数据库系统概论项目报告

潘子睿　　余任杰
2020010960　　2020010966

2022 年 12 月 25 日

目录

1 项目架构

SimDB是一个简单的关系型数据库系统，其能够支持一些基本的SQL语句，完成对数据库的插入、删除、更新、查找等操作。项目的最底层是一个**页式文件管理系统**，在其上构建了**记录管理**和**索引管理**两个模块，分别用来维护数据库中的一条条记录以及在某些记录上建立的索引。 **系统管理**模块通过调用记录管理和索引管理两个模块的接口，实现了具体操作数据库的各项功能。最后，在**解析处理**模块中通过Antlr4解析SQL指令，并将其交给系统管理模块执行。

本项目源代码目前位于https://github.com/pzrain/SimDB。

1.1 支持功能

1.1.1 数据结构

数据库当前支持的数据结构包括：

- 整型（INT）
- 浮点型（FLOAT）
- 字符串型（VARCHAR）

其中VARCHAR仅支持定长字符串。

1.1.2 SQL语句

TODO：具体描述支持的功能，并给出一些SQL语句作为示例。

1.1.3 其他

TODO：其他想要说明的功能

1.2 代码测试

TODO：描述对于项目的测试方法

2 编译运行

本项目基于CMake进行自动构建。需要Antlr4依赖，以及编译器支持C++17特性。使用时，将CMakeLists.txt中的ANTLR4_RUNTIME_DIRECTORY的值设置为Antlr4运行时库antlr4-runtime.h所在的目录，并将ANTLR4_RUNTIME_SHARED_LIBRARY的值设置为Antlr4运行时动态链接库libantlr4-runtime.so所在位置。
成功配置好Antlr4后，在项目根目录下执行以下命令：

```
./run.sh -c
```

即可自动进行编译并运行，生成的可执行文件SimDB位于在./bin目录下。

SimDB的交互方式选用类似于MySQL的命令行交互方式，一个示例如下：

```
Welcome to SimDB, a simple SQL engine.
Commands end with ;
mysql> USE Tsinghua;
Database changed.
Tsinghua> SELECT * FROM student;
Tsinghua> SELECT;
[Parser Error] line 1,6 mismatched input ';' expecting {'*', 'COUNT',
'AVG', 'MAX', 'MIN', 'SUM', Identifier}.
[ERROR] detect 1 error in parsing.
Tsinghua> quit;
Bye!
```

3 系统设计

3.1 页式文件系统

本部分代码位于./src/filesystem，需要注意的是，本部分直接使用了课程实验文档附录中提供的参考实现代码（对其中部分接口做了略微调整）。

数据库是被设计用来存储大量数据的系统，数据库中一个文件的大小甚至可能超过计算机的内存。因此，需要一个**页式文件管理系统**来管理数据库的各个文件，以及一个**缓存**机制，将操作的多个页面缓存在内存中，只在需要时进行替换和写回，以提高读写的效率。参考实现中使用的替换算法为**最近最少使用算法**（LRU）。

3.2 记录管理

本部分代码位于./src/record。记录管理模块是整个数据库系统中相对比较底层的模块之一。其负责管理存入数据库的一条条记录，具体而言，记录管理模块需要支持的功能包括：将某一记录存放在某一文件的特定位置处，并维护好该记录的位置信息、是否为空的标记等；根据指定的位置，从存放数据的文件中取出指定的记录；将某一指定的记录从文件中删除；修改文件中某一指定位置的记录，实际上就是先读取记录，修改后再写回到原来的位置。

同时，本模块还负责维护一张表的结构，具体而言，需要支持的功能包括：增加一张表；删除一张表；给原有的表增加/删除一个表项；修改原有的表中的一个表项等。为了实现以上的所有功能，记录管理模块会调用页式文件系统中定义的各个接口，从而完成实际的文件I/O操作。

在本模块的具体实现中，一个数据库以文件夹的形式存储，该数据库下的一张表以单个文件的形式存放在对应文件夹下。在处理某一文件中的多个内存页时，本模块将其分为两类，分别为表头页和记录页。前者用来存放一张表以及其对应文件的**元数据**，包括表的列数，每列的具体要求，以及该文件的总页数，第一个有空闲位置的页等信息。后者则用来存放对应表下的具体记录。

3.2.1 记录结构

一条记录包含多个表项的具体内容。记录实际上存在两种结构，分别为**序列化**和**反序列化**后的结果，实际存储在文件中的记录是序列化后的结果。

序列化 序列化后的结果为字节的序列，也即将记录中的各个表项拼接在一起，组成一个`char`数组。该字符数组的前两位字节被用来判断记录中各个表项是否为空值（`Null`）。

