项目报告

功能实现

基本运行

`MyDBMS` 运行后，用户可以通过命令行不断输入 `sql` 语句执行对应操作，也可以导入含有若干 `sql` 执行语句的文件进行输入。

操作执行完成后会返回执行信息，并包含执行耗时。**输出格式基本遵从前端约定**。

若执行过程中出现错误，`MyDBMS` 会向用户输出错误信息。

提供终端中的颜色区分，以增加信息的可读性。用户的输入将用白色显示，`MyDBMS` 的信息输出将用蓝色显示，而错误信息输出将用红色显示。

用户可以使用额外语句 `EXIT;` 来退出 `MyDBMS` 。

运行的入口实现在 `utils/Application` 中，而读入用户指令并递交解析与输出信息与结果实现在 `utils/frontend` 中。

系统管理

支持的 SQL 语句有（具体格式与大作业下发文法要求中一致）：

- `CREATE DATABASE ...;`：创建数据库
- `DROP DATABASE ...;`：删除数据库
- `USE ...;`：切换当前数据库
- `SHOW TABLES;`：列出所有表
- `SHOW INDEXS;`：列出所有索引
- `CREATE TABLE ...;`：创建表
  - 支持的数据类型有
    - 整数（`INT`）
    - **字符串（`VARCHAR`），根据实际大小占用磁盘空间，详情见记录管理模块的设计细节一节**
    - 浮点数（`FLOAT`）
    - **空值（`NULL`）**
  - 支持 `NOT NULL` 的非空字段限制
  - 支持 `DEFAULT value` 的默认值字段限制
  - 支持主键约束 `PRIMARY KEY ...`，并会对合法性进行判断
  - 支持外键约束 `FOREIGN KEY ...`，并会对合法性进行判断

- `DESC ...;`：列出表的信息

主要由系统管理模块负责执行。

查询解析

支持的 SQL 语句有（具体格式与大作业下发文法要求中一致）：

- `INSERT INTO ... VALUES ...;`：插入数据
- `DELETE FROM ... WHERE ...`：删除数据
- `UPDATE ... SET ... WHERE ...`：更新数据
- `SELECT ... FROM ... WHERE ...`：查询符合条件的数据

查询语句则支持下列特性：

- 多表 join ，并通过查询策略进行了性能优化
- 全部的 `where` 子句，且支持多个 `where` 子句通过 `AND` 进行连接
  - `WHERE table.column op expr`：根据一列和一个常数或另一列的值比较结果进行查询
  - `WHERE table.column op select_expr`：上一种 `where` 子句的**嵌套查询**版
  - `WHERE table.column IS (NOT)? NULL`：根据是否是空值进行查询
  - `WHERE table.column IN value_list`：根据一列值是否在给定集合内进行查询
  - `WHERE table.column IN select_expr`：上一种 `where` 子句的**嵌套查询**版
  - `WHERE table.column LIKE str`：**模糊查询**
- 聚合查询：除基本的聚合投影算子的支持外，还对与 `NULL` 类型的交互做了正确的处理
  - `AVG`
  - `MAX`
  - `MIN`
  - `SUM`
  - `COUNT`，可以以 `COUNT(*)` 形式使用
- 分组查询：`GROUP BY` 子句，可与聚合查询一起使用，并对与 `NULL` 类型的交互做了正确的处理
- 分页查询：`LIMIT ... (OFFSET ...)?` 子句

主要由查询执行模块负责执行。

完整性约束及模式管理

支持的 SQL 语句有（具体格式与大作业下发文法要求中一致）：

- `ALTER TABLE ... ADD CONSTRAINT ... PRIMARY KEY ...`：添加主键约束
- `ALTER TABLE ... DROP PRIMARY KEY ...`：删除主键约束
- `ALTER TABLE ... ADD CONSTRAINT ... FOREIGN KEY ...`：添加外键约束

- `ALTER TABLE ... DROP FOREIGN KEY ...`：删除外键约束
- `ALTER TABLE ... ADD UNIQUE ...`：添加**唯一约束**

主要由系统管理模块负责执行。

除了相关语句外，在所有可能涉及约束判断的语句中（如建表、删表、记录修改、约束修改等），系统管理模块都会提供合法性判断，并在操作非法时输出报错信息。

此外，`MyDBMS` 可以支持联合主键与联合外键的约束形式。

索引模块

支持的 SQL 语句有（具体格式与大作业下发文法要求中一致）：

- `ALTER TABLE ... ADD INDEX ...`：添加索引
- `ALTER TABLE ... DROP INDEX ...`：删除索引

主要由系统管理模块负责执行。

在 `MyDBMS` 中，除了显式增删索引，主键约束的增加或删除也会自动增加或删除索引文件。

在数据集上可验证索引的结果与性能，添加索引后速度显著超过添加索引前，详情见实验结果一节。

系统架构设计

该项目由如下几个主要模块组成：

- 页式文件系统（`fs`）
- 记录管理（`rs`）
- 索引管理（`is`）
- 查询解析（`qs`）
- 系统管理（`ms`）
- 用户前端（在 `utils` 中）

它们之间的整体架构与实验指导书中所述类似。各模块的具体实现细节和相互关联见后续内容。

各模块详细设计

页式文件系统

`fs` 目录下实现了一个简单的页式文件管理系统。它由三个类（`Filesystem`、`File`、`BufferManager`）和一个结构（`BufferPage`）构成，呈现出如下架构：

