数据库系统概论实验总结报告

洪一宁 吴斯媛

2024 年 1 月 21 日

1 系统架构设计

系统由解析器、查询解析与系统管理模块、记录管理模块、索引模块、文件页管理器和文件管理器组成，其关系如图1所示。

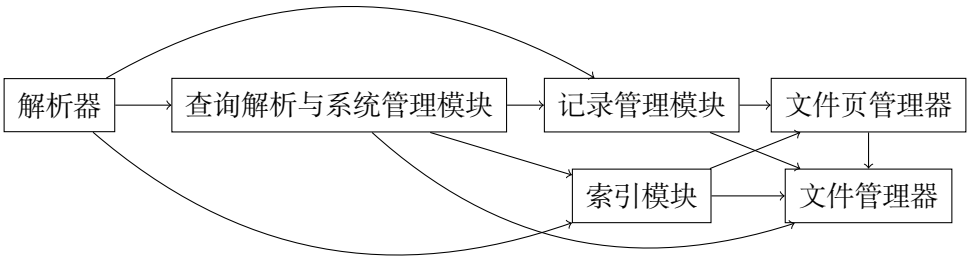

图 1: 系统架构设计图。

其中，文件管理器通过调用 c++ 的文件操作接口，可以对指定文件号和文件页进行直接读写，可以增删文件，可以创建文件夹，可以递归删除文件夹内内容，可以判断文件或文件夹是否存在。

文件页管理器通过调用文件管理器的相关接口，使用 LRU 替换页策略，对文件页的读写和缓存操作进行高效管理。

记录管理模块通过调用下层文件页管理器和文件管理器的相关接口，可以对指定记录文件进行初始化、插入记录、删除记录、更新记录、查询记录、和获取元数据等操作。

索引模块通过调用下层文件页管理器和文件管理器的相关接口，可以对指定索引文件进行初始化、插入索引、删除索引、和搜索索引等操作。

查询解析与系统管理模块被解析器调用，通过调用下层记录管理、索引和文件管理模块的相关接口，执行 DML 或 DDL 语句的具体操作。

解析器读取外部输入命令，进行词法、语法、语义分析，根据分析结果调用下层模块的相关接口进行相应处理，返回并输出结果。

2 各模块详细设计及主要接口说明

2.1 文件管理器

文件管理器参考实验文档附件中的页式文件系统参考实现，设计简单，仅负责执行文件相关的基础操作，如读写、创建删除文件等，其具体设计及接口如下所述。

`readPage(int fileID, int pageID, BufType buf, int off)`：指定文件号和页号，读取文件。通过计算读取位置的偏移量，调用`unistd`头文件中的`read`方法读取整页数据。整页数据使用一个长度为 2048 的`BufType`存储，其中`BufType`采用`unsigned int *`作为数据类型，每个元素可存储四字节数据。

`writePage(int fileID, int pageID, BufType buf, int off)`:指定文件号和页号，写入文件。其实现与`readPage`类似，使用`write`方法实现。

`openFile(const char* name)`：打开文件。通过函数 `open`实现。

`existFile(const char* name)`：判断文件是否存在。通过函数 `open`实现。

`createFile(const char* name)`：创建文件。通过函数`fopen`实现。

`closeFile(int fileID)`：关闭文件。通过函数`close`实现。

`createFolder(const char* path)`：创建文件夹。通过函数`mkdir`实现。

`existFolder(const char* path)`：判断文件夹是否存在。通过函数`stat`实现。

`deleteFolder(const char* path)`：删除文件夹及其所有内容。通过读取文件夹内容，递归调用函数，实现删除指定文件夹及其全部内容的功能。

