# 数据库系统概论 项目报告

周聿浩

2016011347

2019 年 1 月 5 日

## 目录

# 1 项目简介

TrivialDB 是一个简单的数据库管理系统，我们实现了大部分常见的 SQL 语句和类型。同时支持多表连接、复杂表达式运算、多主键约束、外键约束、CHECK 约束、UNIQUE 和 DEFAULT 约束、聚集查询、利用 B+ 树索引的查询优化，同时，我们支持任意长度的 VARCHAR 类型。

整个数据库基于一个自行实现的页式文件系统[1]，同时支持多种不同的页结构，大致分为 B+ 树索引叶节点、B+ 树索引内部节点、B+ 树叶节点和 B+ 树内部节点。在此之上，构建有 B+ 树以及表和索引结构。

最后通过 FLex/Bison 解析用户输入的命令，传递给数据库管理系统进行执行。

另外，本项目于 `https://github.com/miskcoo/TrivialDB` 开源。

## 1.1 编译及运行

本项目需要依赖 FLex/Bison，并且需要编译器支持 C++11 特性。使用 CMake 进行自动构建。

## 1.2 代码测试

我们采用黑盒测试来验证代码正确性。在 testcase 目录下有编写好的 SQL 语句测例，testcase/ans 下有针对各个测例中 SELECT 语句的期望输出。运行 testcase/run_test.py 可以执行各个测例的 SQL 语句并且将输出与期望输出做对比从而验证代码正确性。

# 2 系统功能

## 2.1 数据类型

数据库支持的基本类型有：

- 整型（INT）
- 浮点型（FLOAT）
- 字符串型（VARCHAR），可以支持**超过一个页大小**的长度。
- 日期型（DATE），日期格式为 “`YYYY-MM-dd`”。

日期类型的字面值和字符串相同，在实现中如果必要可以转换为字符串。

## 2.2 SQL 语句

我们支持的 SQL 语句基本和要求类似，一共有如下几种

- 插入语句：`INSERT INTO …VALUES …`
- 删除语句：`DELETE FROM …WHERE …`
- 查询语句：`SELECT …FROM …WHERE …`
- 更新语句：`UPDATE …SET …WHERE …`
- 创建数据库：`CREATE DATABASE …`

[1]没有使用提供的页式文件系统。

- 删除数据库：`DROP DATABASE …`
- 切换数据库：`USE …`
- 显示数据库信息：`SHOW DATABASE …`
- 创建表：`CREATE TABLE …`
- 删除表：`DROP TABLE …`
- 显示表信息：`SHOW TABLE …`
- 创建索引：`CREATE INDEX …`
- 删除索引：`DROP INDEX …`

## 2.3 复杂表达式处理

表达式大致可以分为两种：算术表达式和条件表达式。由于采用 Bison 进行解析，可以支持任意深度嵌套的复杂表达式。我们所支持的基本运算主要如下

- 四则运算，针对整数和浮点数进行。
- 比较运算符，即 `<=`，`<`，`=`，`>`，`>=`，`<>`。
- 模糊匹配运算符，即 LIKE，其实现采用 C++11 的正则表达式库。
- 范围匹配运算符，即 IN，可以在表的 CHECK 约束中以及 WHERE 子句中使用。
- 空值判定运算符，即 IS NULL 和 IS NOT NULL 两种。
- 逻辑运算，包含 NOT、AND 和 OR 三种。

以下是一些复杂表达式运算的例子

```
UPDATE customer SET age = age + 1 WHERE age < 18 AND gender = 'F';
SELECT * FROM customer WHERE name LIKE 'John %son';
SELECT * FROM students WHERE grades IN ('A', 'B', 'C');
SELECT * FROM students WHERE name IS NOT NULL;
```

## 2.4 聚集查询

我们实现了五种聚集查询函数 COUNT、SUM、AVG、MIN 和 MAX。其中 COUNT 不支持 DISTINCT 关键字。例如

```
SELECT COUNT(*) FROM customer WHERE age > 18;
SELECT AVG(age) FROM customer WHERE age <= 18;
```