- `Filesystem` 是整个文件系统的入口类。它负责管理文件系统中用到的所有 `File` 对象，并维护了一个缓存管理器 `BufferManager`。一个文件系统中的所有 `File` 共享一个 `BufferManager` 的引用。
- `File` 类封装了文件读写相关的系统函数。由于这是一个页式文件管理系统，`File` 类对外暴露的只有 `get_page` 和 `close` 方法，其它文件操作相关的方法或者不提供，或者由

`Filesystem` 类统一管理。

- `BufferManager` 是一个基于 LRU 算法的缓存管理系统，用于对页进行缓存管理。每当一个 `File` 对象希望获取一个页时，它不会立即进行文件读写，而是会先尝试从 `BufferManager` 寻找缓存，并根据是否命中的情况考虑是否进行文件读写。此外，当一个 `File` 对象关闭时，它也会从 `BufferManager` 中获取所有属于它的脏页，并进行写回。
- `BufferPage` 结构用于描述一个缓存页。调用 `File::get_page` 方法将会得到一个 `BufferPage` 的指针。如果需要对 `BufferPage` 进行写操作，用户需自行将 `dirty` 字段改为 `true`；当这片缓存被交换或文件被关闭时，脏缓存页的内容会被写回。一旦通过 `File::get_page` 方法获得了一个 `BufferPage` 引用，调用者应尽快使用，以防该 `BufferPage` 因为被交换出缓存队列而失效。

文件系统中涉及的所有编号均从 0 开始。

文件系统的使用样例可参见 `tests/fs-test.cpp`。

记录管理

`MyDBMS` 的数据库文件布局基本遵循实验指导书上的约定，即每个数据库对应一个目录，每张表存储在单独的文件中。

`rs` 目录下实现了一个**支持变长记录**的记录管理系统。它由两个类（`RecordSystem`、`RecordFile`）和两个结构（`RecordFileMeta`、`RecordPageHeader`）构成，呈现出如下架构：

- `RecordSystem` 是整个记录管理系统的入口类。它包含一个文件系统 `Filesystem` 对象。所有记录文件都必须通过 `RecordSystem` 创建，因为通过这种方式创建的文件会包含一些记录文件的元信息。
- `RecordFile` 类包含了一个 `File` 对象。它在 `File` 类的基础上进一步封装，提供了一些读写记录相关的方法。在更上层的模块中，一个 `RecordFile` 对象将会对应一张表。
- `RecordFileMeta` 结构用于描述记录文件的元信息，各字段的含义见代码注释。
- `RecordPageHeader` 结构用于描述记录文件中数据页的页头，各字段的含义见代码注释。

除非特殊说明，记录管理系统中涉及的所有编号均从 0 开始。

记录管理系统的使用样例可参见 `tests/rs-test.cpp`。

记录文件格式约定

定义 $K = 4096$ 。

- $Page(0)$ 为元信息页。该页的数据格式与 `RecordFileMeta` 结构一致。
- $Page(n(K+1)+1)$ 为空闲空间信息页。该页存储了一个包含 K 个页节点和 $K-1$ 个非页节点的满二叉树，每个节点占用 1 字节的空间。二叉树的根节点编号为 1 。二叉树第 k 个节点的两个页节点编号分别为 $2k$ 和 $2k+1$ 。出于实现方便考虑，该页的首个字节不予使用，从下一个字节开始存放二叉树。约定空页的空闲空间情况用 `0xFF` 表示。

- 其余页为数据页。每个数据页的前 64 个字节保留给页头数据使用，页头的数据格式与 `RecordPageHeader` 结构一致。随后是每个数据页的数据部分，从前向后存储记录，从后向前存储槽目录。记录储存格式与实验指导书上所述一致。槽目录数据按照自然数编号从后往前存储，每 2 字节依次表示对应记录的字节偏移量。无效槽的字节偏移量约定为 0 。

空闲空间信息维护

定义 $K = 4096$ ，即本系统一页字节数的一半。

本系统采用大根堆二叉树的数据结构来维护空闲空间信息。出于实现方便考虑，本系统将空闲空间信息页与数据页放在同一个记录文件中进行管理。

每一个空闲空间信息页可以维护一棵包含 K 个页节点的满二叉树，其中每个节点占用 1 字节的空间。每个节点代表一个数据页的空闲空间情况（这里需要将空闲空间大小分成 256 个级别，这样每个节点即可仅占用 1 字节的空间），因此每一个空闲空间信息页可以维护 K 个数据页的空闲信息。

只进行到这里，本系统能够支持的最大的记录表也就只有 K 个数据页了。为了能够提供更大的可拓展性，本系统采用两级空闲空间信息页来维护空闲空间信息。具体而言，本系统允许一个记录表中存在多个空闲空间信息页，每个空闲空间信息页根节点的数值则连续存放在文件的元信息页中。这也解释了 `RecordFileMeta` 结构中的 `fsp_cnt` 和 `fsp_data` 的含义。本系统约定，单个记录文件中至多有 63 个空闲空间信息页。至此，本系统能够支持的单张表数据容量的理论上限为 $63 \times 4096 \times 8K \approx 2G$ ，这已超过最终测试的总数据量。

索引管理

`MyDBMS` 的索引同样是以一个文件的形式保存，一个文件保存每个表关于某一列的单列索引信息，一张表可以拥有多个索引文件，上层是由系统管理模块用额外的方式对表和索引进行管理。索引管理模块提供最基本的索引文件管理能力，包括创建索引、删除索引、获得一个索引文件的句柄以及往索引文件里增删查改。

`is` 目录下实现了一个基于 B+ 树的**单列整型**索引管理系统。它由两个类（`IndexSystem`、`IndexFile`）和两个结构（`IndexFileMeta`、`IndexPageHeader`）构成，呈现出如下架构：

