实用数据库开发实验 2 实验报告

刘一辰 2022201894

一、 实验目的

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 c1, c2, ..., cn）。 Row Group Footer 中存储了 Row Group 的元信息，包括行数、最大值、最小值等。本项目将.tbl 文件格式转换为.pxl 文件格式，并能通过 DuckDB 正确读取。

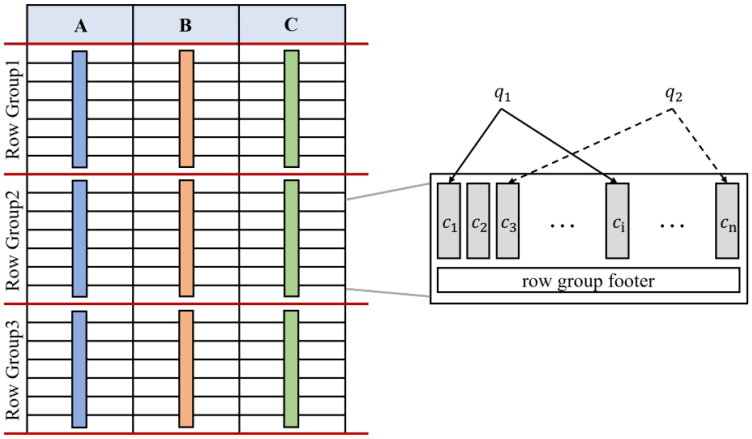


图 1: pixels 列存存储格式

注：本项目已在 GitHub 开源，项目链接为：https://github.com/lyc289/miniPixels

二、 实验过程

2.1 任务一：实现 Column Vector

本实验实现了 Date、Decimal、TimeStamp 和 String 四种数据类型对应的 Column Vector。所有的 Column Vector 都在构造函数中进行内存分配和编码器生成、在析构函数中进行内存释放等操作，并提供 current 方法返回当前元素、print 方法打印底层数据。在读取.tbl 文件后，writer 默认调用 add 方法的 string 重载版本写入数据，因此下面重点描述每个 Column Vector 实现的不同之处。

2.1.1 Date Column Vector

日期型数据的底层存储是 int*，在实现 add 方法时首先要把 string 类型的日期数据转为 int。在解析 string 中的年份、月份、日期后，我使用 std 库中的 std::tm 和 std::time_t 将日期转换为时间戳(单位为秒)，处理成单位为天后通过 static_cast 转换为 int。

由于时区不同，同样的日期通过 DuckDB 读出来会相差一天，因此在此手动将读入的数据天数+1，即可在转为 pxl 文件后通过 DuckDB 正确读写。

```cpp
int DateColumnVector::str2int(std::string &value)
{
    // 时区原因，输入数据需要对齐
    int year, month, day;
    if (sscanf(value.c_str(), "%d-%d-%d", &year, &month, &day) != 3)
    {
        throw std::invalid_argument("Invalid date format, should be year-month-day");
    }
    std::tm time_input = {};
    day+=1;
    normalize_date(year, month, day);
    time_input.tm_year = year - 1900;
    time_input.tm_mon = month - 1;
    time_input.tm_mday = day;
    std::time_t cur_timestamp = mktime(&time_input);
    if (cur_timestamp == -1)
    {
        throw std::runtime_error("Failed to convert date to timestamp");
    }
    return static_cast<int>(cur_timestamp / (60 * 60 * 24));
}
```

```cpp
// 对输入的时间修改day后 检查是否进位
void normalize_date(int& year, int& month, int& day)
{
    int days_in_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
    {
        days_in_month[1] = 29;
    }

    if (day > days_in_month[month-1])
    {
        day = 1;
        month += 1;
        if (month > 12)
        {
            month = 1;
            year += 1;
        }
    }
}
```

在 add 中调用此方法，将 string 型日期数据转换为 int，确保 vector 有充足空间后写入。

```cpp
void DateColumnVector::add(std::string &value)
{
    int days = this->str2int(value);
    if (writeIndex > length)
    {
        ensureSize(writeIndex * 2, true);
    }
    this->set(writeIndex, days);
    isNull[writeIndex - 1] = false;
}

void DateColumnVector::ensureSize(uint64_t size, bool preserveData)
{
    ColumnVector::ensureSize(size, preserveData);
    if (length < size)
    {
        int *oldvector = this->dates;
        posix_memalign(reinterpret_cast<void **>(&dates), 32, size * sizeof(int32_t));
        if (preserveData)
        {
            std::copy(oldvector, oldvector + length, dates);
        }
        delete[] oldvector;
        memoryUsage += (long)sizeof(int32_t) * (size - length);
    }
}
```

