《数据库系统概论》课程项目实验报告

邢竞择 2020012890

2024 年 1 月 19 日

1 整体设计

1.1 基本设计

项目使用 C++ 编写，依赖于 antlr4-runtime 和 antlr4-cpp 这两个外部库进行语法解析，使用 argparse 解析命令行参数，此外还使用 googletest 编写了 B+ 树的功能测试。

1.2 模块设计

模块设计基本上参照实验指导书，分为文件管理模块、记录管理模块、索引管理模块、SQL 解析模块和查询处理模块。文件管理模块的，SQL 解析模块的实现在 `src/frontend` 目录下，其余三个模块的实现均在 `src/engine` 目录下。

1.3 功能实现

除了全部基本功能外，本项目还实现了具有 CI 测例的全部附加功能：

- 多表 join
- 聚合查询、模糊查询、分组查询、嵌套查询、查询结果排序
- 日期、完整 null 支持
- unique 约束

2 具体设计

2.1 工具模块

该模块的代码实现在 `src/utils` 目录下。为了方便参数管理，我实现了一个单例模式的 `Config` 类用于管理全部配置，它负责解析命令行参数并提供给其他模块使用。

为了方便统一数据库的输出，我实现了一系列 `tabulate` 函数，能够按照当前交互模式（交互或者批处理）来输出查询结果。

此外，这部分还提供诸如 `ensure_file` 和 `ensure_directory` 的函数，方便在其他模块中检查文件和目录，提升代码健壮性。

2.2 文件管理模块

该模块的代码实现在 src/storage 目录下。这部分代码在功能上与提供的参考代码类似，但是我使用更加现代化的 C++ 实现进行了优化。

具体而言，文件管理系统实现为单例模式的 FileMapping 类，除了正常文件的打开关闭以外，还支持创建和删除临时文件（利用 mkstemp）。在销毁时，它会关闭所有文件并自动删除临时文件。

分页缓存池被实现成单例模式的 PagedBuffer 类，它通过 FileMapping 管理文件，支持按需读取和写回，支持 LRU 策略的缓存替换。在被销毁时它会自动将全部脏页写入文件中，它通过 std::shared_ptr 来确保 FileMapping 在它之后销毁。

SequentialAccessor 用于实现顺序的读写，它内部保存一个 fd 和一个 offset，能显著地方便上层实现。例如，当需要将表的元数据写入文件时，表可以将 SequentialAccessor 的引用传给属于它的 Fields，Fields 就可以直接将自己的数据写入文件。

此外，由于 B+ 树本身与数据库模块解耦，因此将其放在文件管理模块中，它提供各种查询接口供索引管理模块使用。为了确保 B+ 树实现正确，我实现了一个 googletest 测例，它对 B+ 树进行插入、删除、查找等操作，并检查其正确性，详细实现位于 test /test_btree.cpp。

2.3 记录管理模块

该模块用于实现记录存储，代码位于 src/engine/record.cpp 中。记录数据分页存储，每一页大小为 8 KB，每页开头记录该页有效记录的数量、下一个空页的页号，紧跟着是一个 FixedBitmap，用于记录每一个 slot 是否被使用，数据采用定长类型进行存储。记录长度、每页最大记录数量、第一个空页的页号等数据与表的元数据存储在一起。RecordManager 实现了对单张表记录读写的管理。提供了数据插入、删除、查找、修改的接口。

2.4 索引管理模块

该模块用于实现索引存储，代码位于 src/engine/index.cpp 中。IndexMeta 包含一个 B+ 树和一个引用计数，当引用计数为 0 时，B+ 树会被销毁。数据库进程结束时，主键、外键、unique 约束等在存储时仅需标记它们取了哪些列，下次启动时，数据库仅需按照这些列找到对应的 IndexMeta 即可。

2.5 查询处理模块

查询的执行基于迭代器实现；查询中的限制检查和修改操作分别由 WhereConstraint 和 SetVariable 执行。对于增删改操作，需要对各种约束情况加以考虑。以下详细介绍各个部分的实现思路。

2.5.1 迭代器

迭代器从基本功能上分为两类，一类从文件（记录和索引）中读取数据（并进行筛选），另一类在前一类的基础上进行连接、过滤、聚合等操作。而从具体优化上，可以允许迭代器一次从文件中读取多条记录（可以称为“带缓冲的分块读取”），这样在连接时可以采用分块的优化，减少磁盘 IO 次数。在实现时，迭代器往往需要记录它迭代的来源的列信息、它提供的结果的列信息、当前迭代的位置、传入的约束等信息。本项目实现了以下几种迭代器。

- `RecordIterator` 访问记录文件，分块读取
- `IndexIterator` 访问索引文件，分块读取
- `JoinIterator` 连接两个迭代器，这两个迭代器需要支持分块读取
- `PermuteIterator` 对上一个迭代器的列进行重排
- `AggregateIterator` 对上一个迭代器的结果进行聚合操作，在执行聚合时会顺便进行列重排
- `SortIterator` 对上一个迭代器的结果按照某一关键词进行排序

