

# 《数据库系统概论》项目总结报告

计02 刘明道 2020011156  
计04 高焕昂 2020010951

Course project for *Introduction to Database Management System (2022 Fall)*.

## 使用说明

### 运行环境

- Linux
- C++ 20
- CMake >= 3.22

### 运行步骤

- 检查 `./utils/setup_env.sh`，安装相应依赖
- 使用以下命令构建项目

```
pip install -r requirements.txt
conan profile new default --detect
conan profile update settings.compiler.libcxx=libstdc++11 default

cd build
conan install ..
cmake ..
make front -j$(nproc)
bin/front
```

## 系统架构设计

数据库系统的主要功能模块包括

模块	功能
io	页式文件系统，缓存系统
index	索引系统
record	记录管理：字段，记录，数据页，约束
system	查询解析和执行
node	查询计划树节点

## 各模块详细设计

## 文件管理

和文件管理系统有关的代码主要位于 `src/io` 文件夹下。

- `FileSystem` 类是一个页式文件管理系统，对 Linux 系统调用和 `std::filesystem` 提供的文件输入输出函数进行了一次封装，对外提供了按页为粒度对文件进行读写的接口。
- `Page` 类表示实际的一页数据，提供了额外的标志位标志其是否为脏页，同时提供了 `Lock()` 函数和 `Release()` 函数为上层缓存系统在换页时标记本页是否可以被交换。
- `BufferSystem` 类是一个基于 LRU 算法的缓存系统，这也是数据库系统进行文件读写的接口。缓存系统将部分页缓存在内存中，并使用 LRU 替换算法在适当的时机调用文件管理系统提供的函数将被替换页写入硬盘。

## 记录管理

和记录管理系统有关的代码主要位于 `src/record` 文件夹下。

- `DataPage` 是对 `Page` 类的一次封装，可以同时支持定长记录和变长记录，并对外开放了查看剩余空间和根据槽号插入记录、更新记录与删除记录的接口。删除后，我们默认将尾指针置为 -1。其页面布局遵循以下规则

```
/** DataPage Layout
 * | PageHeader |
 * | Slot 0 | Slot 1 | ... | Slot <slot_count - 1> |
 * | Free Space | ...
 * | Pointer to free space | Pointer to <Slot <slot_count - 1>> | ... | Pointer to <Slot 0> |
 */
```

- `FreeSpaceManager` 类将一张表的各个数据页面的剩余空间组织起来，在面对新记录作为输入，需要申请分配一定的空间的时候，可以将具有对应空间的页面快速地查询出来。具体来说，我们采用了文档中提出的链表实现，将 4096 Byte 的页面划分成 256 个不同的状态，然后使用 (页号 → 状态)，(状态 → 页号列表) 两种映射来辅助查询与修改工作。
- `Field` 类定义了表列的类别，包括 `Int`, `Float`, `Char`, `VarChar` 与 `Date` 这几种。同时，其中定义的 `PrimaryKey`, `ForeignKey`, `UniqueKey`, `IndexKey` 类分别定义了主键、外键、唯一性约束与索引的结构。
- `Record` 类是对一行记录的定义，其由 `Field` 的向量组成。在序列化与反序列化时，我们使用 `NullBitmap` 来记录某一列是否为空。
- `TableMeta` 类是对一张表元信息的表述，其 Layout 如下所示：

```
/**
 * TableMeta Layout
 * Page 0: Fixed info (ensured that will not exceed one page)
 * Page 1 ~ Page n: Foreign Keys (each foreign key has fixed length)
 * Page (n + 1) ~ Page m: Index Keys
 * Page (m + 1) ~ Page k: Field Metas
 * Page (k + 1) ~ Page ([total_page_count] - 1): Free Space Records
 */
```

## 索引管理

和索引管理系统有关的代码主要位于 `src/index` 文件夹下。索引管理系统和记录管理系统很相似，下面做具体介绍：

- `IndexPage` 类是对 `Page` 类的一次封装。注意到内部节点页和叶节点页对应的布局结构不同。对外其表现类似于一个向量，提供了根据槽号拿出记录，插入记录，更新记录，删除记录，切片选取记录，范围插入与范围删除记录的接口。其页面布局遵循以下规则

```
/*
** Index Page Layout (Leaf)
```

```

* | PageHeader |
* | IndexRecord <0> | IndexRecord <1> | ... | IndexRecord <n-1> |
* where IndexRecord <i> == (PageID <i>, SlotID <i>, Key <i>)
*
* Index Page Layout (Internal)
* | PageHeader |
* | IndexRecord <0> | IndexRecord <1> | ... | IndexRecord <n-1> |
* where IndexRecord <i> == (PageID <i>, Key <i>)
*
* For every internal node <j>, we ensure that all elements in the subtree rooted at <j>
* are less than or equal to Key <j>.
*
* The methods of IndexPage act like a vector. ()
*/