## 2.5 属性完整性约束

我们支持多种属性完整性约束，分别是

- 主键约束。一个表可以有多个列联合起来作为主键，只有在所有主键都相同时才认为两条记录有冲突，即这种情况下主键是一个元组。
- 外键约束，每个域都可以有外键约束，引用另外一个表的主键。

- UNIQUE 约束，该约束限制某一列的值不能重复。
- NOT NULL 约束，该约束限制某一列不能有空值。
- DEFAULT 约束，该约束可以在 INSERT 语句不指定值是给某列赋予一个默认值。
- CHECK 约束，该约束可以对表中元素的值添加条件表达式的检查。

下面是一个简单的例子，注意如果在多个列都指定了 PRIMARY KEY，那么就认为主键是一个元组，而不是有多个主键。例如 Infos 表的主键为 (PersonID, InfoID)。

```
CREATE TABLE Persons (
    PersonID int PRIMARY KEY NOT NULL,
    Name varchar(20),
    Age int DEFAULT 1,
    Gender varchar(1),
    CHECK (Age >= 1 AND Age <= 100),
    CHECK (Gender IN ('F', 'M'))
);

CREATE TABLE Infos (
    PersonID int PRIMARY KEY,
    InfoID int PRIMARY KEY,
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

## 2.6 多表连接查询

在 SELECT 语句中，我们支持任意多表的连接操作，例如

```
SELECT * FROM A, B, C WHERE A.ID = B.ID AND C.Name = A.Name
```

并且，对于多个表的连接中形如 A.Col1 = B.Col2 的条件，那么如果这两个列的某一个拥有索引，会利用索引进行查询优化。例如如下查询就可以优化

```
SELECT * FROM Persons, Infos WHERE Persons.PersonID = Infos.PersonID;
SELECT * FROM Persons, Infos, Datas WHERE Persons.PersonID = Infos.PersonID AND
↪  Datas.N IS NOT NULL;
SELECT * FROM Persons, Infos, Datas WHERE Persons.PersonID = Infos.PersonID AND
↪  Datas.ID = Infos.PersonID;
```

具体的优化方法以及何种查询可以优化见实现细节中“查询优化”。

## 2.7 表别名

我们在多表连接查询时支持通过别名（alias）的方式对一个表进行连接，例如

