数据库系统概论项目报告

潘子睿　　余任杰
2020010960　　2020010966

2023 年 1 月 7 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

本节将描述数据库支持的SQL语句，并给出使用样例。其中用[ ]包围的是可以在实际使用中依据需要更改的内容，用* *包围的是可选项。

- **创建数据库**:根据数据库名创建对应的数据库。

  ```
  CREATE DATABASE [database_name];
  ```

- **删除数据库**:根据数据库名删除对应的数据库。

  ```
  DROP DATABASE [database_name];
  ```

- **列出所有数据库**:列举出当前存在的所有数据库。

  ```
  SHOW DATABASES;
  ```

- **切换数据库**:根据数据库名切换到对应的数据库。

  ```
  USE [database_name];
  ```

- **列出所有表**:列举出当前数据库中的所有数据表，需要先切换到某个数据库中。

  ```
  SHOW TABLES;
  ```

- **列出所有索引**:列举出当前数据库中的所有索引，需要先切换到某个数据库中。

```
SHOW INDEXES;
```

- **创建数据表**:根据表名，各字段（field）信息创建数据表，需要先切换到某个数据库中。

```
CREATE TABLE [table_name](
    -- normal field
    [column_name] [type] *NOT NULL* *DEFAULT [default_value]*,
    -- primary key field
    PRIMARY KEY ([column_name]),
    -- foreign key field
    FOREIGN KEY ([column_name]) REFERENCES [ref_table_name]
    ↪  ([ref_column_name])
);
```

共支持三种字段创建语句，第一种是`normal field`，可以根据列名、类型、`NOT NULL`约束（可选）、默认值（可选）来声明一个字段；第二种是`primary key field`，根据指定的列名在对应的列上创建主键；第三种是`foreign key field`，根据指定的列名，以及依赖的表中的列名创建外键。

- **删除数据表**:根据表名删除对应的数据表，需要先切换到某个数据库中。

```
DROP TABLE [table_name];
```

- **列出表的模式信息**:根据表名列出对应表中每个字段，即每一列的信息，需要先切换到某个数据库中。

```
DESC [table_name];
```

- **插入新记录**:提供需要被插入的值，根据表名将一条记录插入对应的表中，需要先切换到某个数据库中。系统仅支持一次性插入一条完整的记录，即包括每一列需要插入的值的一条记录，不支持通过列名来插入对应的列。

```
INSERT INTO [table_name]
VALUES ([value1], [value2], ...);
```

- **删除记录**:根据表名和限定条件删除表中特定的记录，需要先切换到某个数据库中。

```
DELETE FROM [table_name]
WHERE [condition];
```

依据`condition`描述的条件来删除表中的记录，关于`WHERE`的描述见下文。

- **修改记录**:根据表名和限定条件将表中特定记录的某些列的值修改为指定的值，需要先切换到某个数据库中。

```
UPDATE [table_name]
SET [column1] = [value1], [column2] = [value2], ...
WHERE condition;
```

依据condition描述的条件，在符合条件的记录中，根据列名和对应的值修改记录。

- **查询数据**:根据表名、限定条件等选取表中特定的记录并展示，需要先切换到某个数据库中。

```
SELECT [selectors]
FROM [table_name]
*WHERE condition*
*GROUP BY [column_name]*
*LIMIT [integer] *OFFSET [integer]**
```

通过selectors指定选择的对象，支持聚合查询，如以下几种查询均为合法操作：

```
SELECT [column_name] FROM [table_name];
SELECT * FROM [table_name];
SELECT COUNT(*) FROM [table_name];
SELECT SUM([column_name]) FROM [table_name];
SELECT AVG([column_name]) FROM [table_name];
SELECT MAX([column_name]) FROM [table_name];
SELECT MIN([column_name]) FROM [table_name];
```

允许通过condition来限定查询的范围，允许通过GROUP BY来对结果集进行分组，允许使用分页查询。

- **创建索引**:根据表名和列名为数据表添加索引，需要先切换到某个数据库中。

```
ALTER TABLE [table_name] ADD INDEX ([column_name]);
```

- **删除索引**:根据表名和列名删除已经存在的索引，需要先切换到某个数据库中。

```
ALTER TABLE [table_name] DROP INDEX ([column_name]);
```

- **创建主键**:根据表名和列名为数据表添加主键，需要先切换到某个数据库中。

```
ALTER TABLE [table_name] ADD CONSTRAINT PRIMARY KEY ([column_name]);
```

