**MiniSQL总体设计报告**

# MiniSQL系统概述

## 背景

### 编写目的

如果将数据库比作一辆汽车，那我们对SQL语言的理论学习，仅仅是学会了如何驾驶车辆，只是能够使用一些基础的SQL语言以及进行一些数据库的操作。而我们对于数据库系统的理解，还是颇为浅显。

而编写miniSQL的过程，就好比造车的过程。从中我们可以学习到数据库系统（DBMS）的运行原理，从最基础的内存调控、记录处理一层层向上叠加，到索引建立与搜索，再到执行方案生成选择与执行。编写的过程中，是对简单数据库实现原理的学习，也在很大程度上深化了数据库理论知识的理解。对我们实践的锻炼，与对数据库内涵的感悟，都有极大的帮助。

### 项目背景

数据库系统设计实验。

## 功能描述

1）总功能：允许用户通过字符界面输入SQL语句实现表的建立/删除；索引的建立/删除以及表记录的插入/删除/查找；

2）数据类型：支持三种基本数据类型：integer，char(N)，float；

3）表定义：一个表可以定义多达32个属性，各属性可以指定是否为unique，支持单属性的主键定义；

4）索引的建立和删除：对于表的主属性自动建立 B+树索引（单属性单值），对于声明为unique的属性可以通过SQL语句由用户指定建立/删除B+树索引；

5）查找记录：可以通过指定用AND连接的多个条件进行查询，支持等值查询和区间查询；

6）插入和删除记录：支持每次一条记录的插入操作和每次一条或多条记录的删除操作。

## 运行环境和配置

MacOS 或 使用Clion连接Win10下WSL。

## 参考资料

框架来源：ZJUCSDB-MiniSQL

地址：https://www.yuque.com/yingchengjun/minisql

# MiniSQL系统结构设计

## 总体设计：

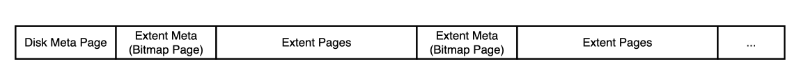
如上图所示的系统架构，解释器SQL Parser在解析SQL语句后将生成的语法树交由执行器Executor处理。执行器则根据语法树的内容对相应的数据库实例（DB Storage Engine Instance）进行操作，而每个DB Storage Engine Instance对应了一个数据库实例（即通过CREATE DATABSAE创建的数据库）。在每个数据库实例中，用户可以定义若干表和索引，表和索引的信息通过Catalog Manager、Index Manager和Record Manager进行维护。

（具体实现细节在下述分模块以及分模块子实验报告中作详细阐述）

## Disk Manager 模块

在MiniSQL的设计中，Disk Manager模块位于架构的最底层。主要负责数据库文件中数据页的分配和回收，以及数据页中数据的读取和写入。其中，数据页的分配和回收通过位图（Bitmap）这一数据结构实现，位图中使用一个Bit对应一个数据页，用于标记该数据页是否空闲（0表示空闲，1表示已分配）。

基于bitmap，我们可以很方便地使用一个位图页加上一段连续的数据页（数据页的数量取决于位图页最大能够支持的比特数）来对磁盘文件中数据页进行分配和回收。为了扩充我们可以表示的磁盘空间，我们还使用一个位图页加一段连续的数据页看成数据库文件中的一个分区，再通过一个额外的元信息页来记录这些分区的信息。通过这种“套娃”的方式，来使磁盘文件能够维护更多的数据页信息，实现如下图所示。

在本部分，主要实现了数据页的分配、删除以及判断数据页是否处于free状态的函数；在Bitmap文件中定义的函数，主要供我们后来提出的“套娃”管理方式调用，因而产生了相应的DiskManager文件的数据页分配、删除、判断free状态以及物理页与逻辑页映射关系的函数。

## Buffer Pool Manager模块

其他模块函数并不直接与Disk产生交互，都需要通过Buffer Pool（内存池）来实现具体Disk数据读取，Buffer Pool Manager根据需求将硬盘中的数据分配到内存中，供调用者使用。

其主要实现函数有FetchPage 、NewPage、UnpinPage等；在这里我们需要指出，内存的管理策略，是先从内存池中寻找数据页是否已被读入，若失败再从free\_list中寻找可以使用的页，最后再考虑使用替换策略。

另外，需要特别注意的是，我们只关注于dirty数据页的状况，因为只有dirty的数据页才会影响disk的数据存储情况；且在unpin函数中，我们需要维护一pin\_count变量；因为可能存在多个进程调用统一数据页的情况，所以当pin\_count数值不为0时，我们只是减少了其数值大小，并没有真正的释放数据页。