2.1.2 Decimal Column Vecotr

Decimal 数据的底层存储是 long*，为方便处理字符串获取 precision、scale 并根据指定的精度对数据进行 round，我构造了一个 StrDecimal 类进行封装，并提供了类似 Java 中 BigDecimal 类的 scale()、precision()、roundDecimal()和 longValue()方法。

```
class StrDecimal
{
private:
    std::string decimal;
public:
    StrDecimal(std::string &num):decimal(num) {}
    std::string roundDecimal(int target_scale);
    int precision();
    int scale();
    long longValue();
};
```

```
void DecimalColumnVector::add(std::string &value)
{
    if (writeIndex >= length)
    {
        ensureSize(writeIndex * 2, true);
    }
    // 转为指定scale
    StrDecimal decimal(value);
    if (decimal.scale() != scale)
    {
        decimal.roundDecimal(scale);
    }
    if (decimal.precision() > precision)
    {
        throw InvalidArgumentException("value exceeds the allowed precision");
    }

    int index=writeIndex++;
    vector[index]=decimal.longValue();
    isNull[index]=false;
}
```

在 add 方法中，确保输入的 precision 合法并将字符串转换到指定的 scale 后，转为 long 类型写入 vector 中的对应位置。add()方法的 double 重载版本和 ensureSize()方法如下所示：

```
void DecimalColumnVector::add(double value)
{
    if (writeIndex >= length)
    {
        ensureSize(writeIndex * 2, true);
    }

    std::string value_str=std::to_string(value);
    StrDecimal decimal(value_str);
    if (decimal.scale() != scale)
    {
        decimal.roundDecimal(scale);
    }
    if (decimal.precision() > precision)
    {
        throw InvalidArgumentException("value exceeds the allowed precision");
    }
    int index=writeIndex++;
    vector[index]=decimal.longValue();
    isNull[index]=false;
}
```

```cpp
void DecimalColumnVector::ensureSize(uint64_t size, bool preserveData)
{
    ColumnVector::ensureSize(size, preserveData);
    if (length < size)
    {
        long *oldVector = vector;
        posix_memalign(reinterpret_cast<void **>(&vector), 32,
                       size * sizeof(int64_t));
        if (preserveData)
        {
            std::copy(oldVector, oldVector + length, vector);
        }
        delete[] oldVector;
        memoryUsage += (long)sizeof(long) * (size - length);
        resize(size);
    }
}
```

2.1.3 Timestamp Column Vector

该类型的处理方法和 date column vector 类似，但底层存储是 long 类型的数组。在 add 前首先将输入的字符串转为 long，同时类似于 date column vector 要注意时区问题。

```cpp
long TimestampColumnVector::str2long(std::string &value)
{
    // 时区问题，实际时间要加18小时
    // 2005-10-01 23:58:19
    int year, month, day, hour, minute, second;
    std::istringstream ss(value);
    if (sscanf(value.c_str(), "%d-%d-%d %d:%d:%d", &year, &month, &day, &hour, &minute, &second) != 6)
    {
        throw std::invalid_argument("Invalid timestamp format, should be year-month-day hour:minute:second");
    }
    hour += 18;
    if (hour >= 24)
    {
        hour%=24;
        day++;
        normalize_date(year, month, day);
    }
    std::tm curtime = {};
    curtime.tm_year = year - 1900;
    curtime.tm_mon = month - 1;
    curtime.tm_mday = day;
    curtime.tm_hour = hour;
    curtime.tm_min = minute;
    curtime.tm_sec = second;
    std::time_t curtimestamp = mktime(&curtime);
    if (curtimestamp == -1)
    {
        throw std::runtime_error("Failed to convert to timestamp");
    }
    return static_cast<long>(curtimestamp) * 1000000;
}
```

```cpp
void TimestampColumnVector::add(std::string &value)
{
    long curTime = this->str2long(value);
    if (writeIndex >= length)
    {
        ensureSize(writeIndex * 2, true);
    }
    this->set(writeIndex, curTime);
    isNull[writeIndex - 1] = false;
}
```