- **删除主键**:根据表名删除已经存在的主键，需要先切换到某个数据库中。

```
ALTER TABLE [table_name] DROP PRIMARY KEY;
```

- **创建外键**:根据表名和列名，以及外键依赖的表名和列名创建外键，需要先切换到某个数据库中。

```
ALTER TABLE [table_name] ADD CONSTRAINT FOREIGN KEY ([column_name])
↪ REFERENCES [ref_table_name] ([ref_column_name]);
```

- **删除外键**:根据表名和列名删除已经存在的外键，需要先切换到某个数据库中。

```
ALTER TABLE [table_name] DROP FOREIGN KEY [column_name];
```

- **创建UNIQUE约束**:根据表名和列名为数据表添加UNIQUE约束，需要先切换到某个数据库中。

```
ALTER TABLE [table_name] ADD UNIQUE ([column_name]);
```

- **删除UNIQUE约束**:根据表名和列名为删除已存在的UNIQUE约束，需要先切换到某个数据库中。

```
ALTER TABLE [table_name] DROP UNIQUE ([column_name]);
```

- **WHERE子句**:用于提取满足条件（condition）的记录，条件之间支持逻辑与（AND）运算。 conditon具体包括以下几种形式：

  1. 比较运算符，大于(>)、小于(<)、等于(=)、大于等于(>=)、小于等于(<=)、不等于(<>)：

     ```
     SELECT * FROM [table_name] WHERE [column_name1] > [value1] AND
     ↪ [column_name2] <> [value2];
     ```

  2. 空值判断：

     ```
     SELECT * FROM [table_name] WHERE [column_name] IS *NOT* NULL;
     ```

  3. 取值列表，要求查询的内容包含在取值列表里：

     ```
     SELECT * FROM [table_name] WHERE [column_name] IN ([value1],
     ↪ [value2], ...);
     ```

     该语句表示查询结果应该是column_name这一列中值等于value1或value2或...的值。

  4. 嵌套子查询：

     ```
     SELECT * FROM [table_name] WHERE [column_name] IN (SELECT
     ↪ [column_name] FROM [table_name] WHERE [condition]);
     ```

     该语句可以理解为先做后半部分括号中的查询，然后在查询结果中选取满足条件的结果。

  5. 模糊查询：

     ```
     SELECT * FROM [table_name] WHERE [column_name] LIKE '%a';
     SELECT * FROM [table_name] WHERE [column_name] LIKE 'a%';
     ```

     对于VARCHAR类型的列，可以使用模糊查询，上述两个语句分别表示查询column_name这一列中以a结尾的结果和以a开头的结果。

1.1.3 其他

- **SQL事务的原子性**：在对数据进行增加（Insert）、删除（Delete）以及修改（Update）的**事务**时，其中的某个操作可能是**“非法的”**。例如，试图插入10条数据，但是中间某一个数据不符合数据表定义的格式或是不满足约束限制等。此时，数据库会维护一个事务的**原子性**，也即事务中的所有操作要么都发生，要么都不发生。因此，在遇到上述的非法情况时，前面已经完成的操作会被视为无效，数据库会回滚到事务开始前的状态。
- **查询优化**：针对SQL多表查询，结合索引对查询时各个条件的顺序进行了调整，以达到加速的效果。具体详见**??**。

1.2 代码测试

在./testcases目录下给出了一系列用于测试项目的.sql文件，包括测试建表、测试增删查改基本功能、测试约束建立与检查以及测试大量数据下的索引性能等。

测试时，在./testcases目录下执行python check.py即可自动运行全部测例，并将结果写入到./testcases/out目录下。同时，如果./testcases/ans目录下有对应测例的结果文件.ans，会将.out与其进行比对，输出**通过**（Pass）或是**不通过**（**Fail**）。另外，可通过python check.py --test <testFileName>来指定执行某一个测例。

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之一：

```
./run.sh -c
./run.sh -batch <input_file> <output_file> -c
./run.sh -onlyc
```

即可自动进行编译并运行，生成的可执行文件SimDB位于./bin目录下。三条命令分别对应启动命令行模式、批处理模式以及只重新编译不运行。

SimDB的交互方式可选用类似于MySQL的命令行交互方式或者批处理模式。命令行交互的一个示例如下：

```
Welcome to SimDB, a simple SQL engine.
Commands end with ;
SimDB> 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!
```