```
SELECT * FROM Persons AS P1, Persons AS P2 WHERE P1.PersonID = P2.PersonID;
```

# 3 底层数据结构

## 3.1 页式文件系统

页式文件系统是最基本的结构，依赖于操作系统的文件管理。一个页的大小为 4KB，一个文件的读写均按照页为单位。为了读写的效率，我们采用缓存机制，每次访问页后会缓存在内存中，并且写操作不会马上写入文件。缓存的替换方法为当其填满后会将最少访问的一个写回文件。

在我们的实现中，页式文件系统主要在 fs 目录内。支持同时打开多个文件。

## 3.2 页结构

一个页可以被解释为多种不同的结构，其基本结构有三种：overflow、fixed、variant。每个页的前 2 个字节用于标识该页为何种类型。

**overflow 页** 该页结构较为简单，用于存储过长的数据。其头部包含三个元素：页标识符（2 字节）、页内容大小（2 字节）和下一页的页号（4 字节）。其于部分均用来存储溢出的数据。完整的溢出数据是用 overflow 页组成的链表存储。例如 VARCHAR 类型过长导致一条记录超过可以接受的长度，那么就会将多余的部分存储在此页。

**fixed 页** 该页结构用于记录 B+ 树的内部节点，其页头按顺序分别为

| 属性名称 | 属性解释 | 属性长度 |
|---|---|---|
| magic | 页标识符 | 2 |
| field_size | 关键字的长度 | 2 |
| size | 页中存储的项数 | 4 |
| next_page | B+ 树中该页的右邻居 | 4 |
| prev_page | B+ 树中该页的左邻居 | 4 |

其余部分为页的数据部分，数据分为两个部分：key 和 child。分别表示 B+ 树节点的关键字和儿子指针。key 部分按照大小排序紧跟着页头后连续存储。child 部分是在页的尾部开始往前连续存储。

**variant 页** 该页结构用于索引 B+ 树的内部节点，以及记录 B+ 树的页节点等需要存储较长数据的页。其插入和删除项不会对数据进行移动，而只修改指向数据的指针。其页头按顺序分别为

| 属性名称 | 属性解释 | 属性长度 |
|---|---|---|
| magic | 页标识符 | 2 |
| flags | 页属性标识（未用） | 2 |
| free_block | 指向页中第一个空闲块 | 2 |
| free_size | 页中空闲块的大小 | 2 |
| size | 页中存储的项数 | 2 |
| bottom_used | 页中尾部连续使用的最长的大小 | 2 |
| next_page | B+ 树中该页的右邻居 | 4 |
| prev_page | B+ 树中该页的左邻居 | 4 |

variant 页较为复杂，各项数据可以不是连续存储的。每当删除一项时会产生碎片，碎片使用链表连接，表头和碎片数用页头中的 free_block 和 free_size 记录。由于各个数据项可以不按照顺

序存储，在紧接着页头的部分有连续存储的指向各个数据项的指针。插入和删除只会改变指针的内容。注意如果发现碎片过多可能会进行重整将数据变为连续存储。

### 3.3 记录结构

每条记录都会有一个唯一的 rowid 标识，它会作为记录 B+ 树的键。记录是存储在记录 B+ 树的页节点中。记录 B+ 树的内部节点的页结构是 fixed 页，叶子节点的结构是 variant 页。如果发现记录过长，则会按照链表的方式将溢出的数据存储在 overflow 页中。

每条记录的前 4 个字节用于存储 rowid，这也方便 B+ 树进行查询比较。紧跟着的 4 个字节用于存储该记录的项是否是空。再往下是用于存储各个列的数据，具体的存储顺序是根据创建时表的结构来决定的，记录 B+ 树不需要了解之后各个列的具体结构。

### 3.4 索引结构

索引 B+ 树的叶节点和内部节点均为 variant 页，因为索引的关键字可能是较长的字符串。索引的关键字包含三个部分：对应项的 rowid、非空标识和索引的键。这三个部分作为一个元组构成索引 B+ 树关键字，注意索引关键字的长度不能够过长。

索引关键字比较时，我们认为空值小于任何非空值，并且当两个键相同时比较 rowid。

### 3.5 表结构

每个表分为两个部分存储，表头单独存储为一个文件，表项和索引通过页式文件系统存储为一个文件。

表头主要包含各个列的信息，以及完整性约束的信息。特别需要注意的是 CHECK 约束的表达式信息被转换为波兰式存储以及解析。列的信息包含列名、列类型、列长度、列在对应记录中的存储偏移量以及列是否有索引等约束信息。另外还会存储索引 B+ 树和数据 B+ 树的根节点对应的页号。

每一个记录的存储结构为 4 个字节的 rowid，4 个字节的空标识之后按照顺序存储对应的列。

## 4 SQL 命令处理

在基础的数据结构之上有表管理器、索引管理器、数据库管理器三个部分。表管理器用于处理一个表的插入、删除和查询等操作，并且会进行数据约束的检查。索引管理器用于对索引 B+ 树进行维护。数据库管理器用于管理一个数据库的所有表。在其之上是 DBMS，它和词法/语法分析器进行交互，转发和处理查询请求。

### 4.1 SQL 语句解析

SQL 语句的解析利用 FLex/Bison 根据词法、语法规则自动解析 SQL 命令。程序会从标准输入读取信息。SQL 的词法和语法分析参考了开源项目的实现。

语法分析的过程会建立抽象语法树，在结束一条语句的处理后会释放抽象语法树的内存。整个解析模块在 parse 目录中。解析器和数据库管理系统的接口定义在 parser/execute.h 内。

### 4.2 查询处理

对于数据库结构管理的语句（建表、建索引等）处理较为简单，直接提出抽象语法树的信息翻译为系统内部的类型即可。

对于 UPDATE、DELETE、INSERT 和 SELECT 语句。其中 INSERT 语句由于不需要直接读取表中记录，直接遍历语法分析树中需要插入的内容利用表管理器插入即可。在表管理器中会有一个临时的记录，插入的过程首先对该记录进行操作，之后表管理器会将这个临时记录按照对应的记录结构插入到记录 B+ 树中，同时更新对应的索引。

另外三种语句均可能有 WHERE 子句，并且需要遍历一个或多个表的信息。我们实现一个 iterate 函数提供一个公共的遍历接口。iterate 会遍历需要的表项，并且通过计算 WHERE 字句中的条件，在满足时通过一个回调函数来针对三种不同的语句进行各自的处理。

## 4.3 查询优化

在多表连接中，如果 WHERE 字句形如 $E_1$ AND $E_2$ AND $\cdots$ AND $E_n$，其中 $E_k$ 为任意表达式。那么如果 $E_k$ 形如 $T_k.C_k = T'_k.C'_k$，其中 $T_k, T'_k$ 为某两个表，$C_k, C'_k$ 为这两个表对应的列。那么若这二者之一拥有索引，我们就有可能利用索引进行查询的加速。

具体优化过程如下：我们首先建立一个图，对于所有连接的表建立一个节点。对于所有具有上述形式的表达式 $T_k.C_k = T'_k.C'_k$，如果 $T_k.C_k$ 有索引，那么就建立一条有向边 $(T_k.C_k, T'_k.C'_k)$，对于 $T'_k.C'_k$ 有索引的情况也是类似的。之后在这个图中寻找一条最长的，经过各个节点最多一次的路径，在遍历时先遍历不在路径上的表，之后再从这条路径倒序遍历，路径上的表的遍历可以直接利用索引加速。

例如，假设有三个表 $T_1, T_2, T_3$，其中 $T_1.name$ 和 $T_2.name$ 有索引，那么如下查询就会按照 $T_3, T_1, T_2$ 的顺序查询，并且查询的过程是遍历 $T_3$，之后利用索引查询 $T_1$，再利用索引查询 $T_1$，最后计算 WHERE 子句。