```

- `IndexField` 类是对可以被索引的字段相应的索引记录的描述，目前支持 `IndexINT`、`IndexINT2` 两种类型，分别对应记录类型的 `Int` 型与两列 `Int` 型的联合索引。其为内部节点与叶结点提供了两套 `FromSrc` 方法与 `Write` 方法，这是因为在联合索引的实现中，内部节点只需存储第一个列的关键码，而叶结点需要以第一列为主排序条件，第二列为次排序条件进行排序。
- `IndexFile` 类整体将 `IndexPage` 类的对象作为结点，将其组织成一颗 B+ 树的形式。对外其提供了索引查找的接口，以 (叶结点页号, 槽号) 的二元组作为迭代器提供沿下侧链遍历所有索引叶结点页的方法。具体来说，其包括记录的读取、插入、删除、范围删除与更新。在内部节点间查询时我们使用二分查找的方法。
- `IndexRecord` 类是索引记录项，同样分为针对内部节点的索引记录和针对叶结点的索引记录。其提供了从 `Record` 类对象转换的方法。
- `IndexMeta` 类是对某个索引系统的元信息的定义，包括 B+ 树的阶数，目前索引系统的页数，根页面页号，索引类型等等。这里我们在创建索引系统时，默认采用的阶数为最大程度利用叶结点页空间的阶数值。

## 查询处理

和查询处理系统有关的代码主要位于 `src/node` 文件夹下与 `src/system/whereConditions.h` 中，我们构造的查询计划树的可能结点类型如下：

- `TrivialScanNode` 和 `IndexScanNode` 两种结点用于扫描任务，其中前者按照数据页的存储顺序进行扫描筛选，后者按照索引进行扫描筛选并在条件合适时提前终止。
- `NestedJoinNode` 和 `HashJoinNode` 两种表合并结点，分别对应文档中提到的嵌套循环连接与哈希连接两种表连接方法。
- `ProjectNode` 计算结点用于对选取出的列进行投影。
- `AggregateNode` 计算结点用于实现查询算法的聚合查询、分组查询功能。
- `OffsetLimitNode` 计算结点可以实现查询算法的 OFFSET 与 LIMIT 功能。
- `LikeCondition` 用于实现模糊查询。其将输入先进行正则表达式的 escape，然后将输入中的 `_` 和 `%` 转换为 `.` 和 `.*` 接下来使用正则表达式引擎进行匹配。
- `InSubqueryCondition` 和 `CompareSubQueryCondition` 分别用于 `IN` 和 `=` 引导的嵌套查询。在构造时，其先执行嵌套的查询计划树，然后判断结果的合法性，若不合法则给出报错。

## 解析器

语法解析使用实验框架提供的默认语法文件 `src/grammar/SQL.g4`，使用 `antlr` 库进行解析，之后生成具有 Visitor 模式的文件。我们继承了自动生成的 Visitor 方法，编写了 `src/system/DBVisitor.h`，以实现语法解析的功能。

绝大部分的解析工作是平凡的，主要是从抽象语法树中将相应的字符串提取出来后，进行如类型检查等初步的合法性检查，然后提取出执行操作需要的所有信息。

理论上，解析器是将 SQL 语句的抽象语法树转换成执行计划树再交给执行器进行执行。考虑到 `SELECT` 之外的语句执行较为简单，我们直接将相应信息交给 `DBSystem` 对象执行，并将结果返回而省略了转化为执行计划树这一步。

对 `SELECT` 语句，我们则构建相应执行计划树：

- 对于每个查询，先访问 `WHERE` 从句获得过滤条件和连接条件
- 根据对每张表的过滤条件和表的索引情况选择是否使用索引，构建每张表的**扫描节点**
- 构造**连接节点**，根据连接条件选择使用嵌套连接或哈希连接
- 根据是否使用聚合算子以及 `GROUP BY` 语句选择构造**聚合节点**或**投影节点**
- 根据是否使用 `LIMIT OFFSET` 语句选择是否添加 **LimitOffset 节点**