在批处理模式中，可从指定的输入文件中读入SQL命令并依次解析执行，最后将结果输出到指定的文件中。对项目的批量测试就基于批处理模式完成。

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`uint8_t`数组。该字节序列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

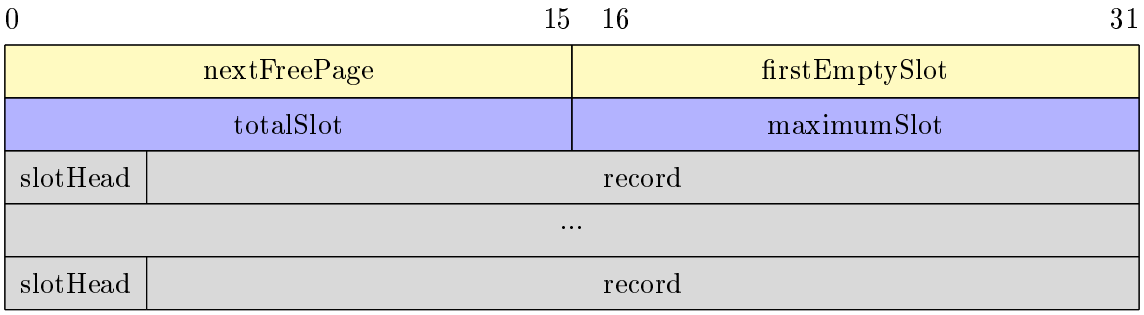
每个文件对应的第一个内存页被处理成**表头页**。表头页中存储的各个字段如下

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，请参见源代码中的`./src/record/RMComponent.h`。

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

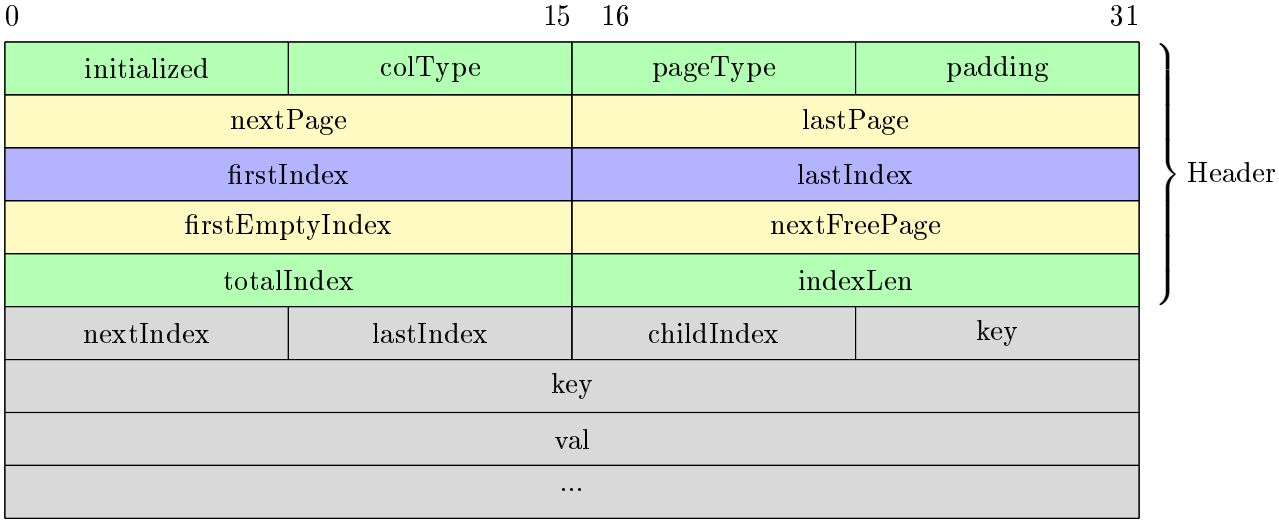
B+树是二叉平衡树的一种，在节点访问时间远远超过节点内部访问时间的时候，具有非常大的优势。B+树的节点分为两类，分别为**内部节点**和**叶子节点**。真实的索引数据被保存在叶子节点，而内部节点只保存一些结构信息。本项目中，B+树采用的结构为内部节点

