# TinySQL Technical Report

支持：

1. 基本的SELECT （不包括Join，不包括嵌套子查询，可投影）
2. INSERT
3. DELETE
4. UPDATE （部分）

我作为组长，负责了如下的部分：

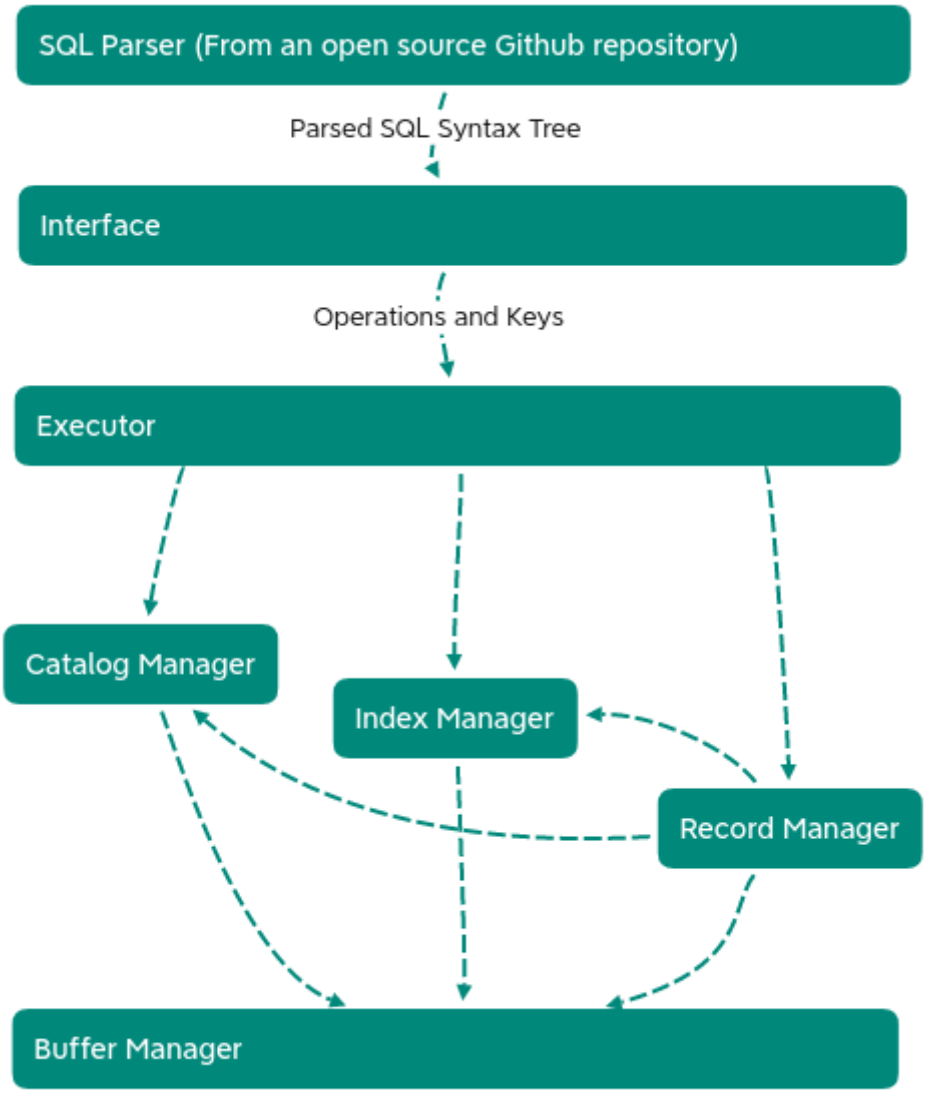
1. 项目的整体结构
2. Index Manager的整体设计与全部实现，包括B+树的存储设计与B+树的实现
3. Record Manager的部分设计，包括Row-based的数据库在磁盘上的存储和读取方法
4. Record Manager的部分实现，主要是Row-based的文件管理相关的内容
5. Catalog Manager的部分实现，主要是联调时的大量Debug工作
6. Parser的调试与整合（主要体现为Interface的设计以及与Executor的交互）