2.2 文件页管理器

文件页管理器可以管理当前打开的文件页，使用 LRU 策略在文件页缓存不足时替换文件页。具体而言，该模块使用一个 `FindReplace` 模块用于实现 LRU 策略，其中 `FindReplace` 使用一个链表和动态数组存储链表节点从而实现 LRU 的 $O(1)$ 操作，使用一个位图 `BitMap` 用于标记脏页，使用一个动态数组 `BufType*` 存储当前打开所有页的地址，使用一个哈希表 `utils::HashMap<utils::HashItemTwoInt>` 以快速查找文件号和页号到打开页面编号的对应关系，使用一个动态数组 `PageLocation*` 用于查找打开页面编号到文件号和页号的对应关系。模块的具体接口如下所述：

`BufType getPage(int fileID, int pageID, int& index)`：获取页面。给定文件号和页号，读取并返回相应的页内容；若页不存在则创建该页；同时将页对应的编号写入 index。

`void access(int index)`：标记访问。给定页的编号，通过 `FindReplace replace`标记访问以实现 LRU 策略。

`void markDirty(int index)`：标记脏页。给定页的编号，使用位图标记脏页，并调用`access`。

`void close()`：关闭所有打开页。将当前所有打开的页面关闭，若是脏页则先写回。

2.3 记录管理模块

2.3.1 核心功能

记录管理模块可以对一个给定的记录文件进行操作，包括初始化文件、插入记录、删除记录、查询记录（包括分页查询、有限制的查询、查询内容输出到临时文件等），获取及修改列的元信息等。记录文件内部的存储均从为低位向高位。

记录文件的元信息包括文件的页数、记录的下一个插入的列的 ID，下一条插入的记录 ID 的值，每条记录 Null Bitmap 的长度，以及对应每一列的列信息。列信息包括列的 ID，列的数据类型（若为 varchar，还包括其最大长度），当前为 Varchar 分配的空间，列的名字，是否可以为 Null 值，是否有缺省值（若有，还包括缺省值的具体值），是否要求唯一。

记录文件的第 0 页记录了以上元信息，其按顺序的组织如下所述：

- 16B 位图，共 128 位，用于标记下方存储列元信息的槽是否被使用。
- 4B 该记录下一个要插入的列的 ID。
- 4B 文件的最大页号。
- 4B 该记录下一个要插入的记录的 ID。
- 4B 该记录文件每条记录 Null Bitmap 的大小。
- 80B $\times C$ ，每 80B 为一个槽，记录列的元信息。每个槽内部的组织如下所示：
  - 4B 列的 ID。
  - 1B 数据类型，INT 为 0，FLOAT 为 1，VARCHAR 为 2，DATE 为 3。DATE 在记录管理模块中全部实现完成，已预留接口，最终未实现该附加功能。
  - 1B 列的名字的长度。
  - 2B Varchar 类型的长度；若数据类型不是 Varchar，此位无意义。
  - 4B 为每条记录中 Varchar 预留的空间。初始时为两倍 Varchar 类型的长度。该接口为列的增删改操作预留，最终未实现该附加功能。
  - 32B 列的名字，按字节从低位至高位存储。
  - 2B 列是否要求非 NULL，是否有缺省值，缺省值是否为 NULL，列是否要求唯一，按 bit 从低位至高位存储；多余的 12 个 bit 预留，暂无用处。
  - 2B 若数据类型为 Varchar，且有非 NULL 缺省值，此处存储缺省值的长度，否则无意义。
  - 32B 存储缺省值。

记录文件从第一页开始，存储记录。每一页包括顶端 64B 的位图，共 512bit，表示下方的槽是否被使用。每个槽的组织如下所示：

- 4B 记录 ID，为该条记录永远不变且唯一的标识。
- 一个向 4B 对齐的 bitmap，表示记录中各数据是否为 null，其长度记录于表首页的元信息中。
- 每列的数据，int 占 4B，float 占 8B，date 占 4B （2B 年，1B 月，1B 日），varchar 占元信息中所述预留空间的大小。每列的长度向 4B 对齐。
- 对于每条记录，再在以上空间基础上增加一倍空间，方便列的增删改操作；最终未实现该功能。