然后将执行计划树交给 `DBSystem` 进行执行操作，再将结果返回

## 系统管理

我们系统管理的有关代码位于 `src/system/DBSystem.h` 与 `src/system/DBSystem.cpp`，这是程序逻辑的主要部分。系统管理支持创建、删除、调整当前正在使用的数据库和数据表的相关接口，同时对解析器解析出的其他命令进行了简单的一步封装。其具体功能和相应的接口见下一节的介绍。

## 主要接口说明

### 文件系统 FileSystem

```
// 创建目录，第二个参数表示是允许目录在创建前就存在
static void MakeDirectory(const std::string &, bool = true);

// 删除目录
static void RemoveDirectory(const std::string &);

// 创建文件
static FileID NewFile(const std::string &);

// 删除文件
static void RemoveFile(const std::string &);

// 打开文件
static FileID OpenFile(const std::string &);

// 关闭文件
static void CloseFile(FileID fd);

// 整页写入
static void WritePage(FileID fd, PageID page_id, uint8_t *src);

// 整页读出
static void ReadPage(FileID fd, PageID page_id, uint8_t *dst);
```

### 缓存系统 BufferSystem

```
// 读取页面
Page *ReadPage(FileID fd, PageID page_id);

// 创建页面
Page *CreatePage(FileID fd, PageID page_id);
```

```
// 将页面写回
void WriteBack(Page *page);

// 标记页面访问
void Access(Page *page);

// 写回 fd 的所有脏页
void ReleaseFile(FileID fd);

// 写回 fd 的所有脏页并关闭 fd
void CloseFile(FileID fd);
```

由于 Page 有统一的创建和回收点，所以这里直接使用裸指针。

## 查询处理系统 DBSystem

查询处理系统的对外接口供 DBVisitor 解析并执行语句时调用。其返回一个 `Result` 的指针，表示执行的结果（包括文本型结果和表格型结果）。如果执行出错，如不满足约束条件等，系统将会抛出相应的异常。

- 数据库的增加，使用，删除和列举

```
std::shared_ptr<Result> CreateDatabase(const std::string &db_name);
std::shared_ptr<Result> UseDatabase(const std::string &db_name);
std::shared_ptr<Result> DropDatabase(const std::string &db_name);
std::shared_ptr<Result> ShowDatabases() const;
```

- 数据表的增加，删除，列举与显示

```
std::shared_ptr<Result> CreateTable(
    const std::string &table_name,
    const std::vector<std::shared_ptr<FieldMeta>> &field_meta,
    std::optional<RawPrimaryKey> raw_pk,
    const std::vector<RawForeignKey> &raw_fks
);
std::shared_ptr<Result> DropTable(const std::string &table_name);
std::shared_ptr<Result> ShowTables() const;
std::shared_ptr<Result> DescribeTable(const std::string &table_name);
```

- 约束的增加与删除

```
std::shared_ptr<Result> AddForeignKey(
    const std::string &table_name,
    const RawForeignKey &raw_fk
);
std::shared_ptr<Result> AddPrimaryKey(
    const std::string &table_name,
    const RawPrimaryKey &raw_pk
);
std::shared_ptr<Result> AddUnique(
    const std::string &table_name,
    const std::vector<std::string> &fields
);
std::shared_ptr<Result> DropForeignKey(
```

```

    const std::string &table_name,
    const std::string &fk_name
);
std::shared_ptr<Result> DropPrimaryKey(
    const std::string &table_name,
    const std::string &pk_name
);

```

- 索引的增加与删除

```

std::shared_ptr<Result> AddIndex(
    const std::string &table_name,
    const std::vector<std::string> &field_name
);
std::shared_ptr<Result> DropIndex(
    const std::string &table_name,
    const std::vector<std::string> &field_name
);

```

- 记录的增删查改

```

std::shared_ptr<Result> Insert(
    TableID table_id,
    RecordList &records
);
std::shared_ptr<Result> Delete(
    TableID table_id,
    const std::shared_ptr<AndCondition> &cond
);
std::shared_ptr<Result> Select(
    const std::vector<std::string> &header,
    const std::shared_ptr<OpNode> &plan
);
std::shared_ptr<Result> Update(
    TableID table_id,
    const std::vector<std::pair<std::shared_ptr<FieldMeta>,
    std::shared_ptr<Field>>> &updates,
    const std::shared_ptr<AndCondition> &cond
);