2.1.4 Binary Column Vector

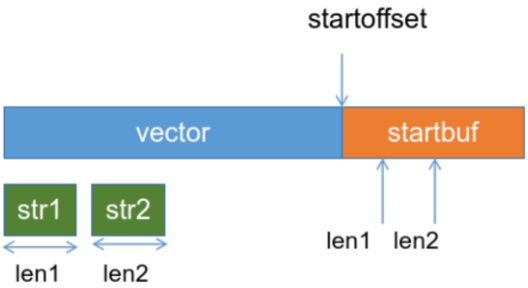


图 2: String 存储结构

Binary Column Vector 的实现较前几种更复杂，还需额外记录变长字符串的 start 和 len，成员变量如下：

```
private:
    int nextFree;
    int smallBufferNextFree;
    int bufferAllocationCount;
    float EXTRA_SPACE_FACTOR=1.2;
    int DEFAULT_SIZE = 1024; // for VectorizedRowBatch
    int DEFAULT_BUFFER_SIZE = 16*DEFAULT_SIZE;

public:
    int buffer_size;
    duckdb::string_t * vector;
    int* start;
    int* lens;
    uint8_t* buffer;
    uint8_t* smallBuffer;
    int buffer_size;
```

该类通过 setRef 将 uint8_t*类型的 buffer 数据写入 column vector，调用 duckdb::string_t 的构造函数实现：

```
void BinaryColumnVector::setRef(int elementNum, uint8_t * const &sourceBuf, int start, int length)
{
    if(elementNum >= writeIndex)
    {
        writeIndex = elementNum + 1;
    }
    this->vector[elementNum] = duckdb::string_t((char *)(sourceBuf + start), length);
    isNull[elementNum]=(sourceBuf==nullptr);
}
```

在 add 方法中，将输入的 string 数据写入 buffer，然后调用 setVal 写入数据：

```
void BinaryColumnVector::add(std::string &value) {
    size_t len = value.size();
    uint8_t* buffer = new uint8_t[len];
    std::memcpy(buffer, value.data(), len);
    add(buffer,len);
    delete[] buffer;
}

void BinaryColumnVector::add(uint8_t *v,int len) {
    if(writeIndex>=length)
    {
        ensureSize(writeIndex*2,true);
    }
    setVal(writeIndex++,v,0,len);
}
```

在 ensureSize 辅助方法中，扩大数组空间时要同时对记录 vector 内容属性信息的 start 数组和 lens 数组进行扩大：

```
void BinaryColumnVector::ensureSize(uint64_t size, bool preserveData)
{
    ColumnVector::ensureSize(size, preserveData);
    if (size > length)
    {
        int* oldStart=start;
        posix_memalign(reinterpret_cast<void **>(&start), 32, size*sizeof(uint32_t));
        int* oldLength = lens;
        posix_memalign(reinterpret_cast<void **>(&lens), 32, size*sizeof(uint32_t));
        duckdb::string_t* oldVector = vector;
        posix_memalign(reinterpret_cast<void **>(&vector), 32, size*sizeof(duckdb::string_t));
        if (preserveData)
        {
            std::copy(oldVector, oldVector+length, vector);
            std::copy(oldStart, oldStart+length, start);
            std::copy(oldLength, oldLength+length, lens);
        }
        delete[] oldStart;
        delete[] oldLength;
        delete[] oldVector;
        memoryUsage += (long)sizeof(uint32_t)*size*2;
        resize(size);
    }
}
```

2.2 任务二：实现 Column Writer

Column writer 的写法较为固定，注意根据 vector 底层存储的数组类型调用相关的工具即可。各个 writer 的实现思路大致都为将 vector 划分为当前 Pixel 和下一个 Pixel，用预计算消除 for 循环中的分支预测。各个 part 根据是否为 null 以不同的方式填充数据，由 newPixel()根据大小端调用不同的 encodingUtils 工具写入到 outputStream。

在 String column writer 中，由于加入了缓冲区机制，需要在缓冲区满足条件时调用 flush()写入文件。同时由于底层 vector 使用 duckdb::string_t 存储，而缓冲区大多使用 uint8_t*，需要将 duckdb::string_t 内部的字符指针通过 GetPointer()取出，得到 char*，再通过 reinterpret_cast 转为 uint8_t*.