记录管理模块包含以下主要接口（非全部借口）。以下接口，返回值为bool类型的，均表示操作成功（合法）与否，后续不再赘述。

初始化记录文件：

```
/**
 * @brief 创建一个记录文件，如果之前文件存在则清空并强行覆盖，
 * 写入文件元信息
 *
 * @param file_path 文件路径，需要保证文件所在的文件夹是存在的
 * @param column_types 所有列的类型
 */
void initializeRecordFile(const char* file_path,
                          const std::vector<ColumnType>& column_types)
```

列操作：

```
/**
 * @brief 更新列元信息是否要求unique
 *
 * @param file_path 文件路径
 * @param column_id 列id
 * @param unique 要更改到的值
 */
void updateColumnUnique(const char* file_path,
                        int column_id, bool unique);
```

```
/**
 * @brief 获取列的元信息
 *
 * @param file_path 文件路径
 * @param column_types 返回的vector, 存储所有列的类型,
 * 会清空vector原来的数据
 */
void getColumnTypes(const char* file_path,
                    std::vector<ColumnType>& column_types);
```

记录操作:

```
/**
 * @brief 插入一条记录, 会进行类型检查&varchar长度&null值检查,
 * 插入位置为从第一页开始找到的第一个空闲位置。
 *
 * @param file_path 文件路径
 * @param data_item 插入的数据
 * @return RecordLocation 插入的位置, (-1, -1)表示插入失败
 */
RecordLocation insertRecord(const char* file_path, DataItem data_item);

/**
 * @brief 给定记录位置, 删除一条记录
 *
 * @param file_path 文件路径
 * @param record_location 记录的位置
 */
bool deleteRecord(const char* file_path,
                  const RecordLocation& record_location);

/**
 * @brief 给定记录位置和更新值, 更新一条记录
 *
 * @param file_path  文件路径
 * @param record_location 记录的位置
 * @param data_item
 * 更改的数据, 不一定所有列都需要存在, 可通过DataItem的columnid指定列
```

```
 */
bool updateRecord(const char* file_path,
                  const RecordLocation& record_location,
                  const DataItem& data_item);
```

查询操作:

```
/**
 * @brief 指定记录位置, 获取记录数据
 *
 * @param file_path 文件路径
 * @param record_locations 要获取的记录的位置 (vector)
 * @param data_items 返回的数据, 会清空vector里原有的
 */
bool getRecords(const char* file_path,
                const std::vector<RecordLocation>& record_locations,
                std::vector<DataItem>& data_items);

/**
 * @brief 获取该记录文件中所有符合限制条件的记录
 * @param file_path 文件路径
 * @param data_items 返回的数据, 会清空vector里原有的
 * @param record_locations 返回的位置
 * @param constraints 约束条件
 */
void getAllRecordWithConstraint(
    const char* file_path, std::vector<DataItem>& data_items,
    std::vector<RecordLocation>& record_locations,
    const std::vector<system::SearchConstraint>& constraints);

/**
 * @brief 获取low_page到upper_page之间的所有记录, 页号左闭右开
 *
 * @param file_path 文件路径
 * @param data_items 返回的数据
 * @param low_page 起始页号
 * @param upper_page 终止页号
 */
```

```
void getRecordsInPageRange(const char* file_path,
                           std::vector<DataItem>& data_items,
                           int low_page, int upper_page);
```

2.3.2 记录管理缓存

为更加高效地处理多个记录文件，使短时间内重复访问不同的文件不会导致性能降低，模块记录了窗口为 10 的缓存，即最多支持同时打开 10 个记录文件。缓存的替换使用朴素的 FIFO 策略。

此外，同时打开的 10 个记录文件的列元信息也以类成员的形式存储，进一步提高模块性能。在修改列元信息时，清空暂存的信息。

2.4 索引模块

2.4.1 核心功能

索引模块可以对一个给定的索引文件进行操作，包括初始化文件、插入索引、删除索引、和查询索引（包括范围查询）。

索引文件的元信息包括该索引文件的 key 由几个 int 组成，以及索引文件 B+ 树的根节点为哪一页，这两个信息各占 4 字节，位于索引文件首页开头。

第二页开始为使用的页号的位图。前 8188 个字节，共 65504 bit 表示 bitmap，最后 4 字节表示使用页号位图的下一页的页号；若该页是链表的最后一个节点，则置-1。