- `IndexSystem` 是整个索引管理系统的入口类。它包含一个文件系统 `Filesystem` 对象。所有索引文件都必须通过 `IndexSystem` 创建，因为通过这种方式创建的文件会包含一些索引文件的元信息。
- `IndexFile` 类包含了一个 `File` 对象。它在 `File` 类的基础上进一步封装，提供了一些读写索引相关的方法。在更上层的模块中，一个 `IndexFile` 对象将会对应一张表的一个索引。
- `IndexFileMeta` 结构用于描述索引文件的元信息，各字段的含义见代码注释。
- `IndexPageHeader` 结构用于描述索引文件中数据页的页头，各字段的含义见代码注释。

B+ 树的参数 `m` 保存在元信息页里，如果实例化时没有设置，默认设置为 `255` 。通过后续的内存布局细节也可以计算出， `255` 是本索引管理系统可以支持的最大合理值。

索引管理系统的使用样例可参见 `tests/is-test.cpp`。

索引文件格式约定

第 `0` 页为元信息页。该页的数据格式与 `IndexFileMeta` 结构一致。内容包括的信息有 B+ 树的阶数 `m` ，B+ 树的总结点数，B+ 树当前根节点的页号。

其余页每页代表一个 B+ 树结点，每个数据页的前 `64` 个字节保留给页头数据使用，页头的数据格式与 `IndexPageHeader` 结构一致，内容包括该结点是否是叶子结点，其内部目前含有多少关键码，父亲结点的页号是多少，以及内部关键码链表和叶子间链表的必要信息。

随后是数据部分，数据部分将从前往后放置键值对， `key` 占用 `4` 字节， `value` 占用 `4` 字节，共占用 `8` 字节。对于非叶子结点，其键值对 `(key, value)` 的 `value` 表示的是关键码 `key` 对应的左子树是哪一页。对于叶子结点，其键值对 `(key, value)` 的 `value` 表示的是关键码 `key` 作为索引列的值，将映射到记录文件中的哪一条记录上（即 `RID` ）。每个关键码在一个结点上都有自己的编号，由编号即可定位到存储位置。同时，为了维护关键码间的次序保证 B+ 树的实现效率，结点上需要维护编号间形成的链表。对于每个关键码，链表信息包括前一项和后一项关键码的编号，在布局中是从后往前放置链表信息，一个关键码的链表信息占用 `2` 字节。

为了方便处理，每个数据页初始时就含有一个编号为 `0` 的关键码，其键值为 `100000000` ，即正无穷。对于非叶节点，该编号关键码可以用来存储整个节点最右边的子树。

至此，我们可以计算为何 `255` 是该布局下的最大合理阶数。由于 `value` 需要存储 `RID` ，因此必须需要 `4` 字节的空间（ `RID` 由 `page_id` 和 `slot_id` 组成，页号上限可按 `60000` 估计，槽号上限可按页大小即 `8192` 估计，分别需要 `2` 个字节）。 `key` 存储的整型范围同样需要 `4` 字节。

如果阶数不超过 `255` ，那么每个编号均可以用 `1` 字节表示，链表信息每个关键码只需要 `2` 字节，因此 `256` （算上编号为 `0` ） 个关键码一共需要 `(8+2)*256+64=2624<8192` 字节。

叶子链表

本 B+ 树实现了叶子间的链表，即每一个叶子结点前一个和后一个叶子结点是哪一页，存储在数据页的页头文件中。叶子间链表只需要在执行增和删时进行一些额外维护即可。

索引管理系统同时还提供出了一个迭代器类型 `IndexScan` 。

对于查询操作，只需要定位 `lower_bound` 所在的叶子与关键码编号，便可自动构造对应的迭代器。迭代器会同时根据结点内关键码间的链表与叶子间的链表快速找到下一个值，大大增加了查询性能。

对于删除操作，支持删除一个区间的关键码，其原理同样是可以根据叶子间链表把删除转化为删除多个叶子。其中，**删除一个叶子可能导致的过接和下溢不会影响其他叶子的存在性，这意味着删除多个叶子是数个独立的子问题。**

解析器

通过 `antlr` 对下发的文法文件 `SQL.g4` 生成了对应的框架代码 `generated` ，并将官网上下载的 C++ 版 `antlr_runtime` 作为库与项目代码链接编译。

自行实现的解析器在 `src/MyVisitor.h` ，通过继承 `SQLBaseVisitor` 实现 Vistor 模式，将读入的 `sql` 语句进行解析。

`MyDBMS` 的解析器类似于解释器，当一条语句的基本要素以及语句类型都完成解析时，解析器会立刻将该语句分发给两大上层模块（查询执行模块与系统管理模块）其中之一，让其处理。

属性文法

解析各要素的实现主要依赖属性文法，即在抽象语法树的不同类型结点上保存一定的信息（其中许多是由查询执行模块或系统管理模块提供好了的封装类型），在递归时利用子节点计算出属性信息，计算本节点的属性信息。有了属性信息后，即可方便地调用执行模块的接口。

查询执行

在 `qs` 目录下实现了一个查询执行器。对于增删查改（如 `INSERT INTO TABLE` 、 `SEARCH ...` ）类 `sql` 语句，解析器将会把执行的权限交给查询执行器，此时查询执行器为主动模块，需要自行组织所有的处理行为，并给出语句的执行结果。查询执行模块需要系统管理模块提供与系统管理、文件管理相关的辅助功能。

记录修改

在 `MyDBMS` 中，查询执行模块与系统管理模块的职能区分是清晰的，记录修改的部分虽然主要由查询执行模块组织和发起，但大部分的原子功能需要由系统管理模块提供，这是因为系统管理模块能直接管理表、索引、主外键约束，因而大部分信息对查询执行模块实际上是透明的。