以下将简述我负责完成的部分与一些调试的心得。

# 项目的概述

## 项目结构

最终项目的结构如下图所示：




**图中的箭头表示调用的关系。如Record Manager有一个箭头指向Catalog Manager，说明Record Manager会调用Catalog Manager。**

SQL Parser: [https://github.com/hyrise/sql-parser] 完成SQL Query的语义解析

Interface: 解析Parser给出的语义树，判断具体操作的内容（如是create类型语句，还是select类型语句），将诸如select, where 等查询中的表名，投影Attribute名，Key值解析出来，调用Executor。

Executor: 完成原子性的select，create，insert操作。

Catalog Manager : 管理有关表名称，Attribute，是否有Index，Index Name等信息

Index Manager : 管理具体的索引文件。具体而言，就是管理磁盘上存储的B+树。

Record Manager : 管理存放表的文件。使用Row-based的设计方法。

Buffer Manager : 管理缓冲区，其他模块对具体的文件的某一块数据的请求都要通过这一模块来获取。

## 一些关键的设计

1. 我们的数据库采用Row-based方法来存储数据，使用Tuple进行管理，并使用懒惰删除和添加的方法。（后续的优化可以是通过添加定期的文件整理来优化文件，但总归其实是偷懒的做法）
2. 文件管理：Record Manager给每一个表单独创建一个文件，存储二进制的数据；Catalog Manager给每一个表单独创建创建一个记录Attribute等信息的文件，存储用特殊的（由区俊豪同学设计的）编码方式写入的字符数据；Index Manager给每一个Index单独创建一个文件，存储B+树。
3. 每一条数据单个Attribute所对应的数据长度不能超过20B，Tuple的长度没有显示地限制，但至少应当小于一个Block的大小。
4. Buffer Manager使用时钟替换算法进行Buffer的替换。
5. Parser使用了外部的开源库，将他们的代码整合成为了我们项目中的一个模块。其github链接已在前面给出。

# Index Manager与B+ 树设计概述

总体而言，B+树是Index的核心。在本次设计中，B+树的一个Node最大的大小被设计为数据库规定的一个Block大小，通过在代码中的约束来控制。

## Page的设计(bpTree_block.h)

Page的大小为一个Block的大小，典型值为4096B。设计为Header + Data的形式

Header: （24B）(注：`db_size_t`定义为`uint32_t`)

```
/**
 * Both internal and leaf blocks are inherited from this.
 *
 * It actually serves as a header part for each B+ tree page and
 * contains information shared by both leaf block and internal block.
 *
 * Header format (size in byte, 24 bytes in total):
 * ---------------------------------------------------------------------------
 * | BlockType (4) | ParentBlockId(4) | CurrentSize (4) | MaxSize (4) |
 * ---------------------------------------------------------------------------
 * | ParentBlockId (4) | BlockId(4) |
 * ---------------------------------------------------------------------------
 */
struct bpTree_Block{
    /**
     * Init a block, neither leaf nor internal node.
     * Usually occurs when a table exists but has no content
     * @param myBId
     * @param parentId
     */
    void init(blockId_t myBId, blockId_t parentId){
        _blockType = INVALID_BLOCK;
        _parent_block_id = parentId;
        _block_id = myBId;
    }

    bool isLeaf() const{
```

```
        return _blockType == LEAF_BLOCK;
    }

    blockType_t _blockType;
    blockId_t _parent_block_id;
    blockId_t _block_id;
    db_size_t _size;
    db_size_t _max_size;
    blockId_t _next_block_id;

    bpTree_Block() = delete;
};
```