对于树中的每个节点，也即除首页和使用页号位图之外的每一页，页开头的 16B 为页头，分别存储上一个同级节点的页号，下一个同级节点的页号，该节点存储多少个孩子，该节点是不是叶节点，各占 4 字节。页头之后按顺序存储该页的所有孩子。

假设该索引文件每个索引的键数量为 k，则树中的每个孩子占 $(k+2)\times 4$ B 空间，取 B+ 树的 M 值为 $M=(8192B-16B)/(4B\times(k+2))-1$。

对于叶节点，每个孩子首先存储 4B 的页号，然后存储 4B 的槽号，然后存储 $k\times$ 4B 的键值。

对于非叶节点，每个孩子首先存储 4B 的子节点页号，然后存储 $k\times$ 4B 的键值，代表子树中最大索引的键值。

索引模块包含以下主要接口（非全部接口）。以下接口，返回值为`bool`类型的，均表示操作成功（合法）与否，后续不再赘述。

初始化索引文件：

```
/**
 * @brief 创建一个索引文件，如果之前文件存在则清空并强行覆盖，
 * 写入文件元信息
```

```
 *
 * @param file_path 文件路径
 * @param index_key_num 索引文件的键有多少个
 */
void initializeIndexFile(const char* file_path, int index_key_num);
```

索引操作:

```
/**
 * @brief 插入一个索引
 *
 * @param file_path 文件路径
 * @param index_value 插入的索引信息
 */
bool insertIndex(const char* file_path, const IndexValue& index_value);

/**
 * @brief 删除一个索引，若有多个匹配的结果，则删除匹配上的第一个索引;
 * 若 exact match 为 false，则匹配时仅考虑键匹配，
 * 而不考虑页号、槽号匹配
 * 若 exact match 为 true，则匹配时不仅考虑键匹配，
 * 还要考虑页号、槽号匹配
 *
 * @param file_path 文件路径
 * @param index_value 删除的索引值
 * @param exact_match 是否要求完全匹配
 */
bool deleteIndex(const char* file_path, const IndexValue& index_value,
                 bool exact_match);
```

查询操作:

```
/**
 * @brief 查询键值介于 index_value_low 和 index_value_high 的所有索引
 * 查询范围左闭右闭
 *
 * @param file_path 文件路径
 * @param index_value_low 查询下界
 * @param index_value_high 查询上界
```

```
     * @param record_locations 查询结果
     */
    bool searchIndexInRanges(const char* file_path,
                             const IndexValue& search_value_low,
                             const IndexValue& search_value_high,
                             std::vector<IndexValue>& search_results);
```

2.4.2 索引缓存

为更加高效地处理多个索引文件，使短时间内重复访问不同的文件不会导致性能降低，模块记录了窗口为 10 的缓存，即最多支持同时打开 10 个索引文件。缓存的替换使用朴素的 FIFO 策略。

2.5 查询解析与系统管理模块

2.5.1 数据库数据文件组织

数据库文件存储于./data下，分为./data/global和./data/base两个文件夹。前者存储全局数据库信息，后者存储具体数据库信息。

./data/global下仅有一个文件，即./data/global/AllDatabase，是一个 record 类文件。该文件表头仅有一列，存储数据库的名字，其对应的记录 ID 就是数据库 ID。例如，文件内存有一条记录，ID 为 3，数据库名字为 “DB1”，就代表当前数据库管理系统仅一个名为 DB1 的 Database，且其 ID 为 3。