```

系统也提供了一些信息查询接口供 `DBVisitor` 使用，包括

- 表 ID 和元信息的查询

```

// 获取表 ID
TableID GetTableID(const std::string &table_name) const;

// 获取表的元信息
std::shared_ptr<const TableMeta> GetTableMeta(TableID table_id);

```

- 页扫描节点的生成。

```

// 获取页的遍历扫描节点
std::shared_ptr<ScanNode> GetTrivialScanNode(

```

```

    TableID table_id,
    const std::shared_ptr<FilterCondition> &cond
);

// 获取索引扫描节点
std::shared_ptr<ScanNode> GetIndexScanNode(
    TableID table_id,
    std::vector<FieldID> field_ids,
    const std::shared_ptr<FilterCondition> &cond,
    const std::shared_ptr<IndexField> &key_start,
    const std::shared_ptr<IndexField> &key_end
);

// 以及根据 `WHERE` 条件和索引情况选择扫描节点
std::shared_ptr<ScanNode> GetScanNodeByCondition(
    TableID table_id,
    const std::shared_ptr<AndCondition> &cond
);

```

此外还有一些工具函数

- 约束检查，用于建立约束或增删改记录时使用

```

// 检查元组的唯一性（遍历，用于建立约束时的检测），将找到的重复记录返回
std::optional<std::shared_ptr<Record>> CheckRecordsUnique(
    TableID table_id,
    const std::vector<FieldID> &fields
);

// 检查元组的唯一性（索引加速，用于已经建立的约束），将找到重复记录的条数返回
std::size_t TupleExists(
    TableID table_id,
    const std::vector<FieldID> &fields,
    const std::vector<std::shared_ptr<Field>> &field_vals
);

// 插入时检查约束
void CheckConstraintInsert(
    const TableMeta &meta,
    const std::shared_ptr<Record> &record
);

// 删除时检查约束
void CheckConstraintDelete(
    const TableMeta &meta,
    const std::shared_ptr<Record> &record
);

// 更新时检查约束
void CheckConstraintUpdate(
    const TableMeta &meta,
    const std::shared_ptr<Record> &record_prev,
    const std::shared_ptr<Record> &record_updated,
    const std::unordered_set<FieldID> &affected
);

```

- 索引维护

- 我们只在 B 树中支持 2 列联合索引，而对于 2 列以上的主键 / 外键 / 唯一约束，我们实现为自动对前 2 列创建索引，对索引给出的结果遍历查找。
- 一个索引可能被多个约束使用。为节约空间，我们实现为对索引引用计数，当索引被引用次数等于 0 时将索引删除。

```
// 添加索引（只看前 lim 个字段）
template<FieldID lim>
void AddIndexLimited(
    TableID table_id,
    const std::vector<FieldID> &field_ids,
    bool is_user
);

// 删除索引（只看前 lim 个字段）
template<FieldID lim>
void DropIndexLimited(
    TableID table_id,
    const std::vector<FieldID> &field_ids,
    bool is_user
);

// 更新 原地更新的记录 的索引
void UpdateInPlaceRecordIndex(
    const std::unordered_set<FieldID> &affected,
    TableID table_id, PageID page_id, SlotID j,
    const std::shared_ptr<Record> &record_prev,
    const std::shared_ptr<Record> &record_updated
);

// 插入新索引记录
void InsertRecordIndex(
    TableID table_id,
    PageID page_id, SlotID j,
    const std::shared_ptr<Record> record
);

// 删除索引记录
void DropRecordIndex(
    TableID table_id,
    PageID page_id, SlotID j,
    const std::shared_ptr<Record> record
);

// 获取索引文件
std::shared_ptr<IndexFile> &GetIndexFile(
    TableID table_id,
    FieldID field_id1,
    FieldID field_id2
);
```

- 还有非常简单的封装以及一些同类函数的重载，用于不同情境，这里就不主一列举了



## 字段 Field

我们定义了 5 种数据类型，他们的继承关系如下

- 抽象类 Field
  - 整型 Int
  - 浮点型 Float
  - 日期型 Date
  - 字符串型 String
    - 定长字符串 Char
    - 变长字符串 VarChar