下面以 decimal column writer 为例具体说明。

```
int DecimalColumnWriter::write(std::shared_ptr<ColumnVector> vector, int size)
{
    std::cout<<"In DecimalColumnWriter"<<std::endl;
    auto columnVector = std::static_pointer_cast<DecimalColumnVector>(vector);
    if (!columnVector)
    {
        throw std::invalid_argument("Invalid vector type");
    }
    long* values=columnVector->vector;
    int curPartLength;          // size of the partition which belongs to current pixel
    int curPartOffset = 0;      // starting offset of the partition which belongs to current pixel
    int nextPartLength = size; // size of the partition which belongs to next pixel

    // do the calculation to partition the vector into current pixel and next one
    // doing this pre-calculation to eliminate branch prediction inside the for loop
    while ((curPixelIsNullIndex + nextPartLength) >= pixelStride)// 0 1 2
    {
        curPartLength = pixelStride - curPixelIsNullIndex;
        writeCurPartLong(columnVector, values, curPartLength, curPartOffset);
        newPixel();
        curPartOffset += curPartLength;
        nextPartLength = size - curPartOffset;
    }

    curPartLength = nextPartLength;
    writeCurPartLong(columnVector, values, curPartLength, curPartOffset);

    return outputStream->getWritePos();
}
```

write()是实际调用的写函数，对数据进行分块读写。

```
void DecimalColumnWriter::writeCurPartLong(std::shared_ptr<ColumnVector> columnVector, long *values,
                                           int curPartLength, int curPartOffset)
{
    for (int i = 0; i < curPartLength; i++)
    {
        curPixelEleIndex++;
        if (columnVector->isNull[i + curPartOffset])
        {
            hasNull = true;
            if (nullsPadding)
            {
                // padding 0 for nulls
                curPixelVector[curPixelVectorIndex++] = 0L;
            }
        }
        else
        {
            curPixelVector[curPixelVectorIndex++] = values[i + curPartOffset];
        }
    }
    std::copy(columnVector->isNull + curPartOffset, columnVector->isNull + curPartOffset + curPartLength,
              isNull.begin() + curPixelIsNullIndex);
    curPixelIsNullIndex += curPartLength;
}
```