添加/删除记录前，查询执行模块需要向系统管理模块确认插入/删除数据的合法性，若非法则进行报错，否则通过记录管理模块的接口插入/删除记录，同时将记录信息插入/删除该表上创建的相关索引中。

更改记录从行为上可以拆分为一次删除加一次添加，但如果涉及到外键约束则不能等价。我们的实现是将判断删除数据是否合法的接口进行了拓展，使之可以仅对某些列作判断，而忽略其他列。因此更改记录时只需要对涉及更改列进行检查即可。

查询策略

在 `qs/RecordSet.h` 下封装了对于查询结果的存储结构，这也被用于承担中间结果。

由于 `MyDBMS` 没有实现物化策略，使用该类作中间结果在函数间传递对于特别大的表项可能会出现崩溃。

基本查询

查询执行器一共提供了三种基本查询方法，可以根据不同查询执行计划的需要调用：

- 整表扫描（`search_whole_table`）：由记录管理模块提供相应接口，获取一张表的所有记录构成的查询结果。

  典型应用场景：`SELECT * FROM table`
- 按索引查找（`search_by_index`）：从一个表中以某一列为索引找出位于某一区间的所有记录构成的查询结果，**只有该列建有索引时才有效。**

  典型应用场景：`SELECT * FROM table WHERE table.column = const`
- 以一个表驱动另一个表查找（`search_by_another_table`）：根据一个表（`table1`）的某查询结果，一个查询条件（形式必须是 `table1.column1 = table2.column2`），从另一个表（`table2`）中得出查询结果。实现是枚举第一个表的查询项，代入到查询条件中转变为 `table2.column2 = const` ，然后利用 `table2` 建在 `column2` 上的索引加速查找。

  典型应用场景：`SELECT * FROM table1, table2 WHERE table1.column1 = table2.column2`

查询执行计划

见 `qs/QuerySystem.cpp` 。

查询执行模块在查询操作上对外暴露两个接口，分别是 `search_entry` 与 `search` ，后者可以对任意一条查询语句计算出查询结果，前者则标识查询语句执行的入口，最终还要将查询结果输出给用户。对于嵌套查询语句，也只会调用一次 `search_entry` ，但每一个查询子句都会调用一次 `search` 。

此外， `search` 还可以被删除记录、更改记录以及约束管理的对应接口使用，因为这些接口都需要对记录进行筛选，可以转化为等价的查询语句。

查询执行模块实现了许多私有接口，来处理查询语句中不同的算子：

- `search_where_clause` ：处理带单个选择算子的查询，采用暴力算法。较为值得一提的是模糊查询的支持，可将模糊查询用的匹配串转化为等价正则表达式后，用 `regex` 库的现成接口判断匹配
- `search_selector` ：处理带单个投影算子的查询，采用暴力算法。聚合查询、分组查询也在这个接口中处理
- `search_where_clauses` ：处理所有的选择算子，采用一定的查询策略进行优化，最后没法纳入策略里的 `where` 子句再调用 `search_where_clause` 处理
- `search_selectors` ：处理所有的投影算子，拆分成单个投影算子的函数叠加，分别调用 `search_selector` 处理

`search` 先调用 `search_where_clauses` 处理掉所有的选择算子，再调用 `search_selectors` 处理掉所有的投影算子。

查询优化策略

即 `search_where_clauses` 中对查询的优化。

优化策略是尽可能的利用索引，利用方式有两种：直接通过一个表的索引查询该表，通过另一个表驱动间接通过一个表的索引查询该表。

因为多表 join 的可能，一开始先扫描所有 `where` 子句，如果一个表可以通过索引查询（即有类似 `table.column op const` 的条件），则可以调用 `search_by_index` 查询。其余的简单子句（如 `table.column IS NULL` 、 `table.column LIKE STR` 、`table.column IN value_list` 等）也可以直接调用 `search_where_clause` 直接处理。

在将多个表 join 之前，先对每个单表按上述策略处理掉一部分选择算子。这种策略也被称为**谓词下推**。

接下来将尝试多个表得到的结果 join 在一起，将根据剩余的诸如 `table1.column1 = table2.column2` 的子句按一定次序处理每个表的结果。例如 `table1` 已经经过一系列选择算子，或已经将 `table1` 作整表查询，那么接下来可以用 `table1` 驱动查询 `table2` ，调用 `search_by_another_table` ，消耗掉对应的选择算子。

将所有表 join 起来后，还剩下没消耗的选择算子依次调用 `search_where_clause` 处理。

系统管理

系统管理模块有三个主要职能：维护数据库和表的系统信息，执行用户输入的系统管理指令，以及为下层模块提供必要的系统接口。

维护系统信息

一共有数据库信息和各个数据库的数据表信息需要维护。

需要维护的数据库信息包含：

- 各个数据库的名称；
- 各个数据库在系统内的唯一编号。

存放于 `global/global.txt` 文件中。

需要维护的数据表信息包含：

- 各张表的名称；
- 各张表在系统内的唯一编号；
- 表中各个域的名称、类型、字符串长度、NULL、默认值等信息；
- 主键、外键、索引、唯一性等信息。

存放于 `global/<db-id>.txt` 文件中。

系统管理模块需要负责查询、修改这部分的信息。

执行用户指令

管理模块需要负责执行的用户指令有：

- 创建、查看、删除、使用数据库；

- 创建、查看、删除、描述数据表；
- 创建、删除索引、主键、外键、唯一约束。

提供必要接口

这一部分职能的主要考虑点是，只有系统管理模块具有各个表的域和约束信息。甚至连各张表存储于什么路径下，都只有系统模块知道。因此，其它模块必须借助系统管理模块提供的接口，才能完整实现功能。

