数据库系统概论项目报告

2017011472 李琪斌
2017011364 罗崚骁

系统架构设计

本数据库管理系统名为 OrangeDB，主要由文件管理系统和数据库管理模块组成。

文件管理系统对纯的二进制流进行读取和写入文件的操作，并且提供了 LRU 缓存机制，在内存中有一定的缓存空间。此外文件系统还提供了在硬盘上分配空间的功能，类似于内存中的 new 操作。

数据库管理模块包含语法解析、语义分析、表格操作和索引四部分。语法解析用于将用户输入转化为抽象语法树，并送给语义分析。语义分析根据指令的不同类型调用不同的数据库操作或表格操作函数。表格操作则集中处理表格中的数据，在必要时使用索引来加速。

数据库采用列式数据库，每个列保存在一个单独的文件中，这样可以方便地增加和删除数据表中的列。索引使用 B 树索引，数据信息也保存在文件中，每个结点大小是一个页的大小，可以减少硬盘读取次数。

数据库管理模块的目录结构如下图所示：

```
dbms
├── fs (文件管理系统)
|   ├── allocator (在硬盘上进行空间管理和分配、)
|   ├── bufpage (缓存页管理模块)
|   └── file (文件管理模块)
├── orange (数据库管理模块)
|   ├── index (B树索引的数据结构)
|   ├── parser (SQL 语法解析器)
|   ├── syntax (SQL 语义分析器)
|   └── table (对单独一个表的操作)
├── utils (定义一些全局变量、帮助函数)
└── terminal (提供终端交互)
```

此外我们还提供了网页前端，分别在 `server` 和 `frontend` 目录下，服务器使用 go 语言编写，内容很少；前端使用 vue 编写，内容也很少。

各模块详细设计

文件系统

在提供的文件管理模块的基础上进行修改，大体思路没有改变，重构了不能跨平台的内容，重新设计了一些数据结构来优化速度（比如缓存页的管理）。

新设计了一个文件类，文件类建立在页式文件系统的基础上，支持按照一定偏移量读取文件内容，自动使用页式文件系统的缓存功能。

此外，还在硬盘级别上实现了类似于操作系统管理堆内存的功能，用于分配不定长记录的空间（见 allocator.h）。

```
class File {
```

```
...
public:
    // 文件系统操作再封装
    static bool create(const String& name);
    static File* open(const String& name);
    static File* create_open(const String& name);
    bool close();
    static bool remove(const String& name);

    // 向文件中 offset 偏移处写字节
    size_t write_bytes(size_t offset, const_bytes_t bytes, size_t n) const;
    // 按照类型写
    template <typename T, typename... Ts>
    size_t write(size_t offset, const T& t, const Ts&... ts) const;

    // 从文件中 offset 偏移处读字节
    size_t read_bytes(size_t offset, bytes_t bytes, size_t n) const;
    // 按照类型读
    template <typename T, typename... Ts>
    size_t read(size_t offset, T& t, Ts&... ts) const;
    template <typename T>
    T read(size_t offset) const;
};
```

数据库

维护数据库的增删使用等，在文件系统上维护文件夹即可，比较简单。

```
namespace Orange {
    void setup();
    bool exists(const String& name);
    bool create(const String& name);
    bool drop(const String& name);
    bool use(const String& name);
    //  正在使用某个数据库
    bool using_db();
    bool unuse();
    String get_cur();
    // 所有数据库
    std::vector<String> all();
    std::vector<String> all_tables();
};  // namespace Orange
```

表

表中的数据是定长的，每一列的数据单独存放于一个文件中（方便对列进行修改，支持 `alter table` 系列语句）,通过计算 rid 和每个数据的大小的乘积来计算存储的偏移量，再通过文件访问。实现上，在表中维护了一个内部类。

每张表使用一个文件存储表的信息（各列信息，主键外键等）。

rid 则通过一个栈来维护：需要新的 rid 时首先检查栈中是否有剩余，有直接从栈中取，否则新的 rid 为当前记录总数；删除一条记录时，回收此 rid（放入栈中）。