对该 part 进行读写，并处理 null 数组。

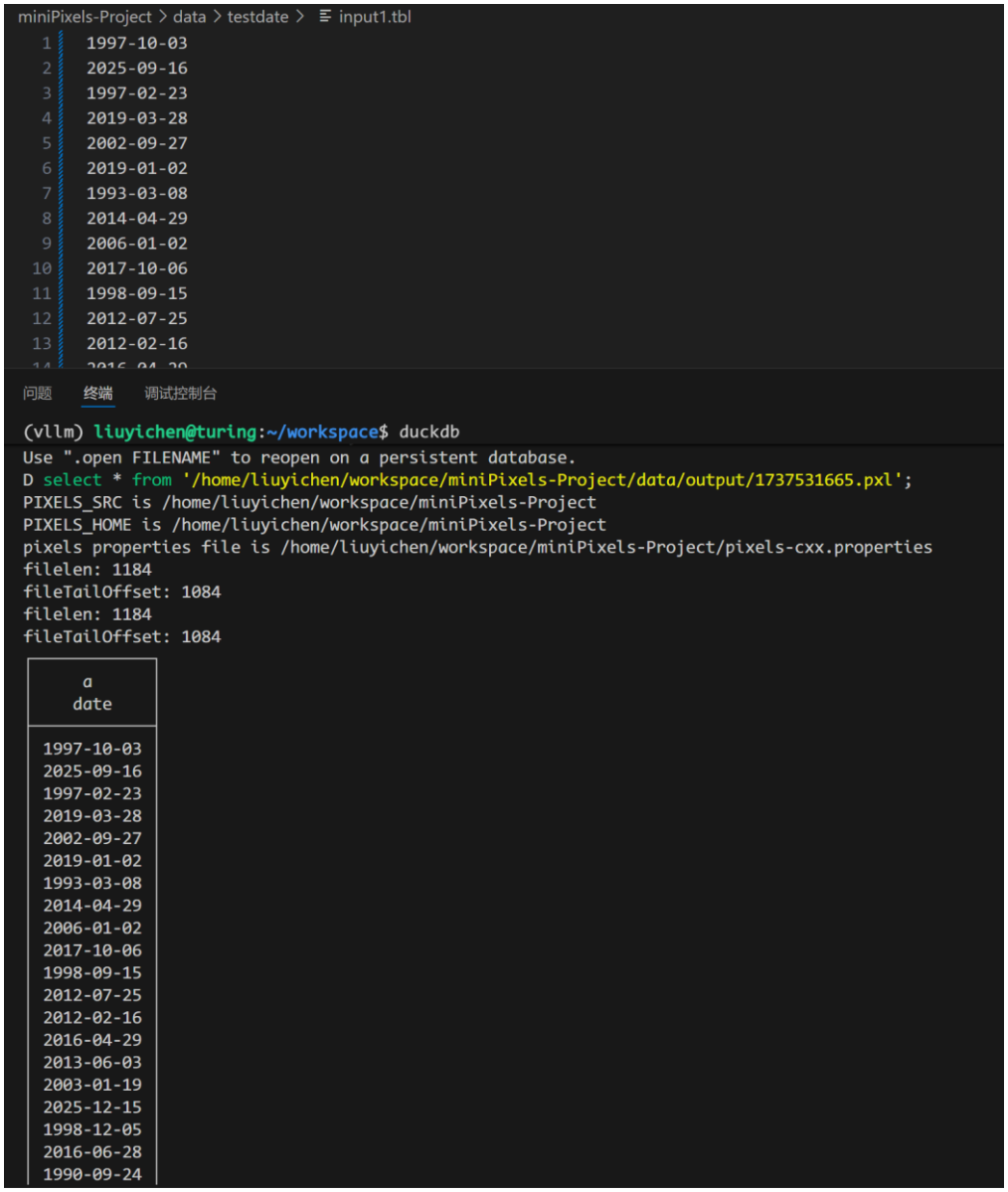
```
void DecimalColumnWriter::newPixel()
{
    std::shared_ptr<ByteBuffer> curVecPartitionBuffer;
    EncodingUtils encodingUtils;
    curVecPartitionBuffer = std::make_shared<ByteBuffer>(curPixelVectorIndex * sizeof(long));
    if (byteOrder == ByteOrder::PIXELS_LITTLE_ENDIAN)
    {
        for (int i = 0; i < curPixelVectorIndex; i++)
        {
            encodingUtils.writeLongLE(curVecPartitionBuffer, curPixelVector[i]);
        }
    }
    else
    {
        for (int i = 0; i < curPixelVectorIndex; i++)
        {
            encodingUtils.writeLongBE(curVecPartitionBuffer, curPixelVector[i]);
        }
    }
    outputStream->putBytes(curVecPartitionBuffer->getPointer(), curVecPartitionBuffer->getWritePos());
    ColumnWriter::newPixel();
}
```