这部分具体需要提供的接口见**主要接口说明**一节。

测试

本系统采用 CMake 进行项目管理，使用 CTest 进行测试。测试的内容分为单元测试和集成测试。

单元测试

单元测试是针对各模块进行的测试，涵盖的范围有文件系统模块、记录管理模块、索引管理模块和系统管理模块等。我们分阶段针对各个模块设计了充分的单元测试，力求尽早发现、排查与解决基础模块的问题。

集成测试

集成测试则是针对系统功能进行的测试，主要的测试逻辑是，给定输入的 sql 指令序列，验证输出结果是否与预期一致。为了实现集成测试，我们首先对原本基于 stdio 的前端进行拓展，使之同时能够支持使用 stringstream 进行输入输出。此外，我们设计了一个 `TestSystem` 类，暴露 `exec`（只执行，不验证）和 `expect`（执行并验证）两个方法。这样，我们就能进行集成测试了。相关代码参见 `tests/dbms-test.cpp`。

持续集成

我们使用 GitHub CI 进行持续集成测试，配置文件见 `.github/workflows/cmake.yml`。

主要接口说明

本部分主要描述各个模块暴露出的公有接口，这些接口能够体现各个模块如何提供完成相应职能的能力，并为其他模块所调用。如对模块内部的实现及重要方法感兴趣，请看模块设计细节或相应代码。

文件系统

FileSystem

整个页式文件管理系统的入口类，负责管理文件系统中用到的所有 File 对象，并维护了一个缓存管理器 BufferManager。所有 File 共享一个 BufferManager 的引用。

- `static void create_file(const char *filename)`：在磁盘中创建一个文件。若文件名已存在，则无事发生
- `static void remove_file(const char *filename)`：从磁盘中删除一个文件

- `File *open_file(const char *filename)`：打开一个文件，创建一个 File 对象，并返回它的指针

File

封装了文件读写相关系统函数的类。

- `BufferPage *get_page(std::size_t page_id, bool bypass_search = false)`：获取一个缓存页。如果需要对获取到的缓存页进行写操作，用户需自行将 BufferPage::dirty 字段改为 true；当这片缓存被交换或文件被关闭时，脏缓存页的内容会被写回。一旦通过该方法获得了一个缓存页的指针，调用者应尽快使用，以防该页因为被交换出缓存队列而失效。
- `void close()`：关闭文件，如果缓存中有该文件的脏页，则将脏页写回磁盘

记录管理

RecordSystem

整个记录管理系统的入口类，包含一个文件系统 Filesystem 对象。所有记录文件都必须通过该类创建、删除或打开。

- `void create_file(const char *filename)`：创建一个记录文件，并维护该文件的元信息
- `void remove_file(const char *filename)`：删除一个记录文件，等价于从文件系统中直接删除文件
- `RecordFile *open_file(const char *filename)`：打开一个记录文件，创建一个 RecordFile 对象，并返回它的指针。调用者需自行保证该文件是一个记录文件。

RecordFile

在 File 类的基础上进一步封装，提供了一些读写记录相关的方法。在更上层的模块中，一个 RecordFile 对象将会对应一张表。

用封装类 `RID` 唯一标识一条记录，其中包含两个 `u_int16_t` 类型，分别表示 `page_id` 页号和 `slot_id` 槽号。

- `RID insert_record(std::size_t record_size, const char *record)`：插入一条记录
- `void delete_record(const RID &rid)`：删除一条记录
- `RID update_record(const RID &rid, std::size_t record_size, const char *record)`：更改一条记录
- `std::size_t get_record(const RID &rid, char *record)`：获取一条记录
- `RID find_first() const`：获取第一条记录，需要配合 `find_next` 使用，用于提供遍历全表的能力
- `RID find_next(const RID &rid) const`：获取下一条记录

索引管理

IndexSystem

整个索引管理系统的入口类，包含一个文件系统 Filesystem 对象。所有索引文件都必须通过该类创建、删除或打开。

- `void create_file(const char *filename)`：创建一个索引文件，并维护该文件的元信息
- `void remove_file(const char *filename)`：删除一个索引文件，等价于从文件系统中直接删除文件
- `IndexFile *open_file(const char *filename)`：打开一个索引文件，创建一个 IndexFile 对象，并返回它的指针。调用者需自行保证该文件是一个索引文件。

IndexFile

在 File 类的基础上进一步封装，提供了一些读写索引相关的方法。在更上层的模块中，一个 IndexFile 对象将会对应关于一张表的一个整型列所创建的索引。

- `void insert_record(int key, const RID &rid)`：往索引文件里插入一条记录
- `void delete_record(int key, const RID &rid)`：往索引文件里删除一条记录，需保证对应记录存在
- `void update_record(int old_key, const RID &old_rid, int new_key, const RID &new_rid)`：往索引文件里更新一条记录，需要保证被更新的记录存在
- `void search(int lower_bound, int upper_bound, IndexScan &index_scan)`：查找某个范围内的记录，结果通过迭代器访问。注意如果修改了索引文件，迭代器会失效，调用者应在此之前使用完

IndexScan

进行查找操作时得到的类似迭代器的封装类。

- `bool get_next_entry(RID &rid)`：获取迭代器中的下一条记录，返回是否获取成功

查询执行

QuerySystem

查询执行模块，负责处理由解析器移交给它的记录修改与记录查询语句。

- `void insert_record(std::string table_name, std::vector<Value> values)`：插入一条记录
- `void delete_record(std::string table_name, std::vector<WhereClause> where_clauses, std::vector<bool> v)`：删除一条记录
- `void update_record(std::string table_name, std::vector<std::string> column_names, std::vector<Value> update_values,`

