab 1 测例进行测试，可以通过 40-database.test。

此后的所有实验，我们只需要使用 SystemCatalog 版本的数据库即可，不再需要使用 make lab1-debug 命令。你可以修改 Makefile 文件，将：

|  |
| --- |
| all: lab1-debug |

修改为:

|  |
| --- |
| all: debug |

以避免在之后的实验中编译出错误的版本。

* + 1. LRU缓存替换策略

在任务 1 中，我们多次调用 BufferPool 的 GetPage 和 NewPage 函数，本任务中，你需要阅读 BufferPool 中这两个函数的实现代码，并为缓存池添加 LRU 替换算法。你只需修改 lru\_buffer\_strategy.cpp 即可，必要时可以在 lru\_buffer\_strategy.h 中添加成员变量。整体代码量很小，但希望通过此任务加深对缓存池的理解，这将帮助你更快地之后的实验。

BufferPool 的主要成员变量有三个：页面缓存 buffers\_ ，页面号到 buffers\_ 下标的映射表 hashmap\_ 以及缓存替换策略成员 buffer\_strategy\_。

其他成员变量主要为系统表相关变量，你无需过多关注。

页面缓存大小默认为 5，每次访问页面时会将页面放入缓存区，并调用 BufferStrategy 的 Access 函数。若缓存区已满，则需要调用 BufferStrategy 的 Evict 函数获取淘汰页面的编号。

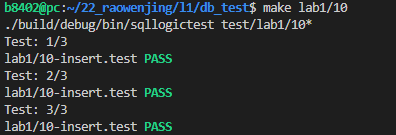
本任务中，你需要实现 LRUBufferStrategy 的 Access 和 Evict 函数，实现 LRU 缓存替换算法，正确实现后你将通过50-buffer\_pool.test测例。

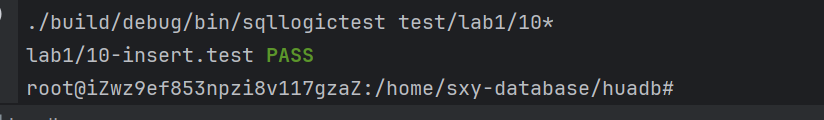
**lru\_buffer\_strategy.cpp**

页面缓存大小默认为 5，每次访问页面时会将页面放入缓存区，并调用 BufferStrategy 的 Access 函数。若缓存区已满，则需要调用 BufferStrategy 的 Evict 函数获取淘汰页面的编号。

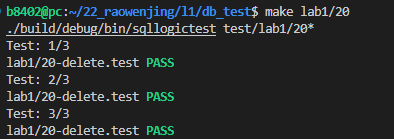
|  |
| --- |
| #include "storage/lru\_buffer\_strategy.h" #include "common/constants.h" namespace huadb { LRUBufferStrategy::LRUBufferStrategy() {  time=new size\_t[BUFFER\_SIZE] ;  for (size\_t i = 0; i < BUFFER\_SIZE; ++i) {  time[i]=-1;   } } void LRUBufferStrategy::Access(size\_t frame\_no) {  for(size\_t i=0;i<BUFFER\_SIZE;i++){   if (time[i]!=-1){  time[i]++;  }  }  time[frame\_no]=0;  return; };  size\_t LRUBufferStrategy::Evict() {  size\_t max\_index=0;  for (size\_t i = 1; i < BUFFER\_SIZE; ++i) {  if (time[max\_index]<time[i]){  max\_index=i;  }  }  return max\_index; } } // namespace huadb |

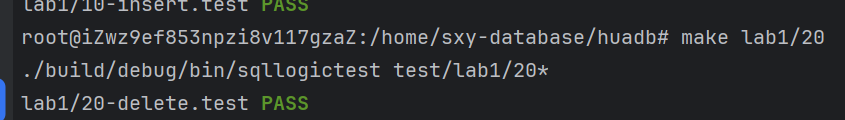
* 1. 实验测试
     1. 记录插入与读取

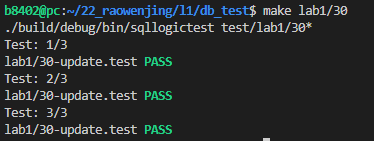


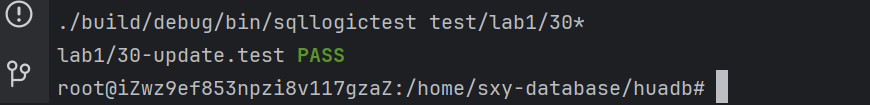


* + 1. 记录删除

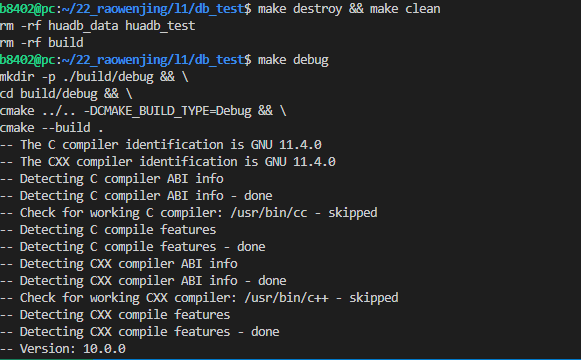


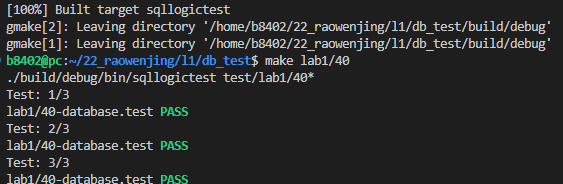


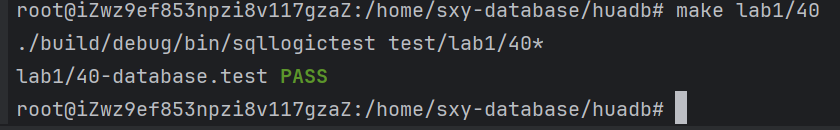




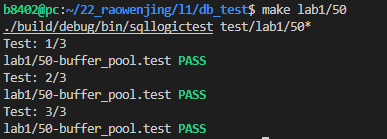
* + 1. 系统表变更

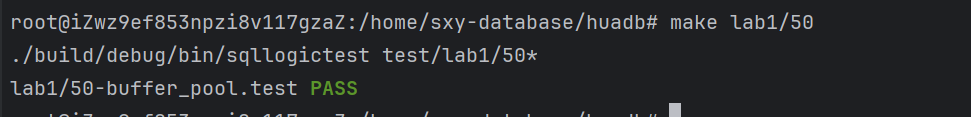






* + 1. LRU缓存替换策略





# 课程报告模板

**数据库课程实验报告**

**(2023-2024学年第一学期)**

**提交日期：年 月 日 本科生姓名：**

|  |  |  |  |
| --- | --- | --- | --- |
| **学 号** |  | **学 院** |  |
| **课程名称** |  | **任课老师** |  |
| **教师评语：** | | | |
| **成绩评定： 分 任课教师签名： 年 月 日** | | | |

# 实验六：HuaDB 数据库内核实验

参考https://thu-db.github.io/huadb-doc/的实验一部分

https://gitee.com/thu-db/huadb

## 一、实验要求

1、完成[实验 1: 缓存管理与页面组织](https://thu-db.github.io/huadb-doc/lab1/1-intro/)的基础功能部分，将实验过程记录下来。

2、在实验过程中你遇到什么困难和问题，你怎么解决的？（没有则可以不写）

扩展：[尝试实现高级功能](https://thu-db.github.io/huadb-doc/lab1/3-advanced/)，若无法实现，说说你的遇到的问题。