迭代器之间构成了一个树的结构，数据先由 `RecordIterator` 或者 `IndexIterator` 进行提取，然后由 `JoinIterator` 连接，接着通过 `PermuteIterator` 或 `AggregateIterator` 进行列重排（或进行聚合），最后如需排序，则通过 `SortIterator`。输出函数不断地从最顶部的迭代器取出元素，使各个迭代器不断地产生、筛选新的记录，整个过程类似于流水线。

`QueryPlanner` 的功能是按照查询语句构建迭代器树，当 `GROUP BY` 所需的列不在查询的列中时，它负责临时将其加入；`LIMIT` 和 `OFFSET` 限制也由它进行处理。如果某一个查询是一个子查询，那么它上层查询的 `WhereConstraint` 会基于它的 `QueryPlanner` 执行；否则，`QueryPlanner` 会直接送给输出模块。

2.5.2 查询描述、执行

`WhereConstraint` 是一个基类，它要求所有子类实现一个 `check` 函数来检查某条记录是否满足要求，它的子类包括

- `ColumnOpValueConstraint` 表示列和值之间的比较关系
- `ColumnOpColumnConstraint` 表示列之间的比较关系
- `ColumnNullConstraint` 进行 `NULL` 检查
- `ColumnLikeStringConstraint` 进行字符串模糊查询
- `ColumnOpSubqueryConstraint` 列和子查询的比较关系
- `ColumnInSubqueryConstraint` 列和子查询的包含关系

`SetVariable` 类中记录了待修改的列的 offset，以及需要修改成的值，它能且仅能对一条记录进行修改，关于约束的检查由上层代码完成。

2.5.3 增删改操作

为了方便调用，增删改操作在 `src/engine/scape_sql.cpp` 中实现，这样前端在解析 SQL 语句时可以直接调用这些封装好的函数。另外，为了优化外键约束的检查，我在 B+ 树的实现

中，允许叶节点存储一个 int32_t 类型的引用计数来表示这条记录被外键引用的次数。

对于 INSERT 操作，检查 PRIMARY,FOREIGN,UNIQUE 约束是否满足，若满足，则向记录以及每个 IndexMeta 管理的 B+ 树插入数据，并更新它引用的外键的引用计数。

对于 DELETE 操作，检查 FOREIGN 约束是否满足（即引用计数是否为 0），若满足要求，则可在索引和记录中删去。

对于 UPDATE 操作，虽然它等价于一次删除和一次插入，但主外键约束使得情况复杂一些。如果修改前后主键对应的列数值不变，那么只要检查外键约束是否满足即可（这就允许它的引用计数大于零）；否则，则可以等价为一次删除和一次插入，如果无法删除或无法插入，则恢复原状并报错。

2.6 系统管理模块

系统管理模块包括 GlobalManager,DatabaseManager,TableManager 三个层级，代码在 src/engine/system.cpp 中。它们都具有元数据的存取功能，而对于 TableManager，它还提供索引、主外键约束、unique 约束的增删功能。

2.7 SQL 语句解析模块

该模块使用 antlr4 进行实现，代码位于 src/frontend 目录下。它首先使用 antlr4 生成语法分析树，接着使用自定义的 Visitor 访问语法树，并在访问过程中实时地执行操作。

3 接口设计

3.1 文件管理模块

```
class PagedBuffer {
  // 为实现 LRU 刷新策略，使用双向链表
  void access(int id);
  // 根据指定的文件描述符和页号，从文件中读取一页
  uint8_t *read_file_rd(PageLocator pos);
  // 读取后将该页标记为脏页
  uint8_t *read_file_rdwr(PageLocator pos);
  // 将指定页标记为脏页，由于页的分配是连续的，可由指针判断属于哪个页
  bool mark_dirty(uint8_t *ptr);
};
class SequentialAccessor {
  void reset(int pagenum_);
  // 读取接口
  uint8_t read_byte();
  template <typename T> T read();
  std::string read_str();
```

```
  // 写入接口
  void write_byte(uint8_t byte);
  template <typename T> void write(T val);
  void write_str(const std::string &val);
};
```

3.2 记录管理模块

```
class RecordManager {
  // 创建表时使用的构造函数
  RecordManager(const std::string &datafile_name, int record_len);
  // 从文件中读取元数据的构造函数
  RecordManager(SequentialAccessor &accessor);
  // 元数据文件写入
  void serialize(SequentialAccessor &accessor);
  // 其他一些模块也有这样一组构造函数和析构函数，为了简洁，未来不再赘述

  // 获取指定位置的记录
  uint8_t *get_record_ref(int pageid, int slotid);
  // 插入记录并返回它的位置（页号和槽号）给索引使用
  std::pair<int, int> insert_record(const uint8_t *ptr);
  // 删除记录
  void erase_record(int pagenum, int slotnum);
};
```