Internal Page和leaf Page均继承自此，只多一个MAXSIZE大小的键值对，MAXSIZE对于leaf和Internal是不同的，均使用宏定义；Internal Page和Leaf Page的键值对类型应当是不一样的，并且其中的一些函数，如二分查找函数，也是不一样的。

```
#define MAPPING_T std::pair<key_t, value_t>

KEY_VALUE_T_DECLARE
struct bpTree_Leaf : public bpTree_Block{
    /**
     * Init a leaf node.
     * @param myBId block id for this node
     * @param parentId block id for its parent
     * @param next_block_id (right) sibling block id
     */
    void init(blockId_t myBId, blockId_t parentId, blockId_t next_block_id);

    /**
     * The smallest key that is greater or equal to the given key.
     * If the given key is greater than all the key, return the position after
the last element.
     *
     * @param key The given key
     * @return the position of the smallest key that is greater or equal to the
given key.
     */
    db_size_t leaf_biSearch(const key_t key);

    MAPPING_T _k_rowid_pair[MAX_LEAF_SIZE + 1]; ///< One extra space for
spanning.

    bpTree_Leaf() = delete;
};

KEY_VALUE_T_DECLARE
struct bpTree_Internal : public bpTree_Block{
public:
    /**
     * Init an internal node.
     * @param myBId block id for this node
     * @param parentId block id for its parent. INVALID_BLOCK_ID for no parent.
     */
    void init(blockId_t myBId, blockId_t parentId);
```

```
    /**
     * Return the position of the greatest key that is smaller or equal to the
given key.
     *
     * @param key The given key
     * @return the position of the greatest key that is smaller or equal to the
given key.
     */
    db_size_t internal_biSearch(const key_t key);

    MAPPING_T _k_child_pair[MAX_INTERNAL_SIZE + 1]; ///< One extra space for
spanning.

    bpTree_Internal() = delete;
};
```

通过以上方法我们就定义了每一个Page的存储方法。注意，这两个类的定义均用来描述一块内存空间，也就是说，它们不应当被用户显式地构造，也不应当有这两个类的Object出现，因此在上面的代码中将其默认构造函数删除。

当我们通过Buffer Manager 获取到了一块大小为4096B的空间的头指针时，使用如下的这段代码即可将其描述为我们上面所定义的这种方式：

```
char* raw = _bfm->getPage(file_name, block_id); //get a block from a specific
file
auto block = reinterpret_cast<BP_TREE_BLOCK_T*>(raw)
```

以上是B+树的Page设计和一点工程技巧。

如您对细节感兴趣，请参看DOC（使用DOXYGEN生成，不同文件根据完成的紧急程度DOC完整程度也不一...看时请息怒）：（源代码可以通过网页底部的链接直接查看）

structbpTree__Block.html

## B+树的实现 （bpTree_disk.h）

B+树的具体实现均参考Database System Concepts中的伪代码，结合我设计的Page来实现。不同的是，这里的B+树没有一个确定的degree (n)。对于连接Leaf Node的Internal Node，其Degree就是Leaf Page中定义的LEAF_MAX_SIZE；对于连接Internal Node的Internal Node，其Degree就是Internal Page中定义的INTERNAL_MAX_SIZE。但B+树的良好性质并不会发生改变，只要还是保证非根节点总是至少半满的，那么B+树良好的查询性质就仍然维持。

其他的实现细节在此略。如您对细节感兴趣，请参看我们的DOC。

classBp__tree.html

## Index Manager

在本项目中，Index Manager其实就是一个B+ 树的Wrapper，只是加上了对文件名的处理，与Buffer Manager的交互等琐碎的工作，并没有什么算法的工作量。DOC如下（有每个函数比较详细的说明）：

classIndexManager.html

# Buffer Manager的存储设计

Buffer Manager的具体算法，如插入、删除、更新等，由本组成员李明泽完成。这里简要说明一下我针对Buffer Manager所设计的一些存储方法。（主要是data_t.h中的内容，涉及到一条记录应该怎么存）

我们定义了基本的数据存储方法，如下所示：

一个Cell（即一个Attribute所对应的位置存放的数据）

```
struct Data{
    BASE_SQL_ValType type;
    union {
        int i_data;
        float f_data;
        char s_data[20];
    } data_meta{};

    Data(){};
    //... Some other functions, like operator> , etc.
}
```