Field 的接口列举如下

```
// 获取字段的占用的存储空间
virtual RecordSize Size() const = 0;

// 将记录序列化到 dst 指向的地址，并将 dst 偏移相应长度
virtual void Write(uint8_t *&dst) const = 0;

// 获取字段的哈希值，用于 Hash Join
virtual std::size_t Hash() const = 0;

// 将字段输出为文本
virtual std::string ToString() const = 0;

// 字段大小比较: NULL 与任何值没有偏序关系，非 NULL 只能进行同类比较，否则抛出异常
virtual std::partial_ordering operator<=>(const Field &rhs) const = 0;

// 字段判等: NULL 与任何值不等，非 NULL 只能进行同类比较，否则抛出异常
virtual bool operator==(const Field &rhs) const = 0;

// 从 src 根据类型加载字段
static std::shared_ptr<Field> LoadField(
    FieldType type,
    const uint8_t *&src,
    RecordSize max_len = -1
);

// 生成特定字段类型的 NULL 值
static std::shared_ptr<Field> MakeNull(FieldType type, RecordSize max_len = -1);
```

## 记录 Record

```
// 从 src 按照表元 meta 的指导读取一条记录，并偏移 src
static std::shared_ptr<Record> FromSrc(const uint8_t *&src, const TableMeta &meta);

// 将记录写入 dst 所指向的位置，并偏移 dst
void Write(uint8_t *&dst);

// 获取记录占用的空间
RecordSize Size();
```

```

// 将记录的全部字段逐一表示为字符串，用于查询结果输出
std::vector<std::string> ToString();

// 将记录表示为单个字符串，用于调试和报错
std::string Repr();

// 复制记录
std::shared_ptr<Record> Copy() const;

// 更新记录中的特定字段
void Update(
    const std::vector<
        std::pair<std::shared_ptr<FieldMeta>, std::shared_ptr<Field>>,
        > &updates
);

// 将记录进行投影
std::vector<std::shared_ptr<Field>> Project(const std::vector<FieldID> target) const;

// 判断记录中是否存在空值
static bool HasNull(const std::vector<std::shared_ptr<Field>>& target_reference);

// 记录判等
bool operator == (const Record& rhs) const;

```

## 数据页 DataPage

```

// 读取 slot 中存放的记录
std::shared_ptr<Record> Select(slotID slot) const;

// 插入一条新记录
slotID Insert(std::shared_ptr<Record> record);

// 删除 slot 中存放的记录
void Delete(slotID slot);

// 更新 slot 中存放的记录
void Update(slotID slot, std::shared_ptr<Record> &record);

// 初始化页面结构
void Init();

```

## 查询计划树节点 OpNode

查询计划的执行采取流水线策略。每个节点的 `over()` 方法表示该节点是否还有输出，而 `Next()` 则给出一部分输出。父级节点可以通过反复调用 `Next()` 方法逐步获取节点的输出，或者使用 `All()` 方法一次获取全部输出。

```
// 节点是否已经遍历完成
virtual bool Over() const;

// 将节点恢复初始状态
virtual void Reset();

// 获取部分输出
virtual RecordList Next();

// 获取全部输出
virtual RecordList All();
```

