实验二：实现多种类型的 CloumnWriter

- 截止时间：2025 年 01 月 20 日 22:00
- 提交方式：OBE **pdf** 实验报告（**需注明自己的 github 代码仓库**）

1 实验概述

在 TPC-H 数据生成时，会生成以 *.tbl* 后缀的文件。数据格式通常是文本形式，每一行代表一条记录，记录中的不同字段（列数据）通过特定的分隔符（通常是 “|”）进行分割。例如在一个存储客户信息的 *.tbl* 文件中，一行数据可能像这样 1|*Customer*|*Address*|*City*|*Region*|*Country*|*Phone*|，其中每个 “|” 隔开的部分代表不同的客户属性。

而在开源系统 pixels 中，我们使用如下的数据格式：首先按行将数据表划分成若干个 Row Group，一个或多个 Row Group 存储在一个文件中。Row Group 内部按列独立编码并压缩，每个列上的所有数据项被存储为一个 Column Chunk（图 1 中 $c_1, c_2, \ldots, c_n$）。Row Group Footer 中存储了 Row Group 的元信息，包括行数、最大值、最小值等。

本次实验同学们需要实现将 *.tbl* 转化为 pixels 所能读取的 *.pxl* 文件格式。

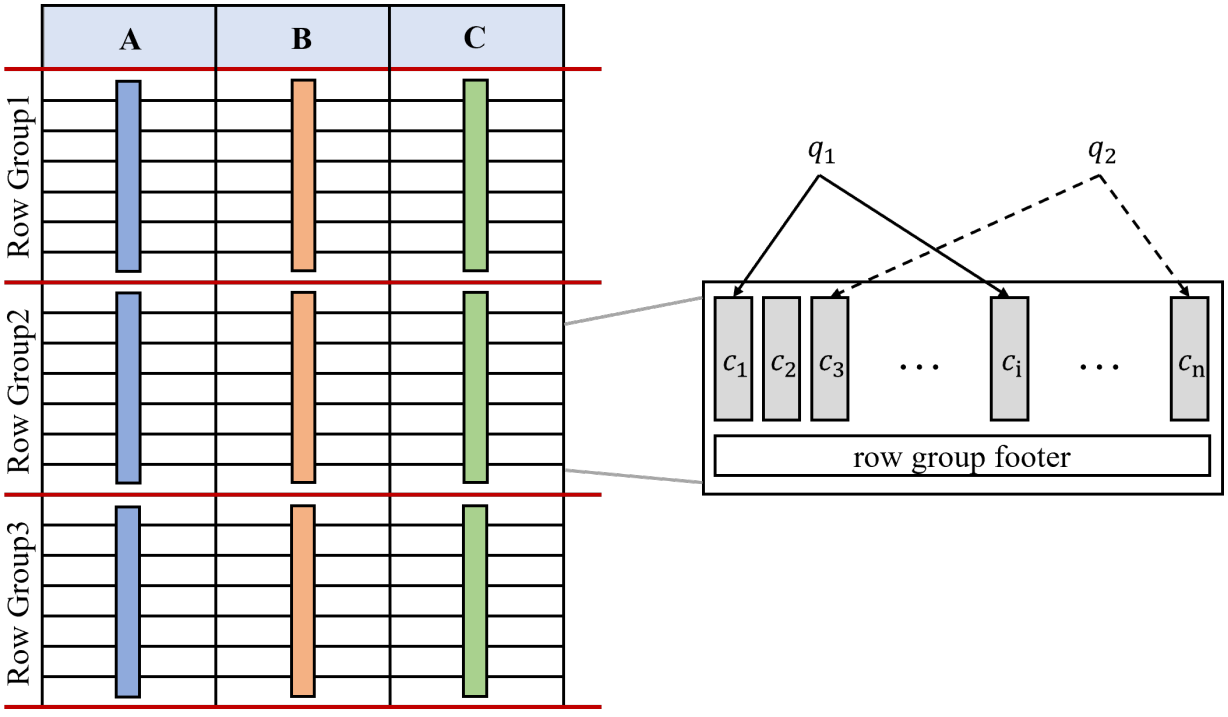

图 1: pixels 列存存储格式

2 实验准备

2.1 Git

实验基于 mini-pixels 的 feature/addWriter 进行开发。同学们需要 fork 相关分支，并创建自己仓库的开发分支。一般情况下，自己仓库的 master(main) 分支要保持与原仓库一致。开发分支的命名通常采用 *type*/*function* 的形式。*type* 表示该分支的类型，例如实现新功能 (feature)、修复