3.3 索引管理模块

```
struct IndexMeta {
  // IndexMeta 本质上是 B+ 树的封装，便于从记录中提取键值
  // 为一个表建立外键约束时，它的外键 Offset 与被引用表的主键 Offset 不同
  // 因此需要通过 remap 函数重新建立偏移量
  std::shared_ptr<IndexMeta>
    remap(const std::vector<std::shared_ptr<Field>> &keys) const;
  // 从记录中提取键值
  std::vector<int> extractKeys(const KeyCollection &data);
  // 插入记录
  void insert_record(KeyCollection data);
  // 小于等于匹配，上层可以利用它查询是否出现过某个 key
  BPlusQueryResult le_match(KeyCollection data);
  // 获取引用计数
  uint32_t *get_refcount(uint8_t *ptr);
};
```

3.4 迭代器

```
class Iterator {
  virtual bool get_next_valid() = 0;
  virtual const uint8_t *get() const = 0;
};
class BlockIterator : public Iterator {
  // 对迭代器缓存的结果块进行操作、查询
  void block_next();
  bool block_end();
  bool all_end();
  // 让迭代器筛选出下一块记录，返回填入的记录数量
  virtual int fill_next_block() = 0;
  // 查询当前块/全部记录是否访问结束
  virtual void reset_all() = 0;
  void reset_block();
};
class RecordIterator : public BlockIterator;
class IndexIterator : public BlockIterator;
class JoinIterator : public BlockIterator;
class PermuteIterator : public Iterator;
// Gather 迭代器将打断流水线，它只有在将上层迭代器的结果全部读取后才能得到新的结果
class GatherIterator : public Iterator;
class AggregateIterator : public GatherIterator;
class SortIterator : public GatherIterator;

class QueryPlanner {
  // QueryPlanner 有大量的成员变量，调用前需要逐个填入，最后调用 generate_plan 生成迭代器
  void generate_plan();
  // 获取一条记录
  const uint8_t *get() const;
  // 进入下一条记录
  bool next();
};
```

3.5 SQL 执行接口

前文提到，数据库语句执行被包装成方便调用的一系列接口，它们能使代码更加整洁，并且可以方便地确保代码的健壮性，具体定义如下：

```
namespace ScapeSQL {
// 简单数据库管理操作
void create_db(const std::string &s);
void drop_db(const std::string &s);
```

```
void show_dbs();
void use_db(const std::string &s);
void show_tables();
void show_indexes();
void create_table(const std::string &s,
                  std::vector<std::shared_ptr<Field>> &&fields);
void drop_table(const std::string &s);
void describe_table(const std::string &s);
// 增删改查操作
void update_set_table(
    std::shared_ptr<TableManager> table,
    std::vector<SetVariable> &&set_variables,
    std::vector<std::shared_ptr<WhereConstraint>> &&where_constraints);
void delete_from_table(
    std::shared_ptr<TableManager> table,
    std::vector<std::shared_ptr<WhereConstraint>> &&where_constraints);
void insert_from_file(const std::string &file_path, const std::string &table_name);
// 增加、删除约束
void add_pk(const std::string &table_name, std::shared_ptr<PrimaryKey> key);
void drop_pk(const std::string &table_name, const std::string &pk_name);
void add_fk(const std::string &table_name, std::shared_ptr<ForeignKey> key);
void drop_fk(const std::string &table_name, const std::string &fk_name);
void add_index(const std::string &table_name, std::shared_ptr<ExplicitIndexKey> key);
void drop_index(const std::string &table_name, const std::string &index_name);
void add_unique(const std::string &table_name, std::shared_ptr<UniqueKey> key);
} // namespace ScapeSQL
```

4 实验结果

本课程项目能够通过全部 CI 测例，在本机（CPU: R7 7840HS）下，限制分页缓存为 64MB，能够在 10 秒内完成全部测试。以下是本机测试截图：

```
Passed cases: comb-fk, comb-fk-schema, comb-pk, comb-pk-schema, data, date, fk, fk-schema, index-data, index
-schema, join, join-data, multi-join, null, optional, pk, pk-schema, query-a, query-aggregate, query-b, quer
y-c, query-d, query-data-a, query-data-b, query-fuzzy, query-group, query-nest, query-order, system, table,
table-data, unique
Failed cases:
Skipped cases:
Disabled cases:
Scores: 80 / 80, Time: 8.198s
```

本人能力有限，在最初设计时没有考虑到变长字符串、增删列的需求，导致难以在现有代码框架上实现这些功能，故没有完成这一部分扩展内容。

5 参考文献

1. 课程组提供的实验指导书
2. Antlr4 教程（`https://tomassetti.me/antlr-mega-tutorial/`）
3. C++ 正则表达式教程（`https://en.cppreference.com/w/cpp/regex`）