## 实验结果

对于数据库的后端，我们将其编译为

- library `main_shared`：数据库逻辑的动态链接库，供其他目标使用。

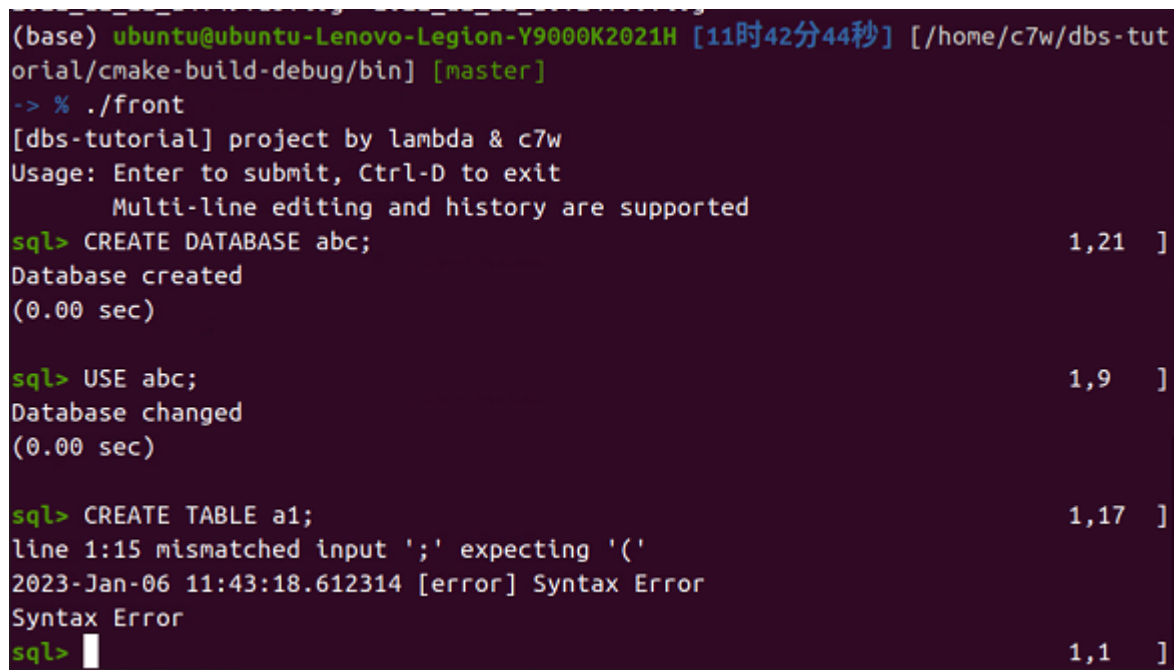
我们支持 3 种前端

- library `connect_db`：在 Python 中调用数据库系统并返回执行结果，例如：

```
import connect_db
results: list = connect_db.query("SELECT * FROM T;");
```

用于对拍测试。

- executable `front`：用户友好的交互界面，使用 `cpp-terminal` 库，支持历史记录回溯，查看当前行号列号，跨列编辑等功能。用户可以在这里输入支持的 SQL 语句，然后交由 `DBvisitor` 中的对应函数进行解析。`front` 会捕获所有异常，并给出提示。



```
(base) ubuntu@ubuntu-Lenovo-Legion-Y9000K2021H [11时42分44秒] [/home/c7w/dbs-tutorial/cmake-build-debug/bin] [master]
-> % ./front
[dbs-tutorial] project by lambda & c7w
Usage: Enter to submit, Ctrl-D to exit
       Multi-line editing and history are supported
sql> CREATE DATABASE abc;                                1,21  ]
Database created
(0.00 sec)

sql> USE abc;                                            1,9   ]
Database changed
(0.00 sec)

sql> CREATE TABLE a1;                                  1,17  ]
line 1:15 mismatched input ';' expecting '('
2023-Jan-06 11:43:18.612314 [error] Syntax Error
Syntax Error
sql>                                                    1,1   ]
```

- executable `debug`：从 `stdout` 中读取命令并执行，用于批量执行语句，如导入数据或者调试测试。

# 验收测试

我们将验收测试的 CSV 文件转换成 SQL 语句执行导入数据库。在验收中我们成功通过了以下功能测试

- **基本功能**：基本运行、系统管理、查询解析、完整性约束、模式管理、索引模块
- **扩展功能**：多表 JOIN、5 种高级查询、扩展数据类型 DATE、高级索引：UNIQUE 约束、高级完整性约束：NULL、联合主外键、联合索引

# 对拍测试

我们编写了若干测试程序位于 `tests` 文件夹下。其中在 `tests/python/` 下我们可以创建 Python 文件生成相应的测例 SQL；在 `tests/python_utils` 下我们提供了连接 MySQL 服务器的 Python 客户端，方便与我们数据库的输出结果进行对拍；在 `tests/sql` 文件夹下是我们生成的测例 SQL 语句代码。

我们对记录增删查改，连接，LIMIT OFFSET，聚合等涉及大量数据的操作，与 MySQL 进行对拍以验证结果的正确性。

# 其它

## 成员分工

- 刘明道：系统与文件管理、记录管理、查询解析
- 高焕昂：索引管理、记录管理

## 参考资料

- [实验文档](#)
- 我们使用了如下 C++ 开源库

```
antlr4-cppruntime/4.9.3
argparse/2.9
boost/1.80.0
cpp-terminal/17cd990
fmt/9.1.0
gtest/1.10.0
magic_enum/0.8.1
tabulate/1.4
```

它们以 git submodule 引用或使用 conan 进行管理，详见 `/.gitmodules` 和 `/conanfile.txt`。引用的外部库和自动生成的代码没有包含在项目文件中。

- 查询计划树的接口参考了 2022 春《数据库专题训练》的实验设计。
- 除上述资料外，未参考任何其他资料或开源代码。