对应每一个数据库，./data/base均有一个文件夹./data/base/DB${DatabaseID}，每一个文件夹内存储该数据库的所有信息。

对于每一个形如./data/base/DB{DatabaseID}的文件夹，其内部有一个类似全局数据库记录文件的文件AllTable，记录数据库中所有数据表的名字。每条记录的记录 ID 就是这些数据表在该数据库内部唯一不变的 ID。在该路径下为每个表格建立一个文件夹，内部存储该表格的全部信息，文件夹路径形如./data/base/DB{DatabaseID}/TB{TableID}。

对于每一个形如./data/base/DB{DatabaseID}/TB{TableID}的文件夹，其内部有以下文件：

- Record：记录类文件，存储该表格的所有数据及列元信息（除主键、外键外的信息）。
- Primary Key：记录类文件，仅一列，代表主键的列编号，存储该表格的主键信息。
- Foreign Key：记录类文件，多列，存储外键中本表对应的列编号、外键引用的表 ID、外键引用的列 ID，和外键的名称。

- Dominate：记录类文件，仅一列，代表表格 ID，存储引用了本表作为外键的表格的编号，存储该信息方便增删改操作时的合法性判定。
- IndexInfo：记录类文件，多列，存储本表每个 index 分别将表中的哪些列按何种顺序作为索引文件的键，以及索引的名称。该表中每个记录的 ID 即为该数据库该表中索引唯一不变的编号。

在每个数据库中数据表的文件夹./data/base/DB{DatabaseID}/TB{TableID}下，还有一个文件夹夹./data/base/DB{DatabaseID}/TB{TableID}/IndexFiles，存储表格对应的所有索引文件。每一个索引根据其编号，命名为INDEX{IndexId}。

2.5.2 系统管理各功能接口

系统管理包含以下主要接口（非全部接口）。以下接口，返回值为bool类型的，均表示操作成功（合法）与否，后续不再赘述。

系统整体操作：

```
/**
 * @brief 初始化系统（若无数据，新建基本元数据；否则不改动）
 */
void initializeSystem();

/**
 * @brief 清空数据库
 */
void cleanSystem();

/**
 * @brief 设置使用的数据库ID
 * @param database_name 数据库名
 */
bool useDatabase(const char* database_name);
```

数据库操作：

```
/**
 * @brief 创建数据库
 *
 * @param database_name 数据库名字
 */
bool createDatabase(const char* database_name);
```

```
/**
 * @brief 删除数据库
 *
 * @param database_name 数据库名字
 */
bool dropDatabase(const char* database_name);
```

数据表操作:

```
/**
 * @brief 创建数据表
 *
 * @param table_name 数据表名字
 * @param column_types 数据表列类型
 * @param primary_keys 数据表主键对应列名称
 * @param foreign_keys 数据表外键对应列及数据表名称
 */
bool createTable(const char* table_name,
                 std::vector<record::ColumnType>& column_types,
                 const std::vector<std::string>& primary_keys,
                 const std::vector<ForeignKeyInputInfo>& foreign_keys);

/**
 * @brief 删除数据表
 *
 * @param table_name 数据表名称
 */
bool dropTable(const char* table_name);
```

数据表 schema 操作:

```
/**
 * @brief 添加主键
 *
 * @param table_id 数据表编号
 * @param column_ids 主键对应的列编号
 * @return
 */
bool addPrimaryKey(int table_id, std::vector<int> column_ids);
```

```
 /**
 * @brief 删除主键
 *
 * @param table_id 数据表编号
 */
bool dropPrimaryKey(int table_id);

/**
 * @brief 增加外键
 *
 * @param table_name 数据表名字
 * @param new_foreign_key 新外键信息, 含新外键名称
 */
bool addForeignKey(const char* table_name,
                   const ForeignKeyInfo& new_foreign_key);

/**
 * @brief 删除外键
 *
 * @param table_name 数据表名字
 * @param foreign_key_name 外键名字
 */
bool deleteForeignKey(const char* table_name,
                      std::string foreign_key_name);

/**
 * @brief 创建索引
 *
 * @param table_name 数据表名字
 * @param index_name 新索引名称
 * @param column_ids 新索引对应的列
 * @param check_unique 新索引是否要求插入的各元素唯一 (方便unique操作)
 */
bool addIndex(const char* table_name, const std::string& index_name,
              const std::vector<int>& column_ids, bool check_unique);


/**
```

```
 * @brief 删除索引
 *
 * @param table_name 数据表索引
 * @param index_name 要删除的索引名字
 */
bool dropIndex(const char* table_name, const std::string& index_name);

/**
 * @brief 增加unique约束
 *
 * @param table_name 数据表名字
 * @param unique_name unique约束名字
 * @param column_ids unique约束对应的列编号
 */
bool addUnique(const char* table_name, const std::string& unique_name,
               const std::vector<int>& column_ids);
```