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。

此外，如果直接对双表进行笛卡尔积然后查询解析，会导致占用的存储空间过高，且花费的时间非常长。在经过上述的优化调整后，通常而言（只要WHERE表达式中某一列上建有索引）都可以通过索引加速。具体地，不需要将双表笛卡尔积的结果全部取出筛选，可以使用WHERE表达式中的首项来做一次过滤，而由于索引的存在，这次过滤是很高效的。这样，最后进入WHERE筛选的就是已经通过了一轮过滤的数据，其数量一般会明显减少。

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

A 附录：主要接口说明

A.1 记录管理

记录管理中主要的对外接口定义在FileHandler与RecordManager两个类中，前者用来处理某个具体的数据表，后者用来管理一个数据库下的所有数据表。接口定义与部分接口说明如下：

```
class FileHandler{
public:
    FileHandler();
    ~FileHandler();
    void init(BufPageManager* bufPageManager_, int fileId_, const char*
    ↪  tableName_);
    int getFileId();
    char* getTableName();
    /**
     * @brief When operating type is TB_INIT, tableEntry represents an
     ↪  array of TableEntry
     *        TB_REMOVE, TB_EXIST requires parameter colName
     *        TB_INIT, TB_ALTER, TB_ADD requires parameter tableEntry
     *        TB_INIT requires parameter num, which is the total number of
     ↪  tableEntry array
     * @return -1 if fail,
     *         0 if succeed for all op other than TB_EXIST (TB_EXIST
     ↪  returns 1 if exists otherwise 0)
     */
    int operateTable(TB_OP_TYPE opCode, char* colName = nullptr,
    ↪  TableEntry* tableEntry = nullptr, int num = 0);
    bool getRecord(RecordId recordId, Record &record);
    /**
     * @brief the page id and slot id of the inserted record will be
     ↪  stored in recordId
     */
    bool insertRecord(RecordId &recordId, Record &record);
    bool removeRecord(RecordId &recordId, Record &record);
    /**
     * @brief parameter recordId specifies the position of the record that
     ↪  needed to be updated
     *        parameter record will substitutes the old record
     */
```

```
27      bool updateRecord(RecordId &recordId, Record &record);
28      /**
29       * @brief ATTENTION: you should manually delete the pointer to avoid
         ↪  memory leak
30       * @return all records stores in this file
31       */
32      void getAllRecords(std::vector<Record*>&, std::vector<RecordId*>&);
33      /**
34       * @brief get specific fields of all records storing in this file
35       *        if you want get i-th field, please set the i-th bit of
         ↪  enable (from low to high) to 1
36       * @return true if successful
37       */
38      bool getAllRecordsAccordingToFields(std::vector<Record*>&,
         ↪  std::vector<RecordId*>&, const uint16_t enable = 0);
39      /**
40       * @brief insert records in bulk
41       */
42      bool insertAllRecords(const std::vector<Record*>&,
         ↪  std::vector<RecordId>&);
43      int getRecordNum();
44      size_t getRecordLen();
45      TableEntryDesc getTableEntryDesc();
46      TableHeader* getTableHeader();
47      void renameTable(const char* newName);
48      void saveTableHeader();
49      /**
50       * @brief transform pageId together with slotId to a int value
51       *        used in indexing
52       */
53      void transform(int& val, int pageId, int slotId);
54      void transform(std::vector<int>& vals, std::vector<int> pageIds,
         ↪  std::vector<int> slotIds);
55      void transformR(int val, int& pageId, int& slotId);
56      void transformR(std::vector<int> vals, std::vector<int>& pageIds,
         ↪  std::vector<int>& slotIds);
57  };
```

```
1   class RecordManager{
2   public:
```

```
    RecordManager(BufPageManager*, const char* databaseName_);
    ~RecordManager();
    /**
     * @brief called when create a table
     *        create the file corresponding to a table in a database
     * @return -1 if fail
     */
    int createFile(const char* tableName);
    /**
     * @brief called when remove a table
     *        remove the file corresponding to a table in a database
     */
    int removeFile(const char* tableName);
    /**
     * @brief open a file according to the tableName
     */
    FileHandler* openFile(const char* tableName);
    int closeFile(FileHandler* fileHandler);
    FileHandler* findTable(const char* tableName);
    void renameTable(const char* oldName, const char* newName);
    bool isValid();
};
```