漏洞 (fixbug) 等；*function* 表示该分支具体功能，使用驼峰命名法，例如 *addWriter*。例如本次开发分支名可以为 *feature*/*addWriter*。

自己仓库的 master 分支在与原仓库保持同步时，开发分支和 master 分支可能出现如下情况：

```
      A---B---C feature
     /
D---E---A'---F master
```

可以通过 git rebase 命令来整合 feature 分支

```
               B'---C' feature
              /
D---E---A'---F master
```

常用的 Git 命令如下：

- git clone
- git remote
- git branch
- git checkout
- git pull
- git add
- git commit
- git push
- git rebase
- git stash

在合并分支时可能会出现 conflict，需要手动解决冲突。冲突简单来讲是同一部分代码存在不同的修改，以至于无法自动合并。解决冲突就是选择如何修改这部分代码。具体如何解决冲突要视情况而定，可以采用 vscode、idea 等工具或者直接在 github 中处理。

2.2 CMake

暂时没找到较好的视频讲解，可参考如下链接学习：

- cmake tutorial
- Modern CMake
- Modern CMake 简体中文版

3 实验介绍

3.1 代码解读

从 cpp/pixels-cli/main.cpp 开始，main 函数会解析我们输入的命令 (LOAD)，并调用 LoadExecutor 类的 execute 函数。LoadExecutor 类的 execute 函数会调用该类的 startConsumers 函数。startConsumers 函数创建 PixelsConsumer 类，并执行该类的 run 函数。

3.1.1 add 方法实现

我们输入命令中的 schema 描述了我们将要输入的表的结构，包括所有的列名和列类型。run 函数里面会根据 schema 创建出对应的 ColumnVector 指针数组 columnVectors。pixels 中有多个数据类型的 ColumnVector，例如 LongColumnVector，它们都继承于 ColumnVector。这里创建的 ColumnVector 指针数组中每个元素都是由子类转换成的父类 ColumnVector 指针。在父类 ColumnVector 中定义了 add 等虚函数，并在子类 LongColumnVector 中实现，所以当我们使用 columnVectors[i] 的 add 方法时，会调用子类的 add 方法。

读取文件内容后，通过调用 add 方法可以将数据暂时保存在对应的 columnVectors 中。

目前已经实现了 LongColumnVector 的 add 方法作为参考。LongColumnVector 用来保存数据类型为 short、int 和 long 的列，类中定义了 longVector 和 intVector 两个变量用来保存数据，并根据标志变量 isLong 来判断具体存储数据的是 longVector 还是 intVector。

在子类 LongColumnVector 中定义了四个 add 方法，接受 string、bool、int、int64_t 类型的参数，并在父类 ColumnVector 中声明了对应的四个 add 虚函数。例如接受 int 类型的 add 方法实现，首先要根据类成员变量 writeIndex 和 length 判断现有 vector 大小是否能够继续增添元素，writeIndex 表示我们将要写入的位置，length 表示当前 vector 的长度。如果当前 vector 已满，我们需要通过 ensure 函数扩展 vector 的大小为原先的**两倍**（其他子类实现的时候也可以按照两倍）。然后根据标志变量 isLong 将 value 写入到对应的 vector 中，并将 isNull 数组对应项置为 false。isNull 是父类中定义的成员变量，用来表示 vector 中对应项是否为空。

```
void LongColumnVector::add(int value) {
    if (writeIndex >= length) {
        ensureSize(writeIndex * 2, true);
    }
    int index = writeIndex++;
    if(isLong) {
        longVector[index] = value;
    } else {
        intVector[index] = value;
    }
    isNull[index] = false;
}
```