`std::vector<WhereClause> where_clauses)`：更新一条记录

- `RecordSet search(SelectStmt select_stmt)`：根据封装的查询子句作查询，返回查询结果
- `void search_entry(SelectStmt select_stmt)`：根据封装的查询子句作查询，给用户输出查询结果

系统管理

Field、PrimaryField、ForeignField

分别用于表示单个字段、主键约束和外键约束的信息。具体含义参见 `ms/Field.h` 相关代码。

Value

用于存储和表示数值。基本思想是利用 `void *` 对象，借助表示数据类型的枚举类，实现动态数据类型。

对外暴露的字段和方法有：

- 枚举类 `type`，有 `NUL`、`STR`、`INT`、`FLOAT` 四种取值。
- 构造方法。
- `asInt`、`asString`、`asFloat`、`isNull` 等获取数值的方法。
- 加法运算符。对整型和浮点型支持加法运算。
- 比较运算符。比较原则为：
  - 两个 `NUL` 类永远相等；
  - 两个不同类型的大小关系取决于类型枚举值的序；
  - 两个相同非空类型直接比较数值。

ManageSystem

系统管理模块，主要负责管理维护系统级信息，执行用户的系统管理指令，并为其它模块提供必要接口。

一个 `ManageSystem` 实例中有一个 `RecordSystem` 对象和一个 `IndexSystem` 对象，并保存一个 `QuerySystem` 的引用。此外，`ManageSystem` 还负责保存前端 `Frontend` 的引用。由于其他模块需要通过 `ManageSystem` 打开记录文件和索引文件，`ManageSystem` 还需维护当前打开的记录文件和索引文件名称与对象的映射关系。最后，`ManageSystem` 还需维护当前打开的数据库的信息。

- 公有字段 `const Frontend *frontend`、`QuerySystem *qs`：含义如前所述。
- 静态方法 `static ManageSystem load_system(const std::string &root_dir, const Frontend *frontend)`：用于构造一个管理系统对象，需要提供数据库系统的根目录以及数据库前端的引用。
- `bool ensure_db_valid() const`：用于验证当前是否有数据库打开。

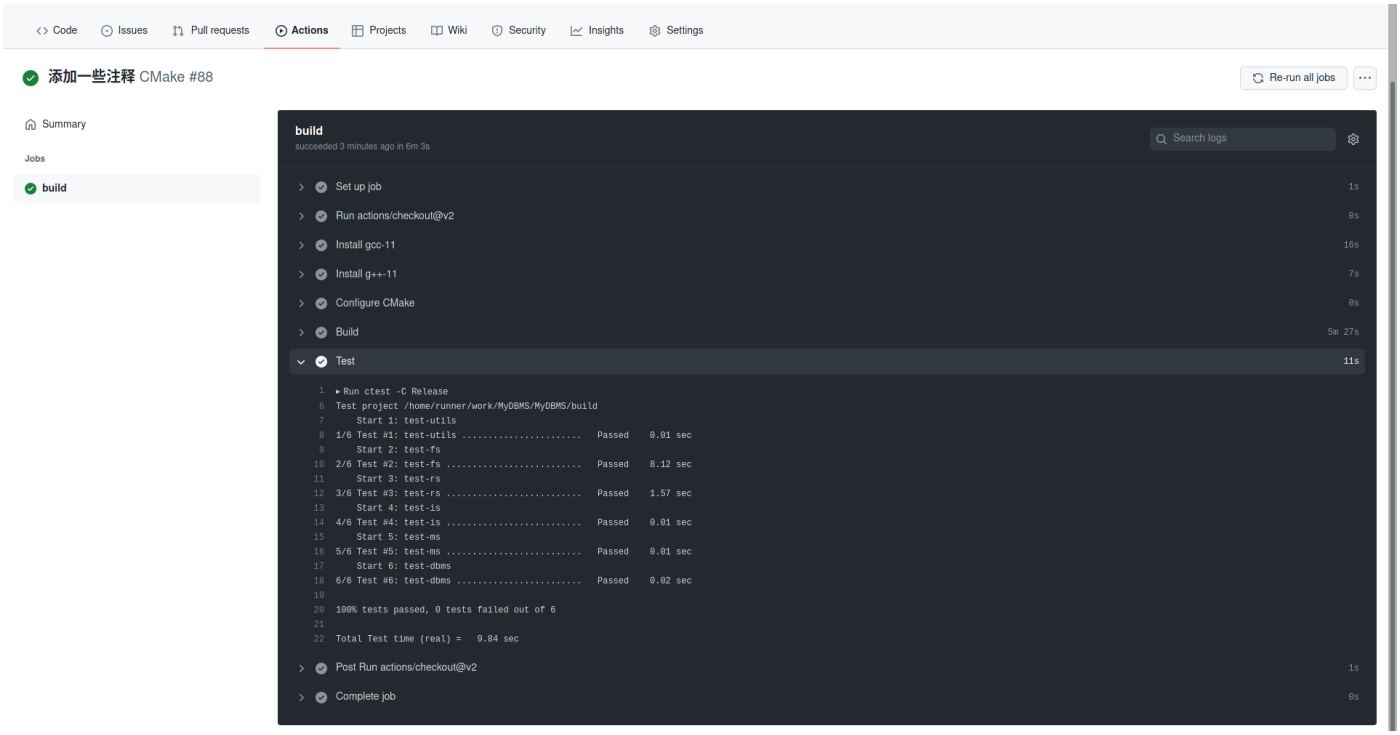
- `void create_db(const std::string &db_name)`：用于执行用户打开数据库的指令。
- `void drop_db(const std::string &db_name)`：用于执行用户删除数据库的指令。
- `void show_dbs()`：用于执行用户展示数据库的指令。
- `void show_tables()`：用于执行用户展示数据表的指令。
- `void use_db(const std::string &db_name)`：用于执行用户打开数据库的指令。
- `void create_table(const std::string &table_name, const std::vector<Field> &field_list, const std::vector<PrimaryField> &primary_field_list, const std::vector<ForeignField> &foreign_field_list)`：用于执行用户创建数据表的指令。
- `void drop_table(const std::string &table_name)`：用于执行用户删除数据表的指令。
- `void describe_table(const std::string &table_name)`：用于执行用户描述表的指令。
- `void create_index(const std::string &table_name, const std::vector<std::string> &column_list)`：用于执行用户创建索引的指令。
- `void drop_index(const std::string &table_name, const std::vector<std::string> &column_list)`：用于执行用户删除索引的指令。
- `void add_primary_key(const std::string &table_name, const PrimaryField &primary_restriction)`：用于执行用户添加主键的指令。
- `void drop_primary_key(const std::string &table_name, const std::string &restriction_name)`：用于执行用户删除主键的指令。
- `void add_foreign_key(const std::string &table_name, const ForeignField &foreign_restriction)`：用于执行用户添加外键的指令。
- `void drop_foreign_key(const std::string &table_name, const std::string &restriction_name)`：用于执行用户删除外键的指令。
- `void add_unique(const std::string &table_name, const std::vector<std::string> &column_list)`：用于执行用户添加唯一约束的指令。
- `Error::InsertError validate_insert_data(const std::string &table_name, const std::vector<Value> &values)`：查询一个数据是否可插入。给出表名和插入的数据各项的值，返回一个枚举类型，表示是否合法，若非法其原因是什么。数据各项的值顺序和建表时列的顺序一致。
- `Error::DeleteError validate_delete_data(const std::string &table_name, const std::vector<Value> &values, const std::vector<bool> &mask)`：查询一个数据是否可删除。给出表名和删除的数据各项的值，返回一个枚举类型，表示是否合法，若非法其原因是什么。