在本模块中，我们最先使用的替换策略是lru\_replacer。Least Recently Used算法，将最近最少使用的数据页回收。最开始的实现是使用了一个双向链表进行记录。

链表容量为内存能容纳的最大数据页数。将内存中已经不再引用的数据页号放入链表，当内存满时，可以回收一个页来开辟出新空间时，把队列最前端（最早放置）的页号对应的页回收。需要注意，当复用链表中的页时，需要将其从队列中取出，并重新放入链表的最后端（表示最近使用）。

但在具体的实现过程中我们发现，因为lru\_placer的遍历策略，会使得Pin函数大量占用插入语句的运行带宽，造成极大的时间消耗；因而在此基础之上我们提出了另一种替代方案；这一数据结构将在后续部分进行介绍。

## Record Manager模块

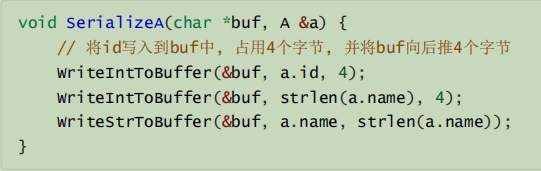
在minisql的设计中，Record Manager负责管理数据表中所有的记录（精确到行），它能够支持记录的插入、删除与查找操作，并对外提供相应的接口。

对于record记录而言，我们存有以下几个相关概念（对映于数据库中表的构成）：1. 列，用于定义和表示数据表中的某一个字段，即包含了这个字段的字段名、字段类型、是否唯一等；2. 模式，用于表示一个数据表或是一个索引的结构，其中一个模式由一个或多个列组成；3. 域，对应于一条记录中某一个字段的数据信息，如存储数据的数据类型，是否是空，存储数据的值等等；4. 行，与元组的概念等价，用于存储记录或索引键，其中一行由一个或多个域组成。

在minisql的数据表存储中，我们为了方便统一处理，都需要提供对映的函数将上面提到的行，域，列，模式四个类型转化为字节流对象，以写入数据页中。另外，为了能够从磁盘中恢复这些对象，我们同样需要能够提供一种反序列化的方法，把字节流对象转换为我们需要的对象类型。

在这里，我们对序列化与反序列化作出一定的解释：

Serialize (序列化) 就是将row、column、schema以及field等对象以字节流(char\*)的方式储存在硬盘中，使其被持续化地储存，并可以在需要时通过反序列化操作读出我们想要的数据。对映的伪代码如下（以自定义结构A为例，A包括id（int）与name（string）属性）：



Deserialize (反序列化) 反序列化就是从disk中读出所存的内容，将以(char\*)方式存在disk中的数据读出并存到row、column、schema、field等对象中。总得来说，序列化与反序列化操作是对称的。

另外此模块还涉及了Table Heap这一重要概念。一个table heap里面用双向链表的储存了大量的table page，而table page里又存储了一个又一个row（tuple/entity）。在此我们可以使用一个Rowid定位到一个唯一的Row，通过高32位获取对映的page\_id，低32位获取Row在该page中的对映位置（slot\_num），从而介导执行器的实际操作。

在这部分，我们主要实现了InsertTuple、UpdateTuple、ApplyDelete等函数，以及TableIterator迭代器，供于执行器进行简便地调用与实现。

## Executor模块：

由于创建删除查询数据库、数据表、索引等操作对应的语法树较为简单，它们均不用通过Planner生成查询计划，而是直接被声明为private类型的成员，其执行过程对上层模块隐藏，上层模块调用ExecuteEngine的execute方法并传入语法树结点即可无感知地获取到执行结果。

但是针对Insert、Delete等语法树较为复杂的函数，可以在获得planner（已给出，不用实现）生成的具体的查询计划后，生成执行计划的具体算子，步骤如下——先遍历查询计划树，将tree上的PlanNode换成对应的子Executor。

本次算子采用火山模型，即Executor会将整个 SQL 构建成一个 Operator 树，查询树自顶向下的调用接口，数据则自底向上的被拉取处理。每一种操作会抽象为一个 Operator，每个算子都有 Init() 和 Next() 两个方法。Init() 对算子进行初始化工作。Next() 则是向下层算子请求下一条数据。当 Next() 返回 false 时，则代表下层算子已经没有剩余数据，迭代结束。具体算子包括如下：

SeqScan算子：由SeqScanPlanNode指定顺序扫描的表名和谓词，然后由SeqScan顺序扫描该表，根据谓词满足条件通过Next()返回一条符合条件的Row