反序列化 反序列化即为将字节的序列整理成为更易操作的格式，实际过程中对记录的修改都是基于反序列化的形式。反序列化会将字节数组的各个表项提取出来，构造成一个链表。

记录位置 每条记录的位置被维护成一个**二元组**`(pageId, slotId)`，表示该记录被存放在第`pageId`页上的第`slotId`槽中。

3.2.2 表头页结构

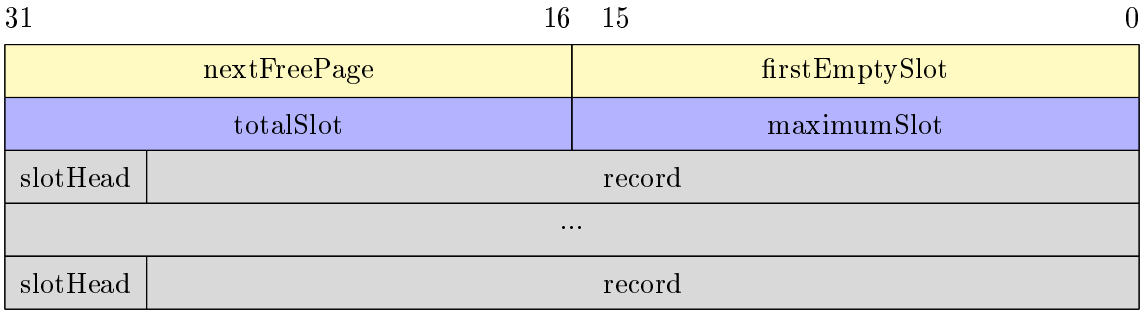
每个文件对应的第一个内存页被处理成**表头页**。表头页中存储的各个字段如下

名称	占用字节数	描述
valid	1	该页是否已经初始化
colNum	1	表中的项数
entryHead	1	第一项在entrys中对应的下标
firstNotFullPage	2	第一个非满页的页码
recordLen	2	表中定长记录的长度
totalPageNumber	2	当前总页数
recordSize	4	一页上所能存放的记录数
recordNum	4	总记录数
entrys[TAB_MAX_COL_NUM]	-	各表项的具体描述
tableName[TAB_MAX_NAME_LEN]	-	各表项的名称

其中，每个表项使用一个类`TableEntry`来描述，表中的`entrys`即为`TableEntry`的数组。`TableEntry`类的具体实现，参见附录中的各模块接口详细说明。

3.2.3 记录页结构

文件对应的各个内存页中，除了第一个为表头页外，其余均为**记录页**。记录页的结构组织如下：



记录页中的`nextFreePage`，以及表头页中的`firstNotFullPage`，将所有已分配的空闲页串成了一张链表。这样，在插入一条记录时，可以迅速找到有空闲位置的页；删除记录时，如果该页上的记录已经被删完了，就将其添加到空闲页链表的尾部。`totalSlot`记录的为当前页面内记录的总数，如果其达到了`maximumSlot`，说明页面已被填满，需要将其从空闲页链表中去除。对于非空闲页，该域中的内容可以是任意值。

如上图所示，记录页中的前8个字节用于记录和空闲页管理以及页内槽数相关的信息，剩下的空间中会**紧密**排列着各条记录。由于记录是定长的，因此一旦记录的长度确定，就可以计算出页面上最多能存放的记录总数，以及每条记录存放位置的偏移。与空闲页的管理类似，空闲槽也被组织成链表，对任一个空闲槽，其对应的下一个空闲槽的编号被记录在`slotHead`中。链表尾部空闲槽该域的值为`-1`。而对于非空闲（已经写入了记录的）的槽，其`slotHead`域会被记录为`SLOT_DIRTY`以进行区分。

3.3 索引管理

本部分代码位于`./src/index`。索引管理模块与记录管理模块类似，只是其处理的不是插入数据库的具体记录，而是针对这些记录而建立的索引。具体而言，索引管理模块需要支持的功能包括：为某一项记录建立索引；删除指定的索引；向已经建立的索引中再插入一项；搜索已经建立的索引等。建立的索引以B+树的形式被存储在内存中，因此，索引管理模块也需要调用页式文件管理系统中提供的接口。

3.2.1中提到，每条记录由一个**记录位置**唯一标识，从而可以用它作为对于记录的索引。实际操作过程中，对于数据库中某一张表下的某一列建立索引，也即为其建立一棵B+树，将该列的值作为`key`，同时将代表记录位置的二元组`hash`成单个元素作为`val`，一起存入B+树中。在查找时，指定`key`，可以在$\mathcal{O}(\log(n))$时间内找到对应的`val`，进而也就得到了记录的位置。因此，索引管理模块被用来实现对查找的加速，以及建立主键索引等功能。