- `char *from_record_to_bytes(const std::string &table_name, const std::vector<Value> &values, std::size_t &length)`：获取一个记录对应的字节序列。给出表名和插入的数据各项的值（保证是合法的，无需再检验），将其转化成对应的字节序列返回，同时需要给出序列长度。返回的字节序列需要由调用者自行释放。
- `std::vector<Value> from_bytes_to_record(const std::string &table_name, char *buffer, std::size_t length)`：获取一个字节序列对应的记录。给出表名、字节序列以及字节数，返回数据各项的值（保证记录是合法的，无需再检验）。
- `std::vector<std::size_t> get_index_ids(const std::string &table_name)`：获取一个表上的所有索引。
- `bool is_index_exist(const std::string &table_name, const std::string &column_name)`：查询一个表上是否有某个索引。
- `std::string get_column_name(const std::string &table_name, std::size_t column_id)`：获取一个表某一列的名称。给出表名和列的id（从`0`开始编号，按建表时的顺序增长。`ALTER`语句增加的列，id设成当前最大id值加一。`ALTER`语句如果删除一列，需要将原本id大于它的列id减一），返回该列的名称。
- `bool is_column_exist(const std::string &table_name, const std::string &column_name)`：查询一个表上是否有某列。
- `Field get_column_info(const std::string &table_name, const std::string &column_name)`：获取一个表某一列的字段信息。
- `bool is_table_exist(const std::string &table_name)`：获取一张表是否存在于当前打开的数据库中。
- `RecordFile *get_record_file(const std::string &table_name)`：获取一张表的记录文件对象。系统管理模块需要自行保证同名文件不被重复打开。
- `IndexFile *get_index_file(const std::string &table_name, const std::string &column_name)`：获取一个索引的索引文件对象。系统管理模块需要自行保证同名文件不被重复打开。
- `std::size_t get_record_length_limit(const std::string &table_name)`：获取一张表的记录长度上限。
- `std::size_t get_column_num(const std::string &table_name)`：获取一张表的列数。