系统信息查询:

```
/**
 * @brief 获取所有数据库名
 *
 * @param database_names 返回的结果
 * @param column_types 返回的表头
 */
void getAllDatabase(std::vector<record::DataItem>& database_names,
                    std::vector<record::ColumnType>& column_types);

/**
 * @brief 获取所有数据表名
 *
 * @param table_names 返回的结果
 * @param column_types 返回的表头
 */
bool getAllTable(std::vector<record::DataItem>& table_names,
                 std::vector<record::ColumnType>& column_types);

/**
```

```
 * @brief 获取描述一张数据表的信息
 *
 * @param table_name 数据表的名字
 * @param table_info 数据表各元信息查询的结果 (Field Type Null Default)
 * @param column_types 数据表各元信息查询的表头
 * (Field Type Null Default)
 * @param primary_keys 数据表主键信息
 * @param foreign_keys 数据表外键信息
 * @param index 数据表索引信息
 * @param unique 数据表unique信息
 */
bool describeTable(const char* table_name,
                   std::vector<record::DataItem>& table_info,
                   std::vector<record::ColumnType>& column_types,
                   std::vector<std::string>& primary_keys,
                   std::vector<ForeignKeyInputInfo>& foreign_keys,
                   std::vector<std::vector<std::string>>& index,
                   std::vector<std::vector<std::string>>& unique);
```

2.5.3 查询解析各功能接口

数据库内容插入操作:

```
/**
 * @brief 向数据表中插入新纪录, 检查主键、外键、null、unique约束,
 * 同时按需求更新索引
 *
 * @param table_name 数据表名字
 * @param data_items 插入的数据
 */
bool insertIntoTable(const char* table_name,
                     std::vector<record::DataItem>& data_items);

/**
 * @brief 从CSV文件中加载数据到数据表中, 更新索引
 *
 * @param table_name 数据表名字
 * @param file_path CSV文件路径
```

```
 * @param delimeter CSV文件分隔符
 */
bool loadTableFromFile(const char* table_name, const char* file_path,
                       const char* delimeter);
```

数据库内容删除操作:

```
/**
 * @brief 从数据表中删除记录，检查其他表引用本表外键的合法性，更新索引
 *
 * @param table_id 被删除表的编号
 * @param constraints 被删除数据的约束
 * @return
 */
bool deleteFromTable(int table_id,
                     std::vector<SearchConstraint>& constraints);
```

数据库内容更新操作:

```
/**
 * @brief 更新数据表中的记录，检查其他表引用本表外键的合法性，
 * 检查主键、外键、null、unique约束，更新索引
 *
 * @param table_id 数据表编号
 * @param constraints 更新数据的约束
 * @param update_data 更新的数据
 */
bool update(int table_id, std::vector<SearchConstraint>& constraints,
            const record::DataItem& update_data);
```