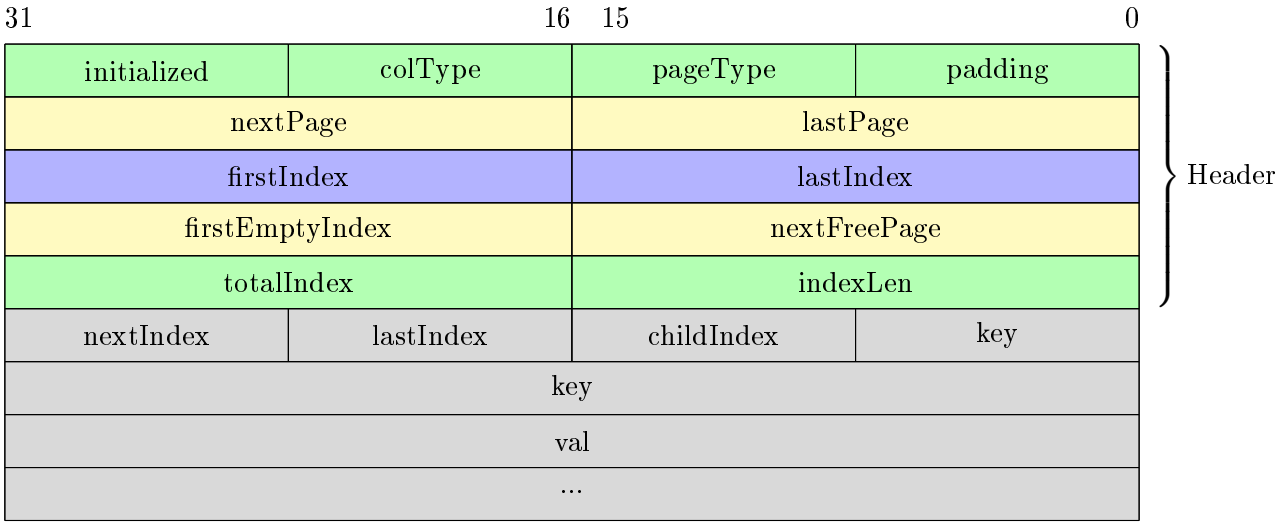
3.3.1 B+树实现

B+树是二叉平衡树的一种，在节点访问时间远远超过节点内部访问时间的时候，具有非常大的优势。B+树的节点分为两类，分别为**内部节点**和**叶子节点**。真实的索引数据被保存在叶子节点，而内部节点只保存一些结构信息。本项目中，B+树采用的结构为内部节点的关键字个数与孩子个数相等，以及关键字按照单调递增的顺序排列，每个关键字都是对应子树中的最大值。 B+树的每个内部节点维护了指向其两个兄弟、父亲和孩子节点的指针，以便于搜索。对于B+树的插入、删除和查找，以及处理上溢和下溢的方法，这里不再赘述，详细的可以参见参考文献。

3.3.2 索引页结构

索引头页 索引头页为索引文件中的第一页，记录一些必要的控制信息，包括根页的页码、第一个非满页、总页数等。

索引页 索引页的结构如下：



每个索引页的前20个字节用来存储页面的元数据，剩下的位置被用来摆放索引。每条索引除了包含必要的key以及val以外，还需要额外记录下一条、上一条以及孩子索引的位置。这些是由B+树在插入、删除索引时维护的。同样的，页面上的空闲槽位被串连成一张链表，对于空闲的槽位，其nextIndex位置存放的就是下一条空闲槽位的位置。

3.4 系统管理

本部分代码位于./src/system。系统管理模块是连接SQL解析模块与底层模块的桥梁。其负责接收SQL解析模块的请求，调用及维护两个底层模块记录与索引的接口。系统管理的功能主要分为两部分，分别为对数据库的管理以及查询解析。前者对应SQL中的DDL（Data Definition Language），负责维护数据库的结构，数据表的建立、删除、修改，添加约束等等。后者对应SQL中的DML（Data Manipulation Language），负责处理记录的增、删、查、改。由于这两个功能相对比较独立，在其他的实现中，可能被分在两个不同的模块中。但是在本项目中，为了方便SQL解析模块的调用，以及使结构更加清晰，将二者统一在系统管理模块中。

3.4.1 数据库管理

数据库管理包括对于数据库（database）以及数据表（table）的维护。

数据库 对每个数据库，建立一个文件夹，其下存放**数据库元文件**、所有**记录文件**以及所有**索引文件**。对数据库的操作包括建立数据库、删除数据库、切换数据库、显示数据库下所有的数据表等。在操作某张具体的表时，首先必须切换（选择）为某个数据库。