实验结果

CI 结果

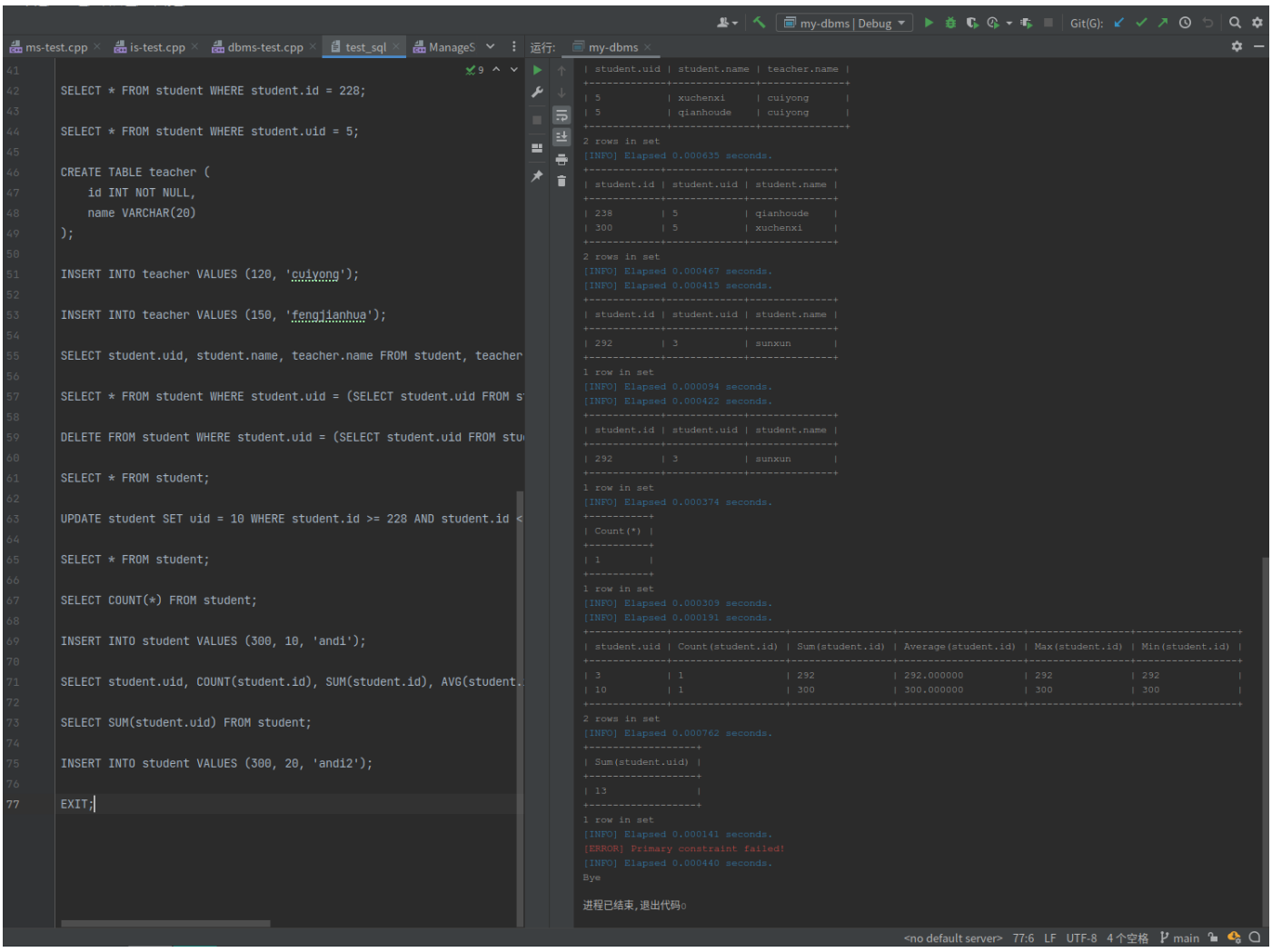
这是 GitHub CI 的运行结果截图：



指令执行

本系统可以正确地与用户进行交互，执行用户指令，输出执行结果。

这是执行仓库中 `tests/test_sql` 脚本的运行结果截图：



性能测试

本系统有效利用索引进行加速。以下是使用与不使用索引两种情形下的执行结果截图。

```
USE DATASET;
[INFO] Database changed.
[INFO] Elapsed 0.000790 seconds.
SELECT LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY, LINEITEM.L_ORDERKEY, LINEITEM.L_LINENUMBER, LINEITEM.L_COMMENT FROM LINEITEM
 WHERE LINEITEM.L_SUPPKEY < 10 AND LINEITEM.L_COMMENT >= 'z';
+--------------------+--------------------+--------------------+----------------------+------------------------------+
| LINEITEM.L_PARTKEY | LINEITEM.L_SUPPKEY | LINEITEM.L_ORDERKEY | LINEITEM.L_LINENUMBER | LINEITEM.L_COMMENT           |
+--------------------+--------------------+--------------------+----------------------+------------------------------+
| 28484              | 3                  | 1072288            | 4                    | zzle quickly according to the |
| 32247              | 8                  | 1288486            | 5                    | zle carefully bold deposits   |
| 30720              | 1                  | 1704965            | 4                    | ze boldly among th            |
+--------------------+--------------------+--------------------+----------------------+------------------------------+
3 rows in set
[INFO] Elapsed 37.156487 seconds.
ALTER TABLE LINEITEM ADD INDEX (L_SUPPKEY);
[INFO] Query OK, 0 rows affected.
[INFO] Elapsed 36.608055 seconds.
SELECT LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY, LINEITEM.L_ORDERKEY, LINEITEM.L_LINENUMBER, LINEITEM.L_COMMENT FROM LINEITEM
 WHERE LINEITEM.L_SUPPKEY < 10 AND LINEITEM.L_COMMENT >= 'z';
+--------------------+--------------------+--------------------+----------------------+------------------------------+
| LINEITEM.L_PARTKEY | LINEITEM.L_SUPPKEY | LINEITEM.L_ORDERKEY | LINEITEM.L_LINENUMBER | LINEITEM.L_COMMENT           |
+--------------------+--------------------+--------------------+----------------------+------------------------------+
| 30720              | 1                  | 1704965            | 4                    | ze boldly among th            |
| 28484              | 3                  | 1072288            | 4                    | zzle quickly according to the |
| 32247              | 8                  | 1288486            | 5                    | zle carefully bold deposits   |
+--------------------+--------------------+--------------------+----------------------+------------------------------+
3 rows in set
[INFO] Elapsed 0.113246 seconds.
```

小组分工

我们按照阶段进行分工，具体如下：

step 1: 文件管理	@孙迅
step 2: 记录管理	@孙迅
step 3: 索引管理	@王之栋
step 4: 查询处理	@王之栋
step 5: 解析器	@王之栋
step 6: 系统管理	@孙迅
拓展功能	根据涉及模块分配

参考文献

参考了实验指导书和课件。

未参考借鉴任何开源代码。除 `antlr` 自动生成的代码外均为小组成员手写代码。