数据库内容查询操作:

```
/**
 * @brief 查询数据表中的记录，返回符合条件的记录
 *
 * @param table_id 数据表编号
 * @param constraints 查询条件
 * @param result_datas 查询结果
 * @param column_types 查询结果的表头
```

```
     * @param record_location_results 查询结果的记录位置
     * @param sort_by 接口遗留，暂未使用
     */
    bool search(
        int table_id, std::vector<SearchConstraint>& constraints,
        std::vector<record::DataItem>& result_datas,
        std::vector<record::ColumnType>& column_types,
        std::vector<record::RecordLocation>& record_location_results,
        int sort_by);

    /**
     * @brief 查询数据表中的记录，将符合条件的记录保存到文件中，
     * 记录符合条件的记录数
     *
     * @param table_id 数据表编号
     * @param column_types 查询结果的表头
     * @param constraints 查询条件
     * @param save_path 保存路径
     * @param total_num 符合条件的记录数
     */
    bool searchAndSave(int table_id,
                       std::vector<record::ColumnType>& column_types,
                       std::vector<SearchConstraint>& constraints,
                       std::string& save_path, int& total_num);
```

2.5.4 查询解析重要算法

查询优化算法

首先，优化查询限制条件：

1. 合并对于同一列的限制，使用 vector 存储对每一列的条件。
2. 将所有“等于”限制修改为大于等于且小于等于。
3. 对于每一列，若有多个范围限制，则尽可能合并；对于 int 类，仅保留大于等于、小于等于两种范围限制。

然后，根据优化后的查询限制条件，查询能够对当前有范围限制的列产生最多限制的索引。

若存在这样的索引，则查询索引，获取一系列可能结果在记录文件中的位置。获取记录文件中这些位置对应的记录，一边获取记录，一边根据优化后的查询条件进行筛选。

若不存在这样的索引，则直接查询记录文件中的每一条记录，即全表搜索，一边搜索，一边根据优化后的查询条件进行筛选。

多表 Join 优化算法

对于多表 join，首先将限制区分为表和表之间的 join 限制，和对于单表查询的限制。

然后，根据每个涉及到的表格的单表查询限制，进行记录查询，将查询结果存储于临时文件中，记录下查询到的记录数量。

从查询得到数量最少的表开始 join 操作。每次，寻找一个尚未 join 的表，和现有已 join 好的部分进行合并。寻找尚未合并表的依据是优先考虑与已 join 表有最多合并限制条件的表；若 join 限制的数量相同，则考虑之前单表查询到的记录条数最少的表。

每次 join，首先依据第一个与待 join 表相关的限制条件对已 join 表进行排序；然后，以 8 页为单位，分次读取之前单表查询得到的待 join 表的查询结果；每得到 8 页的记录，对这些记录进行相应的排序，然后双指针扫描，将符合所有限制条件的结果进行合并。直至读取完此前单表查询得到的所有结果，合并完成为止。

重复以上过程，直至所有表格被合并完成。

2.6 命令解析器

以 antlr-4 生成的编译器基本代码为基础，实现了 SQLMyVisitor 类，实现了基本的增删查改、修改主外键索引、数据表信息显示功能，并且支持 NULL 值，UNIQUE 约束，以及查询操作中的 ORDER BY - LIMIT - OFFSET 语法。

2.6.1 增删查改操作主要接口

```
visitCreate_table(
     antlr4::SQLParser::Create_tableContext* ctx)
visitDelete_from_table(
     antlr4::SQLParser::Delete_from_tableContext* ctx)
visitSelect_table(
     antlr4::SQLParser::Select_tableContext* ctx)
visitUpdate_table(
     antlr4::SQLParser::Update_tableContext* ctx)
```

2.6.2 数据表描述操作主要接口

```
visitDescribe_table(
     antlr4::SQLParser::Describe_tableContext* ctx)
```

2.6.3 修改主外键及索引操作主要接口

```
visitAlter_add_index(
     antlr4::SQLParser::Alter_add_indexContext* ctx)
visitAlter_drop_index(
     antlr4::SQLParser::Alter_drop_indexContext* ctx)
visitAlter_table_add_pk(
     antlr4::SQLParser::Alter_table_add_pkContext* ctx)
visitAlter_table_drop_pk(
     antlr4::SQLParser::Alter_table_drop_pkContext* ctx)
visitAlter_table_add_foreign_key(
     antlr4::SQLParser::Alter_table_add_foreign_keyContext* ctx)
visitAlter_table_drop_foreign_key(
     antlr4::SQLParser::Alter_table_drop_foreign_keyContext* ctx)
```

