大数据原理与技术 平时作业 3

21311274 林宇浩

一、算法说明

当我们需要处理大规模数据流时，如网络流量监控、社交媒体数据分析等，传统的方法可能会因为数据量过大而难以处理。DGIM 算法便是为了解决这类问题而设计的。

DGIM 算法主要解决的问题是在数据流中近似计算某个特定元素出现的频率。典型的场景是，我们需要知道在过去一段时间内某个特定事件发生的次数，但是数据量太大无法存储完整的历史数据，也无法对每个事件都进行实时计数。

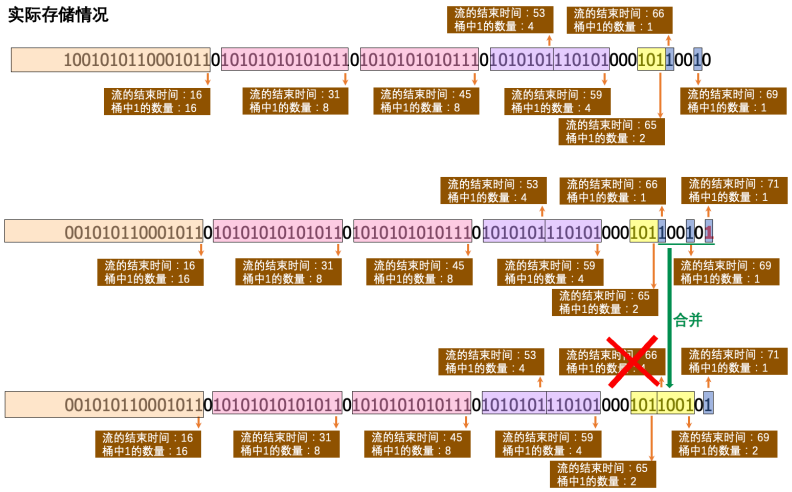
算法的核心思想是利用窗口和桶来进行数据的压缩和近似计算。具体来说，DGIM 将数据流划分为长度递增的时间窗口，并在每个窗口内维护一组桶。每个桶都有一个时间戳，记录了桶中的 1 的数量以及该 1 的最后出现时间距离当前时间的距离。例如，假设我们有一个长度为 10 的时间窗口，每个窗口内有不同数量的桶。在一个新的数据流到达时，我们将根据当前时间戳创建一个新的桶，并将新的数据写入到桶中。然后，我们会检查窗口内的桶是否超出了预设的数量限制，如果超出了，则会将较老的桶进行合并。合并的过程是将两个相邻的桶合并成一个新的桶，同时将合并后的桶的时间戳设置为较新的桶的时间戳。在合并时，会保留合并后的桶中的 1 的数量，并更新距离当前时间的距离。这样一来，我们可以通过这些桶来估计在过去某个时间段内某个特定元素出现的频率。

需要注意的是，由于桶的数量和大小是有限的，所以算法会引入一定的误差。但是，通过合理设置窗口的大小和桶的数量，我们可以控制误差的范围，并在保证存储空间较小的情况下获得较好的近似计算结果。