3.1.2 ColumnWriter 实现

函数中创建的 pixelsWriter 将负责把 columnVectors 写入 *.pxl* 文件中。在 cpp/pixels-core/lib/PixelsWriterImpl.cpp 中，addRowBatch 一次将一个 RowBatch 写入文件，addRowBatch 调用 writeColumnVectors，将 RowBatch 中的 ColumnVector 的数据 (就是上一步 add 加入的数据) 写入文件，随后判断是否达到一个 rowGroup 大小，如果达到，则写入 rowGroup 信息，如果判断已经达到了一个文件中最大行数，则重新初始化一个 PixelsWriter 实例，写入一个新的文件，大体代码逻辑如下

```
for (std::string originalFilePath : queue) {
while (std::getline(reader, line)) {
    if (initPixelsFile) {
        pixelsWriter = std::make_shared<PixelsWriterImpl>(...);
    }
    initPixelsFile = false;

    ++rowBatch->rowCount;
    ++rowCounter;
    // 判断分割符 切分 line
    std::vector<std::string> colsInLine;
    boost::sregex_token_iterator it(line.begin(), line.end(), boost::regex(regex), -1);
    for (; it != boost::sregex_token_iterator(); ++it) {
        colsInLine.push_back(*it);
    }
    for(int i = 0; i < columnVectors.size(); ++i) {
        if (判断是 NULL) {
            columnVectors[i]->addNull();
        } else {
            columnVectors[i]->add(colsInLine[i]);
        }
    }
    if (rowBatch->rowCount == rowBatch->getMaxSize()) {
        pixelsWriter->addRowBatch(rowBatch);
        rowBatch->reset();
    }
    // 创建一个新的文件
    if (rowCounter >= maxRowNum) {
        if (rowBatch->rowCount != 0) {
            pixelsWriter->addRowBatch(rowBatch);
            rowBatch->reset();
        }
```

```
            pixelsWriter->close();
            this->loadedFiles.push_back(targetFilePath);
            rowCounter = 0;
            initPixelsFile = true;
        }
    }
    }
        }
        // 剩余 line 写入文件
        if (rowCounter > 0) {
            if (rowBatch->rowCount != 0) {
                pixelsWriter->addRowBatch(rowBatch);
                rowBatch->reset();
            }
            pixelsWriter->close();
            // 写入 FileTail
            this->loadedFiles.push_back(targetFilePath);
        }
```

PixelsWriter 会根据数据类型，初始化对应类型的 ColumnWriter, 这部分代码在 cpp/pixels-core/lib/writer/ColumnWriterBuilder.cpp 中，这里应用了虚函数实现的多态，根据传入的类型将创建的派生类指针转为基类:

```
    std::shared_ptr<ColumnWriter> ColumnWriterBuilder::newColumnWriter(
    std::shared_ptr<TypeDescription> type,
    std::shared_ptr<PixelsWriterOption> writerOption) {
    switch(type->getCategory()) {
        case TypeDescription::SHORT:
        case TypeDescription::INT:
        case TypeDescription::LONG:
            return std::make_shared<IntegerColumnWriter>(type, writerOption);
```

ColumnWriter 将 ColumnVector 写入文件中，大体流程为

```
int IntegerColumnWriter::write(std::shared_ptr<ColumnVector> vector, int size)
{
    std::cout<<"In IntegerColumnWriter"<<std::endl;
    auto columnVector = std::static_pointer_cast<LongColumnVector>(vector);
    if (!columnVector)
    {
        throw std::invalid_argument("Invalid vector type");
    }
    long* values;
```

```
    if(columnVector->isLongVectore()){
      values=columnVector->longVector;

    }else {
        values = columnVector->intVector;
    }

    int curPartLength;          //
    int curPartOffset = 0;      //
    int nextPartLength = size; //

    // 写入的 size 大于 pixelsStride
    // 需要申请一个新的 partition
    while ((curPixelIsNullIndex + nextPartLength) >= pixelStride)
    {
        curPartLength = pixelStride - curPixelIsNullIndex;
        // LongColumnvector->vector<long>
        writeCurPartLong(columnVector, values, curPartLength, curPartOffset);
        // vector<long>->ByteBuffer
        newPixel();
        curPartOffset += curPartLength;
        nextPartLength = size - curPartOffset;
    }

    curPartLength = nextPartLength;
    writeCurPartLong(columnVector, values, curPartLength, curPartOffset);
    // 写入的底层数据结果是 ByteBuffer 可以将其视为一个数组,
    // 其有两个下标 writepos 和 readpos
    // ? 少调用了一次 mewpixels()? 其实在 ColumnWriter::flush() 时调用
    return outputStream->getWritePos();
}
```

3.2 代码运行

在源代码的 cpp 目录下执行 make -j（建议根据 cpu 情况制定线程数），编译得到 pixels-cli（相对于源代码的 cpp 目录，位于 build/release/extension/pixels/pixels-cli/pixels-cli）。

大家可以创建一个简单的数据表，包含两列 int 类型的数据

```
0|0|
1|2|
2|4|
```

```
3|6|
4|8|
```

执行 pixels-cli，可以输入如下命令：

```
LOAD -o /home/pixels/data/input -t /home/pixels/data/output -s struct<a:int,
b:int> -n 10 -r \|
```