2.6.4 修改 UNIQUE 约束操作接口

```
visitAlter_table_add_unique(
     antlr4::SQLParser::Alter_table_add_uniqueContext *ctx)
```

2.6.5 查询条件与约束

对于查询语句的条件，通过 Condition 类表示查询条件，再转换为 SearchConstraint 类表示文件管理模块实现具体的查询操作时使用的查询约束，格式如下。

```
// 表示约束的比较类型
enum class Operator { EQ, NEQ, LT, LEQ, GT, GEQ, IS, UNDEFINED };
// 表示约束的数据类型
enum class VariableType {
    INT, FLOAT, STRING, NULL_OR_NOT, TABLE_COLUMN
};

struct Condition {
    std::string table_name, column_name;  // 被选择的表名与列名
    Operator op;                          // 约束的比较类型
    VariableType type;                    // 约束的数据类型

    // 以下值仅在约束的数据类型为该类型时有效
```

```
    // e.g. WHERE a < 1; 则 int_val 有效。
    int int_val;
    double float_val;
    std::string string_val;
    std::string table_name_other, column_name_other;
    bool is_null;
};

struct SearchConstraint {
    // 是对哪一列的约束
    int column_id;
    // 约束的数据类型 (也就是对应列的数据类型)
    record::DataTypeName data_type;
    // 对应约束的比较类型 EQ, NEQ, GT, GEQ, LT, LEQ
    std::vector<ConstraintType> constraint_types;
    // 对应约束的值
    std::vector<record::DataValue> constraint_values;
};
```

2.6.6 前端输出显示主要接口

通过 ParseResult 类的 `void print(bool output_mode, int duration)` 接口进行输出。其中 `output_mode` 可以调整输出格式，以适应批量测试模式和用户交互模式，后者的可读性更强，并且支持查询用时显示，如图2。

3 实验结果

完成了所有必做内容，并且完成了以下选做实验：多表 JOIN、高级查询-排序分页 ORDER LIMIT OFFSET、高级索引-UNIQUE 约束、高级完整性约束-NULL。并支持批量测试模式和用户交互模式的输出。

4 小组分工

- 吴斯媛：编译器基本代码生成、命令解析器模块、用于查询操作的 Condition 模块。
- 洪一宁：文件管理器模块、文件页管理器模块、记录管理模块、索引模块、和查询解析与系统管理模块。

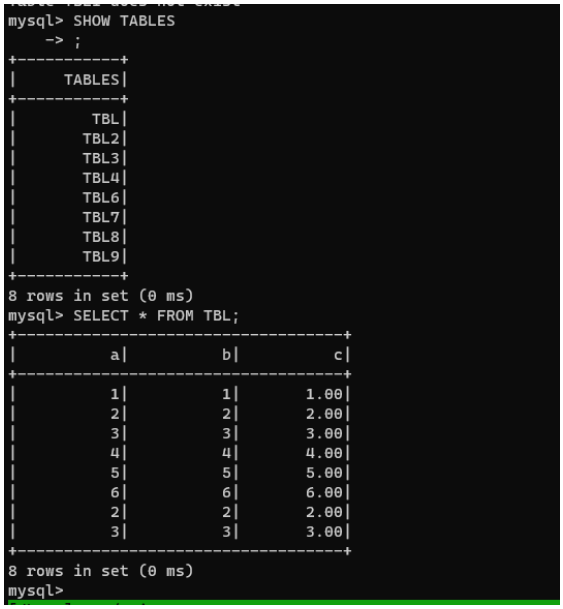

图 2: UI 效果图

5 参考文献、使用工具及第三方库

- 参考了数据库系统概论实验文档及实验文档附件提供的页式文件系统参考实现。
- 参考了数据库系统概论课件。
- 编译器基本代码生成工具：`https://github.com/antlr/antlr4/`
- 使用 GoogleTest 对项目进行单元测试。