二、算法改进

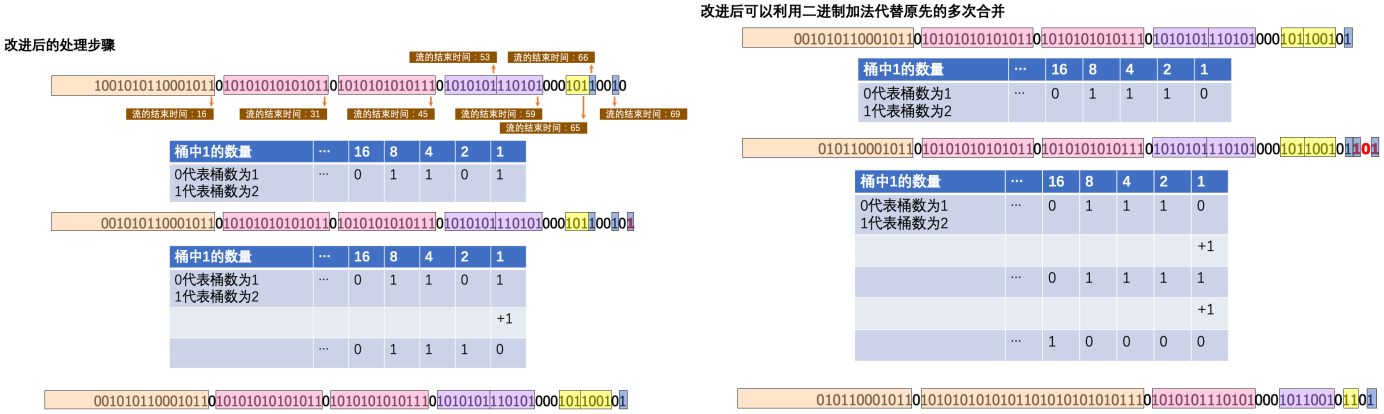
1、对存储桶数量的方式的改进

原 DGIM 算法中，桶的存储情况如下所示，对于每个桶需要维护其对应的时间戳以及桶的尺寸：

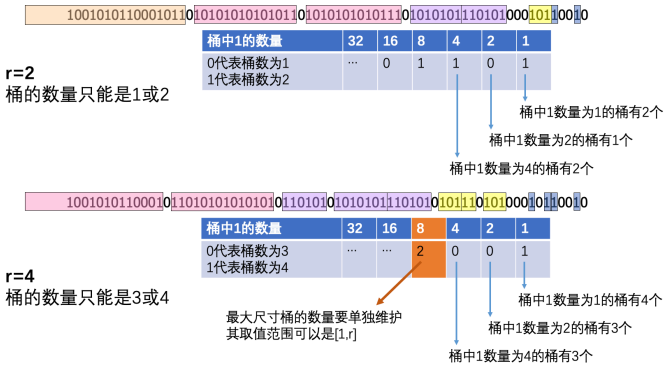


当数据流中有新的 1 到达时，我们为其创建一个尺寸为 1 的新桶，并记录时间戳。当某尺寸的桶的数量超过 r 时，我们将这个尺寸中最旧的两个桶进行合并，即将要合并的两个桶中更旧的桶删除，将更新的桶的尺寸扩大一倍，从而满足桶的数量只能为 r 或 r-1 的要求。

对存储桶数量的方式的改进如下图所示，桶的数量存储在一个 bool 值列表中，列表第 i 位代表桶的尺寸为2^i，元素值为 1 代表对应尺寸桶的数量为 r，元素值为 0 代表对应尺寸桶的数量为 r-1。当数据流中有新的 1 到达时，只需要将 bool 数组整体作为一个二进制数进行递增操作即可，可以看到递增操作能正确反映出桶数量的变化。同时当发生级联合并时，改进后结构能够更加方便地被维护。级联合并指的是小尺寸桶的合并导致大尺寸桶的数量超过 r，从而要对大尺寸的桶继续进行合并，从而还可能造成更大尺寸桶的数量超过 r，引发连锁反应。对于级联合并，改进后的结构不需要安排特殊的扫描操作，其合并直接体现在了二进制数的进位中，一次进位即代表了一次合并操作。而二进制加法又是计算机底层非常支持的运算，通过使用 bool 数组和二进制加法能够极大提高维护桶结构的效率。