/home/pixels/data/input 代表我们输入文件目录，其中包含若干 *.tbl* 文件作为输入，/home /pixels/data/output 作为我们输出文件目录，而 struct<a:int,b:int> 就代表读取数据表的 schema，a:int 代表该列字段名为 a，字段类型为 int，b:int 同理。10 代表一个文件中保存的最多行数。\| 表示 *.tbl* 文件中所用的分隔符。

最终大家应该实现输出文件目录中包含转换的 *.pxl* 文件，并能被 pixels reader 成功读取。

3.3 实验任务

3.3.1 任务一

在父类 ColumnVector 中增加 add 虚函数定义，并在子类 DateColumnVector、DecimalColumnVector、TimestampColumnVector 中实现 add 方法和 ensureSize 辅助方法。关于该方法的代码解读，可参考 add 方法实现。

以上类的头文件在源代码的 cpp/pixels-core/include/vector/ 目录下，源文件在源代码的 cpp/pixels-core/lib/vector/ 目录下。

3.3.2 任务二

实现上述几种类型的 ColumnWriter，可以成功写入 *.tbl* 文件到 *.pxl* 文件中。以上类的头文件在源代码的 cpp/pixels-core/include/writer/ 目录下，源文件在源代码的 cpp/pixels-core/lib/writer/ 目录下。可以参考ColumnWriter 实现

3.3.3 任务三

实现写入文件可以被 duckdb 通过 pixels 正确读取，相关的测试文件可以参考 pixels-duckdb /examples/pixels-example，也可以直接运行 duckdb 进行测试，因为编译 duckbd 时，已经将 pixels-extension 链接了进去

3.3.4 附加任务

stringColumnWriter 的实现比较复杂，如果感兴趣可以尝试。

3.4 Tips

- 在实现代码中，可以类似 tests/writer 下的 PixelsWriterTest，分步测试是否正确。
- 使用 Clion 的 debug 功能进行代码调试
- 可以参考 pixels-java 的相关代码，但是部分函数用法不一样，需要注意

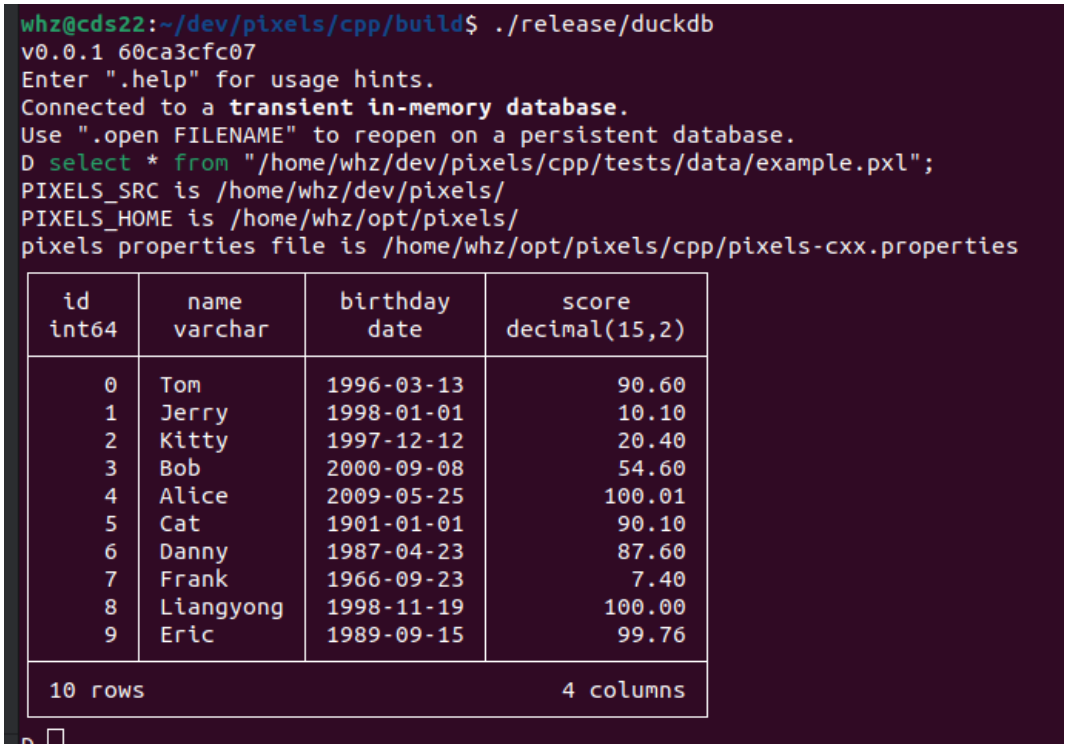

图 2: 使用 duckdb 进行测试