IndexScan算子：仅支持单列索引。当Planner检测到select的谓词中的列上存在索引，而且索引只包含该列时，会生成IndexScanPlan，其他情况则生成SeqScanPlan，此项目支持谓词中出现and的情况，也就是说我们只需在Init()求出每个索引的rowid集合取交集，然后构建需要返回的row集合，在每次Next()返回一个row即可。

Insert算子：通过子Executor即ValueExecutor提供的值，将行插入表中并更新索引。

Update算子：通过子Executor即SeqScanExecutort提供更新的值，使用GenerateUpdatedTuple方法将根据PlanNode中提供的更新属性构造出一个新的元组，从而修改指定表中的行并更新索引。

Delete算子：删除符合条件的行并删除索引，与Update算子类似。

## CatalogManager模块：

在内存中，以TableInfo和IndexInfo的形式存储表和索引，从而维护与之对应的表和索引的元信息和操作对象。对于索引来说，其元信息是在定义时被存储的，而其表信息、模式信息都是在反序列化（具体序列化和反序列化的解释见2.4模块介绍）后元信息生成的。因此为了便于维护所有表和索引的定义信息的durability（持久化性）和recoverability（可恢复性:在重启时从数据库文件中恢复），我们在元信息类TableMetadata和IndexMetadata实现序列化和反序列化操作。

为每一个表和索引分配一个单独数据页是一个较为简单的处理方式，因此我们同样需要一个数据页和数据对象CatalogMeta来记录管理元信息存储与数据页的对应关系。本节主要是在既定的序列化和反序列化方法中确定序列化所需的存储大小，包括三个类CatalogMeta、IndexMetadata、TableMetadata，需要它们对应的GetSerializedSize()的返回值。

CatalogManager类具备维护和持久化数据库中所有表和索引的信息，能够在数据库实例（DBStorageEngine）初次创建时初始化元数据，并在后续重新打开数据库实例时，从数据库文件中加载所有的表和索引信息，构建TableInfo和IndexInfo信息置于内存中。

此外，CatalogManager类部分接口，如CreateTable、GetTable、GetTables、DropTable、GetTableIndexes可以对上层模块（Executor）提供对数据表的操作方式，而CreateIndex、GetIndex、DropIndex则是可以对上层模块提供对指定索引的操作方式。

## Indexmanager模块：

为实现数据的快速定位，基于磁盘的B+树动态索引结构是一个好的选择，可以支持随机查找和高效访问有序记录。此模块负责数据表索引的实现和管理，包括：索引的创建和删除，索引键的等值查找，索引键的范围查找（返回对应的迭代器），以及插入和删除键值等操作，并对外提供相应的接口。

B+树结点中的数据是通过数据页存储的，针对数据页的存储形式，本实验实现了三种类型：BPlusTreeInternalPage、BPlusTreeLeafPage和

BPlusTreePage（是前两者的公共父类）。

BPlusTreePage包含了中间结点和叶子结点共同需要的数据，其中有flag（标记数据页的类型为internal还是leaf）、索引键长、结点中键值对的当前数量和容量、当前结点和父节点的数据页号等。

BPlusTreeInternalPage不存储实际的数据，它只按照顺序存储m个键和m+1个指针。实际实现时将第一个键设置为INVALID，也就是说，顺序查找时需要从第二个键开始查找。由于中间结点在任何时候都需要满足half-full的条件，因此在删除操作导致半满条件不符合时需要和邻结点或另一个结点进行Merge合并，相应地在插入时导致结点超出容量时也需要分裂成两个近半满结点，从而维护B+树的稳定性。

BPlusTreeLeafPage按照顺序存储m个键和m个值，其中键由一个或多个Field序列化得到，叶结点和中间结点一样遵循着键值对数量的约束，同样也需要完成对应的合并、借用和分裂操作。

此外，需要补充一些基本概念，索引键(Key)是索引列的值序列化后得到的字符串,值(Value)根据结点的类型而有所不同，其中叶节点储存row\_id信息，而内部节点存储的是page\_id。KeyManager负责对GenericKey进行序列化/反序列化比较。

B+树索引目前只支持unique\_key，在出现插入的key-value键值对出现重复时，会导致索引更新失败。B+树目前支持创建、插入、删除、查找和释放等操作，其遍历通过迭代器实现，与堆表实现类似，B+树索引迭代器可以在范围查询时将所有的叶结点组织成为一个单向链表，然后沿着特定方向有序遍历叶结点数据页中的每个键值对。

# 测试方案和测试样例

（这部分主要针对上面程序功能来设计测试案例）

# 分组与设计分工