提供了对表和数据的增删查改的接口。对于数据的查询操作，所有的返回结果都为一张表，实现中对应类 `TmpTable`，而数据库中存储的表为类 `SavedTable`，它们都有查询的接口（为了支持嵌套查询），都继承于一个表的基类。

```
class Table {
...
protected:
    std::vector<Column> cols;

    ...

    virtual byte_arr_t get_field(int col_id, rid_t rid) const = 0;
    auto get_fields(const std::vector<Column>& cols, rid_t rid) const;

    // 返回所有合法 rid
    virtual std::vector<rid_t> all() const = 0;
    // 用 where 进行筛选
    virtual std::vector<rid_t> filt(const std::vector<rid_t>& rids,
        const ast::single_where& where) const;

public:
    ...
    // 返回满足 where clause 的所有 rid
    // 为了测试放到了 public
    virtual std::vector<rid_t> where(const ast::where_clause& where,
        rid_t lim = MAX_RID) const;
    virtual TmpTable select(const std::vector<String>& names,
        const ast::where_clause& where, rid_t lim = MAX_RID) const = 0;
    TmpTable select_star(const ast::where_clause& where, rid_t lim = MAX_RID)
const;
}

class TmpTable : public Table {
// 实现抽象方法
...
};

class SavedTable : public Table {
private:
    // 各列数据接口
    struct ColumnDataHelper {
        File* f_data;
        int size;  // 每个值的大小
        ast::data_type type;
        FileAllocator* alloc;
        String root, name;
        ...

        void change(const ast::field_def& def, const std::vector<rid_t>& all);

        // 对于 vchar 返回指针，其它直接返回真实值
        byte_arr_t store(const byte_arr_t& key) const;
        byte_arr_t restore(const_bytes_t k_raw) const;
        // 对于 varchar 返回指针
        auto get_raw(rid_t rid) const;
        //  返回 rid  记录此列的真实值
        byte_arr_t get_val(rid_t rid) const;
```

```
        // 返回一系列 rid 对应的该列值
        std::vector<byte_arr_t> get_vals(const std::vector<rid_t>& rids) const;

        void insert(rid_t rid, const ast::data_value& value);

        void remove(rid_t rid);
    };

    std::vector<ColumnDataHelper*> col_data;

    // 读写表的基本信息
    void write_info();
    void read_info();

    ...
public:
    // 创建表
    static bool create(const String& name, const std::vector<Column>& cols,
                       const std::vector<String>& p_key,
                       const std::vector<f_key_t>& f_key_defs);

    // 根据表名获取表
    static SavedTable* get(const String& name);
    // 描述表
    TmpTable description() const;

    // 关闭表，写回数据缓存
    bool close();

    // 删除表
    static bool drop(const String& name);

    // 按列插入数据，补齐默认值，返回插入成功条数
    rid_t insert(const std::vector<String>& col_names, ast::data_values_list
values_list);
    // 插入完整的数据
    rid_t insert(const ast::data_values_list& values_list);

    // 按照 where 删除记录，返回被删除的数据条数
    rid_t delete_where(const ast::where_clause& where);

    // set null，用于 update 操作
    void set_null(rid_t rid, const std::vector<String>& col_names);
    // 更新
    void update_where(const ast::set_clause& set, const ast::where_clause&
where);
    // 索引增删
    void create_index(const String& idx_name,
        const std::vector<String>& col_names, bool primary, bool unique);
    void drop_index(const String& idx_name);
    // 主键增删
    void add_p_key(String p_key_name, const std::vector<String>& col_names);
    void drop_p_key(const String& p_key_name);
    // 外键增删
    void add_f_key(const f_key_t& f_key_def);
    void drop_f_key(const String& f_key_name);
    // 列增删改
    void add_col(const Column& new_col);
```

```
    void drop_col(const String& col_name);
    void change_col(const String& col_name, const ast::field_def& def);
    // 重命名表
    static void rename(const String& old_name, const String& new_name);
    // 多表连接
    static TmpTable select_join(const ast::select_tb_stmt& select, rid_t lim =
MAX_RID);
    // 一系列聚合函数
    TmpTable count(const String& col_name) const;
    TmpTable sum(const String& col_name) const;
    TmpTable avg(const String& col_name) const;
    TmpTable min(const String& col_name) const;
    TmpTable max(const String& col_name) const;
};
```