```
SELECT * FROM T1, T2, T3 WHERE T1.name = T2.name AND T2.name = T3.name AND
↪  T1.gender = 'F'
```

## 4.4 表达式计算

表达式计算是在处理查询语句是一个可能需要的操作，我们实现了一个 expression 类用于保存计算结果。在 iterate 函数中，每遍历到一个记录就会将其内容加入到一个缓存中。之后根据语法分析树计算表达式的值的时候如果需要读取对某个列的引用就会在缓存中进行读取。

针对语法分析树的表达式计算是编译原理课程的内容，在此不详细叙述。

## 4.5 完整性约束

**主键约束** 对于主键约束，我们在创建表是会至少保证主键中至少有一个拥有索引。之后在检测约束是会对这一个拥有索引的列通过索引查询对应的记录，之后再检查其与主键是否完全相等。另外，如果创建表时没有主键约束，那么 rowid 就被认为是主键。

**外键约束** 对于外键，在检测约束是会利用数据库管理系统顶层模块将查询转发到对应的表中进行处理。直接通过索引判断是否有对应的值即可。

**CHECK 约束** 该类约束在处理时先缓存表内容，之后利用表达式计算引擎计算对应表达式信息即可。

**UNIQUE 约束** 这类约束的实现和主键约束类似，直接通过索引判断即可。

**NOT NULL 约束** 这类约束实现较为简单，直接判断记录的空值记录区域。

**DEFAULT 约束**　这类约束直接在初始化一个记录时将其对应内容填充为给定的值即可。