A.2 索引管理

索引管理中主要的对外接口定义在类IndexManager中，用来管理一个数据库下所有的索引文件。接口定义与部分接口说明如下：

```
class IndexManager{
public:
    IndexManager(BufPageManager* bufPageManager_, const char*
    ↪ databaseName_);
    ~IndexManager();
    /**
     * @brief init index when open a database
     */
    int initIndex(std::vector<std::string> tableNames,
    ↪ std::vector<std::vector<std::string>> colNames,
    ↪ std::vector<std::vector<uint16_t>> indexLens,
    ↪ std::vector<std::vector<uint8_t>> colTypes);
    /**
```

```
10          * @brief  rename the tableNamess
11          */
12         void renameIndex(const char* oldTableName, const char* newTableName);
13         /**
14          * @brief build index
15          * @return return 0 if succeed, otherwise -1
16          *         so are the returning values of other functions in
            ↪  IndexManager
17          */
18         int createIndex(const char* tableName, const char* indexName, uint16_t
            ↪  indexLen, uint8_t colType);
19         /**
20          * @brief drop index, actually drop the corresponding B+ tree
21          */
22         int removeIndex(const char* tableName, const char* indexName);
23         /**
24          * @brief return true if indexName has been created
25          */
26         bool hasIndex(const char* tableName, const char* indexName);
27         int insert(const char* tableName, const char* indexName, void* data,
            ↪  const int val);
28         /**
29          * @brief insert index in batch mode
30          *        the sequence of data is not necessarily ordered
31          */
32         int insert(const char* tableName, const char* indexName,
            ↪  std::vector<void*> data, std::vector<int> val);
33         /**
34          * @brief search those index whose key is equal to data
35          *        than store their val in vector res
36          */
37         int search(const char* tableName, const char* indexName, void* data,
            ↪  std::vector<int> &res);
38         /**
39          * @brief search those index whose dat falls in [lData, rData]
40          *        set lData or rData to nullptr to get searching range [lData,
            ↪  infinity] or [-infinity, rData]
41          */
```

```
42      int searchBetween(const char* tableName, const char* indexName, void*
        ↪  lData, void* rData, std::vector<int> &res, bool lIn = true, bool
        ↪  rIn = true);
43      /**
44       * @brief remove those index whose key is equal to data and val equal
         ↪  to val (if val is not set to -1)
45       */
46      int remove(const char* tableName, const char* indexName, void* data,
        ↪  int val = -1);
47      /**
48       * @brief update those index whose key is equal to data and val equal
         ↪  to oldVal
49       */
50      int update(const char* tableName, const char* indexName, void* data,
        ↪  int oldVal, int newVal);
51      /**
52       * @brief Display all the index
53       */
54      int showIndex();
55      int showIndex(const char* tableName);
56      void transform(const char* tableName, const char* indexName, int& val,
        ↪  int pageId, int slotId);
57      void transform(const char* tableName, const char* indexName,
        ↪  std::vector<int>& vals, std::vector<int> pageIds, std::vector<int>
        ↪  slotIds);
58      void transformR(const char* tableName, const char* indexName, int val,
        ↪  int& pageId, int& slotId);
59      void transformR(const char* tableName, const char* indexName,
        ↪  std::vector<int> vals, std::vector<int>& pageIds,
        ↪  std::vector<int>& slotIds);
60      bool isValid();
61  };
```