索引

索引中记录的是当前索引属于哪张表，维护了哪些列，并且为每个索引创建一棵 B 树用于有序地维护数据。支持联合索引，按照索引的列的顺序进行多关键字排序。不过查询时，对于高维偏序问题的实现仍然是按照第一列在 B 树上缩小范围，后面的关键字线性进行判断。

null 值不会放入索引，因此查询时，使用索引的条件必须是查询涉及的列和索引的列完全一致，否则可能会漏查。

B 树的实现中，varchar 存放的仍然是指针，获取值时需要一次额外的 IO；另外，为了保证数值的唯一性，需要将 rid 也一并维护，作为最后一个关键字。

索引基本信息的维护、B 树数据、编号的维护和表的基本信息以及数据、标号维护类似。

```
class Index {
private:
    SavedTable &table;
    String name;
    std::vector<Column> cols;
    bool primary, unique;

    int key_size;
    BTree *tree;

    void write_info();
    void read_info();
public:
    // 判断是否存在某个值
    bool constains(const std::vector<byte_arr_t>& vals) const;
    // 返回所有等于对应 key 值的记录编号
    std::vector<rid_t> get_on_key(const_bytes_t raw, rid_t lim = MAX_RID) const;
    // 新建索引
    static Index* create(SavedTable& table, const String& name, const
std::vector<Column>& cols, bool primary, bool unique);
    // 加载索引
    static Index* load(SavedTable& table, const String& name);
    // 删除索引
    static void drop(const Index* index);

    // 插入数据，raw: 索引值；val：真实值
    void insert(const byte_arr_t& raw, rid_t rid);
    // 删除数据
```

```
    void remove(const byte_arr_t &raw, rid_t rid);
    // 对于谓词进行查询
    std::vector<rid_t> query(const std::vector<Orange::preds_t>& preds_list,
rid_t lim) const;
    // 更新数据
    void update(const byte_arr_t &raw, const byte_arr_t& new_raw, rid_t rid);
};
```

```
class BTree {
private:
    // B 树结点类型
    struct node_t {
        const BTree &tree;
        byte_t data[PAGE_SIZE];
        bid_t id;

        int& key_num();
        bid_t& ch(int i);
        bytes_t key(int i);
        void set_key(int i, const_bytes_t key);
        rid_t& val(int i);
    };
    // 防止内存问题，使用了唯
    using node_ptr_t = std::unique_ptr<node_t>;

    node_ptr_t new_node();
    node_ptr_t read_node(bid_t id) const;

    node_ptr_t root;
    // 只需要额外保存 B 树根的编号
    void read_root();
    void write_root();
public:

    void init();
    void load();

    void insert(const_bytes_t k_raw, rid_t v);
    void remove(const_bytes_t k_raw, rid_t v);
    bool contains(const std::vector<byte_arr_t>& ks) const;
    // 返回所有值等于 ks 的记录
    std::vector<rid_t> get_on_key(const std::vector<byte_arr_t>& ks, rid_t lim)
const;
    // preds_list[i] 表示第索引中 i 列的约束
    std::vector<rid_t> query(const std::vector<preds_t>& preds_list, rid_t lim)
const;
};
```

语法解析与语义分析

借助 Boost.Spirit 库实现从 SQL 语句到抽象语法树的解析工作。具体操作是首先定义了抽象语法树结点的结构体（在 sql_ast.h 中），然后参照 Spirit 库提供的文档和例子编写产生式规则。结点中大量使用了 boost::optional 和 boost::variant，前者表示可能出现 0 次或 1 次的语法成分，后者表示多个成分只会出现一个。这样的好处是便于分发抽象语法树的结点到各自对应的处理函数中。

AST 结点样例

```
struct column {
    boost::optional<std::string> table_name;  // 表格名称，可选
    std::string col_name;                     // 列名称
};
```

AST 处理函数样例

```
switch (stmt.kind()) {  // 根据不同类型调用不同处理函数
    case StmtKind::Sys: return sys(stmt.sys());
    case StmtKind::Db: return db(stmt.db());
    case StmtKind::Tb: return tb(stmt.tb());
    case StmtKind::Idx: return idx(stmt.idx());
    case StmtKind::Alter: return alter(stmt.alter());
    default: unexpected();
}
```

实验结果

- 支持产生式文档中提供的所有语句
- 支持查询单个聚合函数
- 有较快的操作速度
- 支持多表连接

小组分工

罗崚骁：缓存页设计，文件设计，数据库和表格的设计，B树索引

李琪斌：硬盘空间分配，语法解析模块，网页前端

共同完成：重构文件管理系统，语义分析，终端，单元测试

参考文献

[1]Thomas H. 等.算法导论（原书第 3 版）[M].机械工业出版社:北京,2013:277.