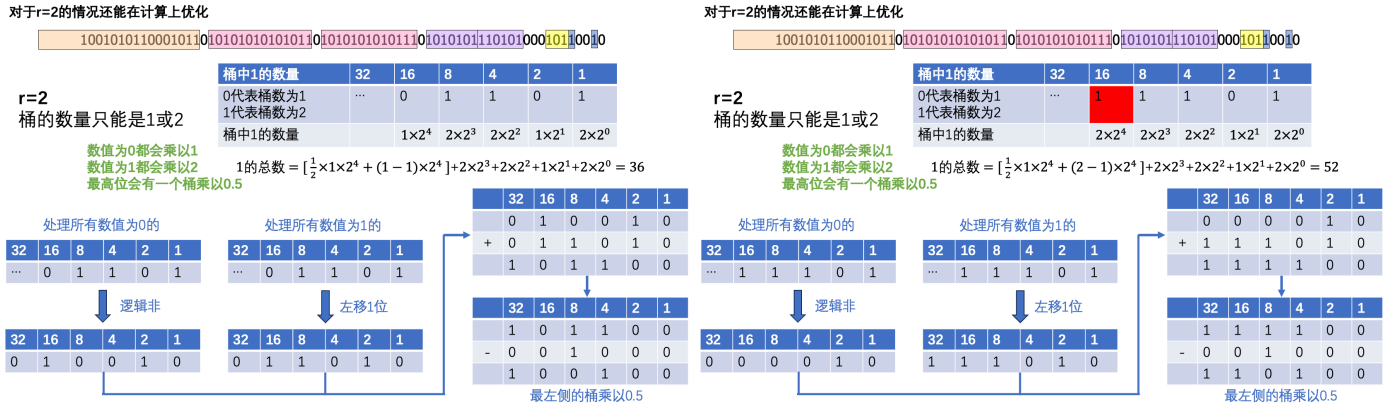


这里需要注意的是当 r 值不等于 2 时，最大尺寸桶的数量的取值范围可以是[1, r]，不能只用两个状态进行表示，所以我们可以将最大尺寸桶的数量单独维护，如下图所示。



2、对 1 数量估计值查询操作的改进

当查询操作不频繁时，在 r 等于 2 的情况下可以通过二进制位运算对查询操作进行加速，如下图所示。窗口中 1 数量的估计方式为将所有桶中 1 的数量全部相加，但由于最旧的桶的起始位置可能已经超出窗口边界，所以对于最旧的桶，在相加时我们只取其数量的一半进行估计，如下图中的公式所示。可以看到公式中的数据均是 2 的倍数，所以我们可以将乘 1 和乘 2 运算都用位运算来进行，然后再用补码加法来减掉最大桶的一半，最后即可正确得到窗口中 1 数量的估计值，这相比进行乘法和幂运算减少了很多操作。



在本次实验中窗口每达到一个新的数据都要进行一次估计，这时即可使用一个变量对窗口中 1 的总数进行长期维护，当到达新的 1 时将变量的值递增，当有桶被丢弃时将变量的值则减去桶的尺寸。这样能够节省一些多余的操作。

3、对存储时间戳的方式的改进

时间戳是改进后的 DGIM 算法中主要的空间复杂度来源，对时间戳维护的开销占了很大一部分。在 DGIM 算法中我们可以发现，合并桶时总是合并的是最旧的两个桶，然后形成一个更大的新桶，而其他的桶则不会受到影响，这让我们联想到队列的操作，我们可以为每个尺寸都维护一个时间戳队列，当队列中元素的数量超过 r 时，即可将

队尾的两个元素出队，然后为更大尺寸的队列进行一次入队操作，要入队的元素即是第二个出队的元素。通过使用队列的结构，可以更加高效地对时间戳进行维护。

但是使用队列也有维护队列的开销，由于特定尺寸桶的数量不会超过 r，所以我们可以使用长度固定的静态数组来实现循环队列。更进一步，通过循环队列的入队出队具有的数学规律，可以对循环队列的结构进行进一步的改进，改进后如下所示。

```
bool    桶数    倒计时  时间戳队列              总数：0
-1      0       1       0 0 0

bool    桶数    倒计时  时间戳队列              总数：1
-1      1       1       0 0 1

bool    桶数    倒计时  时间戳队列              总数：2
-1      2       1       0 1 2

bool    桶数    倒计时  时间戳队列              总数：3
-1      3       1       1 2 3

bool    桶数    倒计时  时间戳队列              总数：4
0       2       1       2 3 4
-1      1       2       0 0 2

bool    桶数    倒计时  时间戳队列              总数：5
1       3       1       3 4 5
-1      1       1       0 0 2

bool    桶数    倒计时  时间戳队列              总数：6
0       2       1       4 5 6
-1      2       2       0 2 4

bool    桶数    倒计时  时间戳队列              总数：7
1       3       1       5 6 7
-1      2       1       0 2 4

bool    桶数    倒计时  时间戳队列              总数：8
0       2       1       6 7 8
-1      3       2       2 4 6
```

```
bool    桶数    倒计时  时间戳队列              总数：8
0       2       1       6 7 8
-1      3       2       2 4 6

bool    桶数    倒计时  时间戳队列              总数：9
1       3       1       7 8 9
-1      3       1       2 4 6

bool    桶数    倒计时  时间戳队列              总数：10
0       2       1       8 9 10
0       2       2       4 6 8
-1      1       4       0 0 4

bool    桶数    倒计时  时间戳队列              总数：11
1       3       1       9 10 11
0       2       1       4 6 8
-1      1       3       0 0 4

bool    桶数    倒计时  时间戳队列              总数：12
0       2       1       10 11 12
1       3       2       6 8 10
-1      1       2       0 0 4

bool    桶数    倒计时  时间戳队列              总数：13
1       3       1       11 12 13
1       3       1       6 8 10
-1      1       1       0 0 4

bool    桶数    倒计时  时间戳队列              总数：10
0       2       1       12 13 14
0       2       2       8 10 12
-1      1       4       0 4 8

bool    桶数    倒计时  时间戳队列              总数：11
1       3       1       13 14 15
0       2       1       8 10 12
-1      1       3       0 4 8
```