A.3 系统管理

系统管理中主要的对外接口定义在类DatabaseManager中，用来对所有的数据库、以及各数据库下的数据表进行管理，给解析处理模块提供统一的调用接口。接口定义与部分接口说明如下：

```
1   class DatabaseManager {
2   public:
```

```
3       DatabaseManager();
4       ~DatabaseManager();
5       /**
6        * @brief -
7        * CREATE DATABASE <databse name>;
8       */
9       int createDatabase(string name);
10      /**
11       * @brief -
12       * DROP DATABASE <database name>;
13      */
14      int dropDatabase(string name);
15      /**
16       * @brief save all the tables of the last opened table if there is
         ↪  one, then open the new table.
17       * USE DATABASE <database name>;
18      */
19      int switchDatabase(string name);
20      /**
21       * @brief show all the databases created
22       * SHOW DATABASES
23       * @return 0 if succeed, -1 if encounter error
24       */
25      int showDatabases();
26      string getDatabaseName();
27      /**
28       * @brief list all the tables' name of the currently opened database.
29       * SHOW DATABASE <database name>;
30       * SHOW TABLES
31      */
32      int listTablesOfDatabase();
33      /**
34       * @brief set the property of all the columns of the table.
35       * CREATE TABLE <table name>(
36       *     <colname> coltype,
37       *     ...
38       * );
39      */
40      int createTable(string name, vector<FieldItem> normalFieldList);
41      /**
```

```
42          * @brief only print all columns' name and data type.
43          * DESC TABLE <table name>
44          */
45         int listTableInfo(string name);
46         /**
47          * @brief also modify meta data.
48          * DROP TABLE <table name>;
49          */
50         int dropTable(string name);
51         /**
52          * @brief see the function in TableManager.cpp
53          * ALTER TABLE <old table name> RENAME TO <new table name>;
54          */
55         int renameTable(string oldName, string newName);
56         /**
57          * @brief Create a index on a specific table.column
58          * 'ALTER' 'TABLE' Identifier 'ADD' 'INDEX' '(' identifiers ')'
59          */
60         int createIndex(string tableName, string colName);
61         /**
62          * @brief Drop the index on table.column
63          * 'ALTER' 'TABLE' Identifier 'DROP' 'INDEX' '(' identifiers ')'
64          */
65         int dropIndex(string tableName, string colName);
66         int hasIndex(string tableName, string colName);
67         /**
68          * @brief Display all the index of a table
69          * 'SHOW' 'INDEXES'
70          */
71         int showIndex();
72         /**
73          * @brief add primary key constraint in the table entry and modify
            ↪  meta data
74          * ALTER TABLE <table name> ADD PRIMARY KEY (<column name1>, <column
            ↪  name2>, ...);
75          * @param colNum the quantity of column need to be added primary key
76          *
77          */
78         int createPrimaryKey(string tableName, vector<string> colNames, int
            ↪  colNum);
```

```
79      int dropPrimaryKey(string tableName);
80      /**
81       * @brief Create a Unique Key object. Refer to createPrimaryKey for
         ↪  more information
82       *
83       * @param tableName
84       * @param colNames
85       * @param colNum
86       * @return int
87       */
88      int createUniqueKey(string tableName, vector<string> colNames, int
         ↪  colNum);
89      int dropUniqueKey(string tableName, vector<string> colNames, int
         ↪  colNum);
90      /**
91       * @brief add foreign key constraint in the table entry and modify
         ↪  meta data
92       * ALTER TABLE <table name> ADD CONSTRAINT <foreign key name>
93       * FOREIGN KEY(<column name>) REFERENCES <reference table
         ↪  name>(<reference column>);
94      */
95      int createForeignKey(string tableName, string foreignKeyName, string
         ↪  colName, string refTableName, string refTableCol);
96      /**
97       * @brief use foreign key's name to find the entry and modify it.
98      */
99      int dropForeignKey(string tableName, string foreignKeyName);
100     /**
101      * @brief select statement, including fuzzy, nesty, group by……
         ↪  selection
102      * 'SELECT' selectors 'FROM' identifiers ('WHERE' where_and_clause)?
         ↪  ('GROUP' 'BY' column)? ('LIMIT' Integer ('OFFSET' Integer)?)?
103      * @param dbSelect
104      * @return number of queries affected, -1 if encounters error
105      */
106     int selectRecords(DBSelect* dbSelect);
107     /**
108      * @brief update statement
109      * 'UPDATE' Identifier 'SET' set_clause 'WHERE' where_and_clause
110      * @param dbUpdate
```

```
111         * @return int
112         */
113        int updateRecords(string tableName, DBUpdate* dbUpdate);
114        /**
115         * @brief insert statement
116         * 'INSERT' 'INTO' Identifier 'VALUES' value_lists
117         * @param dbInsert
118         * @return int
119         */
120        int insertRecords(string tableName, DBInsert* dbInsert);
121        /**
122         * @brief delete statement
123         * 'DELETE' 'FROM' Identifier 'WHERE' where_and_clause
124         * @param dbDelete
125         * @return int
126         */
127        int dropRecords(string tableName, DBDelete* dbDelete);
128    };
```

A.4 解析处理

解析处理中的主要对外接口即为MyParser类中的parse()，用于解析输入的SQL语句，其定义与说明如下。Antlr4中Visitor模式定义的各个接口详见./src/parser/MySQLVisitor.h。

```
1   class MyParser {
2   public:
3       MyParser(DatabaseManager* databaseManager_):
        ↪  databaseManager(databaseManager_);
4       ~MyParser();
5       /**
6        * @brief parse the input SQL string
7        */
8       bool parse(std::string SQL);
9   };
```