根据大小端情况调用不同的工具写入 outputStream。

三、 结果验证

将上述代码编译后执行 pixels-cli，通过 LOAD 指令将测试.tbl 文件转为.pxl 文件，并通过 DuckDB 读取。结果如下：

1. Date



2. Decimal

miniPixels-Project > data > test_decimal > test_decimal.tbl

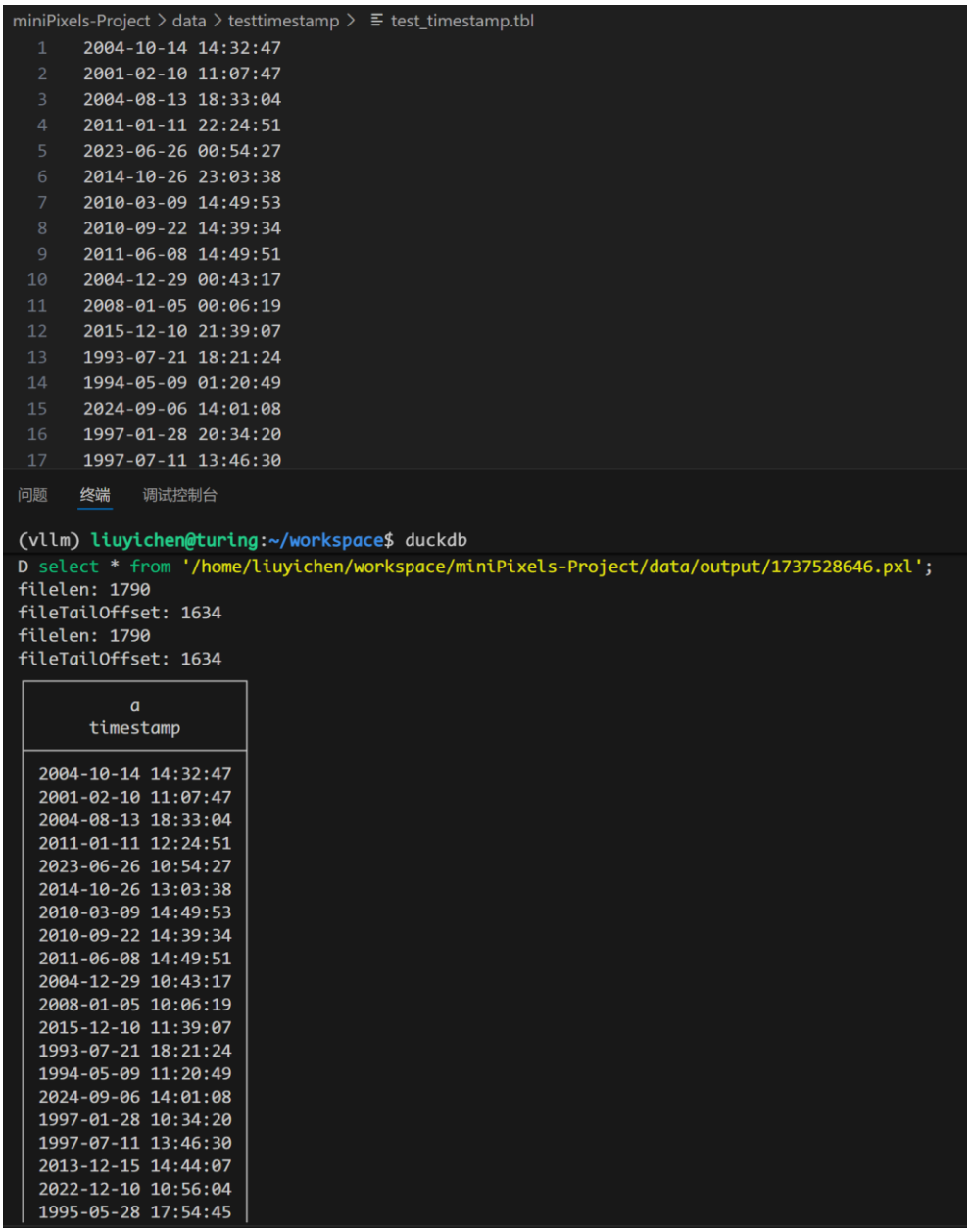
```
1   34476661.13
2   9581641.68
3   97486758.64
4   2451781.32
5   330310.41
6   28381028.71
7   36915791.06
8   78169018.65
9   25024491.12
10  64608051.20
11  75169058.60
12  32558787.76
13  59501512.29
14  51011771.52
15  47060971.03
16  24526058.90
17  19731170.12
```

问题 终端 调试控制台

```
(vllm) liuyichen@turing:~/workspace$ duckdb
D select * from '/home/liuyichen/workspace/miniPixels-Project/data/output/fulldec.pxl';
filelen: 1787
fileTailOffset: 1634
filelen: 1787
fileTailOffset: 1634
```

a decimal(15,2)
34476661.13
9581641.68
97486758.64
2451781.32
330310.41
28381028.71
36915791.06
78169018.65
25024491.12
64608051.20
75169058.60
32558787.76
59501512.29
51011771.52
47060971.03
24526058.90
19731170.12
86239561.90
74720663.56
82939319.01

3. TimeStamp



四、 总结

在该实验中，通过实现 ColumnVector、ColumnWriter 并完成从.tbl 到.pxl 文件的转换，我加深了对列存格式的底层存储的理解，学会了使用 cmake 编译大型项目、用 lldb 调试大型项目，增强了配环境、动手开发和 debug 的能力。