数据表 每个数据库下可以建立多张数据表。对数据表的操作包括建表、删除表、修改表项、显示所有表项的基本信息，建立约束等等。此外，还包括为表中的某一列或某几列建立索引。在建立约束或删除约束的时候，根据具体情况也会自动的建立或删除相关索引。

3.4.2 查询解析

处理查询解析的过程分为三步：

1. 首先进行语法以及格式等方面的检查。
2. 遍历Where表达式（如果有的话），将所有符合条件的记录提取出来。
3. 根据其他限制条件（例如，GROUP BY，或者选择的域信息）对所有记录做一次过滤，得到最终结果。

需要注意的是，在插入、删除、修改记录时，均需要对约束信息进行检查。具体而言，插入记录时，需要检查主键约束、Unique约束，以及相对应的外键是否存在；删除记录时，需要检查是否存在其他表的外键关联到本条记录；修改记录可以视为先删除再插入对应记录，需要小心地、按某种顺序检查上两点中提到的约束。另外，如果操作的列上建有索引，还需要对索引进行同步的操作，以保证正确性。

如果在执行插入、删除或者修改多条记录的过程中遇到了错误，考虑到SQL事务的**原子性**，会将已经成功进行的所有操作复原。例如，如果在插入第三条记录的时候因为主键约束导致插入失败，会首先将前两条已经插入的记录从数据库中删除，然后终止本次插入操作并报相关错误。

3.5 解析处理

本部分代码位于./src/parser。解析处理模块是数据库系统中最顶层的模块，其直接接受用户的SQL语句输入，根据需要进行一定程度上的查询优化，并将解析的结果转交给系统管理模块。系统管理模块实际操作数据库，再将执行的结果返回给解析处理模块。

3.5.1 SQL语句解析

本项目采用开源的Antlr4这一语言识别工具来构建SQL语法分析器。具体支持的文法在./src/parser/antlr4下的文件SQL.g4中定义。 Antlr4会对收到的SQL语句进行解析，并生成**抽象语法分析树**。使用Antlr4的**访问者模式**，我们就可以方便地遍历语法分析树，并

在遍历的同时调用系统管理模块中的接口，执行相应的操作。

最终的实现中，使用自定义的`MyANTLRErrorListener`继承了`AnTLRErrorListener`，并在此基础上重写了处理`syntaxError`的函数。这样，程序在解析的过程中如果遇到语法错误，就会报告并退出此次解析，不会再进入后续对抽象语法树的遍历以及实际操作数据库的过程。

3.5.2 查询优化

由于数据库在针对**多表联合查询**时的性能较差，因此优化主要是针对这一部分。具体而言，如果多表连接查询时的`WHERE`子句形如E_1 AND E_2 AND $\cdots$ AND E_n，其中E_i为形如$T_i.C_k = T_j.C_l$的任意表达式，T_i、T_j为某两张表，C_k、C_l为某两列，那么如果在C_k或C_l上建有索引，就可以对该查询进行加速。具体而言，例如$T_1.C_1 = T_2.C_2$，而表T_1的列C_1上建有索引，那么将该查询调整为$T_2.C_2 = T_1.C_1$，先遍历查询表T_2，然后再利用索引查询表T_1，就能起到加速效果。

在实际处理查询优化时，会对每一个多表查询的`WHERE`子句建立一个图，每个$T.C$用一个节点来代表，在两个点$T_1.C_1$和$T_2.C_2$间连边当且仅当表达式$T_1.C_1 = T_2.C_2$或$T_2.C_2 = T_1.C_1$存在。如果在加入某一条边时，发现两个顶点已经位于原图的某个连通分量中，那么就说明这条边所对应的表达式是冗余的，直接将其去掉即可。

对于未建立索引的节点，可以找到图中的一条**最长路径**，使得这一路径除了起点外，其余节点均已经建立索引。否则，说明与其相邻的所有节点均未建立索引。查询时按照路径上从起点到终点的顺序，即可实现索引加速。最终可将原图上的所有边完全划分为多条不相交路径，每条路径只可能为以下三种形式之一：

a. 除了起点外，其余所有节点均建立了索引。

b. 所有节点均建立了索引。

c. 只包含一条边，涉及的两个节点均未建立索引。

除了迫不得已而产生的c类型的路径，其余均可利用索引进行加速。完成优化后，语法分析树受到了相应调整，后面的遍历解析就在此基础上继续进行。

4 项目分工

- 记录管理、索引管理：潘子睿
- 系统管理、解析处理：余任杰

5 参考文献

以下列出了本项目完成过程中所参考的部分资料，包括课程的实验文档、往届的实现，以及相关网站等。

i. 往届实现的数据库项目

ii. CS346 Redbase project

iii. B+树的定义与实现

iv. 使用Antlr4完成SQL的语法解析

v. 课程实验文档