Tuple分为两种，一种为MemoryTuple，另一种为DiskTuple，分别为在内存中存储的Tuple和在磁盘中存储的Tuple。定义如下：

```
typedef std::vector<Data> MemoryTuple;

/**
 * @brief One row in a table
 *
 * DiskTuple is a row in a table.
 */
struct DiskTuple{
    db_size_t _total_len;
//    db_size_t _current_len;
    bool isDeleted_;
    Data cell[];

    DiskTuple()= delete;
    DiskTuple(DiskTuple&) = delete;
    DiskTuple(DiskTuple&&) = delete;

    void serializeFromMemory(const MemoryTuple& in_tuple);
    MemoryTuple deserializeToMemory(const std::vector<int>& pos = {});
    [[nodiscard]] std::vector<Data> getData() const;  //返回数据
    [[nodiscard]] db_size_t getSize() const{
        return _total_len;
    }  //返回元组的数据数量
    [[nodiscard]] db_size_t getBytes() const{
        return sizeof(DiskTuple) + sizeof(Data) * _total_len;
    }
//    DiskTuple<len>& operator=(DiskTuple t);
    bool isDeleted();
    void setDeleted();
```

```
};
```

如上所示，MemoryTuple为了方便起见直接定义为一个Data的Vector；而DiskTuple则为一个未完成类型（包含一个不定长的Data类型的数组），并且不允许用户显式地声明。（将几个常用的构造函数都删除了）在从磁盘中读取数据时，需要通过BufferManager获得对应文件块数据的指针，并将其`reinterpret_cast`成一个DiskTuple的指针，然后通过getData函数将其中的数据转化成一个DiskTuple（也就是Data的vector）。

相应的，一个RecordPage设计如下：

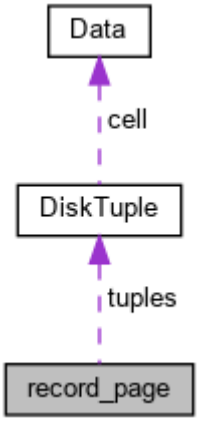
```
struct record_page{
    int tuple_num;
    db_size_t record_bytes;
    DiskTuple tuples[];

    DiskTuple& tuple_at(int offset){
        if (offset == 0) return tuples[0];
        else{
            if (offset >= tuple_num){
                throw;
            }
            else{
                char* start = (char*)tuples;
                start = start + offset * tuples[0].getBytes();
                return *((DiskTuple*)start);
            }
        }
    }

    ///< Safety purpose
    template<class T>
    DiskTuple& operator[](T offset) = delete;
};
```

**特别注意：** record_page有一个很关键的函数：`tuple_at`。这里想要实现的效果类似于tuples[offset]返回对应的Tuple，但是如果直接使用方括号对数组进行访问有如下的问题：DiskTuple包含一个不定长的Data数组Cell，那么这里的DiskTuple的数组使用方括号进行寻址时，其对DiskTuple指针做运算，加上和减去的长度都只不过是DiskTuple的Cell为空时的大小（与sizeof(DiskTuple）一致）。这样就会产生很严重的问题：DiskTuple的内存在访问时没有对齐。

因为暂时没有想到更好的方法，所以这里根据0号Tuple中的信息（同一个Table中的Tuple是一样长的，并且0号Tuple访问时无需考虑内存偏移的问题）手动计算了相应offset下Tuple的地址，实现了和`[]`访问类似的效果。




Record Manager 的 DOC如下：

classRecordManager.html

# 关于Parser的一点说明

Parser使用了开源的库sql-parser。正常的SQL语句应当都能够被处理，但是绝大多数我们的tinySQL处理不了。对于处理不了的SQL语句，会输出`>> Unsupported valid SQL command. May support later.`。只有错误的SQL语句会输出相应的错误信息。

# 简短使用说明

本项目支持的语句包括：

```
select [] from [] where [] (and []...) # DO allow AND, do not allow subquery
insert into [tableName] values(...)
delete from ... where [] (and [] ...) # DO allow AND, do not allow subquery
create table ...
create index...
drop table ...
drop index ...
show tables;

#special : read from file
READ FILE; # Then input the file name according to the prompt
```

构建项目的方法（Linux下使用命令行构建）

```
# bash at our code release directory
mkdir build
cd build
cmake ..
make
# the tinySQL and libSQLparser.so is generate. You can directly test here. If you use test files, copying them to build directory may be helpful
```

# 开发心得

对需要精确控制内存和存储的软件开发有了更深的认识，对数据库中的许多知识有了切实的感受；同时也学会了使用一些重要的工具，如doxygen