这里展示了窗口长度为 10，r 等于 3 情况下改进后结构的运行情况，为了更直观的展示，输入数据使用了全 1 数据流。可以看到，尺寸为 1 的桶，每次都进行更新；尺寸为 2 的桶，每 2 轮更新一次；尺寸为 4 的桶，每 4 轮更新一次，可以从上图的更新倒计时处详细观察其运行情况。通过这个数学规律，我们可以通过 bool 数组尾部 0 的数量来判断哪些桶要进行更新，如果 bool 数组尾部有 1 个 0，则需要更新最小 2 个桶的时间戳队列，如果 bool 数组尾部有 2 个 0，则需要更新最小 3 个桶的时间戳队列，以此类推。这样我们就不需要维护循环队列的长度了，通过 bool 数组即可知道各个队列什么时候需要更新。同时我们也不需要维护循环队列的尾指针了，在长度为 r 的数组中我们可以刚好保留要传递给下一层级队列的时间戳数据，除最小桶的时间戳队列外，其他桶每次更新只需要直接将上一层级桶尾部的数据入队即可，我们不需要再执行出队操作，需要出队的元素会在长度为 r 的数组中由于入队操作而被自动丢弃，数学规律使得保留到更新时机的元素刚好是下一层级的桶所需要的。这样子要维护的变量和操作得到减少，要维护的变量就只有一个 bool 数组，一个 O(logN)×r 大小的矩阵用于存储队列，以及一个整型数组作为队列的头指针。要维护的操作只有根据 bool 数组尾部 0 的数量来对整形数组进行递增。

三、展示

r 值	运行时间(s)	平均相对误差	方差	最大桶数
2	17.4288	10.451%	92171148390.12	48
4	17.2853	5.2605%	23007178656.39	88
8	17.4001	2.62846%	5739906824.29	168
16	17.3263	1.31304%	1430829665.79	320
32	17.3716	0.655147%	358042273.88	608

实验中窗口大小设为了一千万，数据流长度为五千万。这里方差的计算没有使用相对误差，而是使用了绝对误差进行统计。

可以看到，随着 r 值的增加，平均误差得到减小，可以证明误差最大不超过$O(\frac{1}{r})$，所以 r 值越大平均误差越小。同时，r 值越大方差也越小，这与平均误差变小相符合，另外 r 值越大桶数越多，最大桶的尺寸越小，这限制了估计值和真实值的偏差范围，使得方差得到控制，这也是最大误差不超过$O(\frac{1}{r})$结论的原因。

对于运行时间，可以看到随着 r 值的上升，桶数也在上升，但是运算时间基本不变，这是由于我们结构的维护消耗不受桶数影响所导致的，所以我们能够在运行时间不变的情况下，通过增大 r 值使得平均相对误差下降。这本

质上是通过空间换精度，最大桶数的计算公式为$r \times \left\lfloor \log_2\left(\frac{N}{r-1}+1\right)\right\rfloor$，可以看到最大桶数的增长是比线性增长要慢的，而且每个桶实际的存储成本只需一个数字用于记录时间戳。假设我们的窗口长度为$2^{32}-1=4294967295$，即 unsigned int 的最大值，十亿量级，这样每个桶的时间戳用一个 unsigned int 足够存储，如果我们设置 r 值为 1000，则最大桶数为 23000，所需空间约为 0.176 MB，这对于如今的电脑内存也并不是一个很大的数字。

r 值	窗口大小 N	运行时间(s)	平均相对误差	最大桶数
4	10^7	17.2853	5.2605%	88
4	10^6	17.6227	5.33647%	64
4	10^5	17.2868	4.0974%	48
4	10^4	17.0597	5.12047%	36

上表对窗口大小 N 变化的影响进行了统计，数据流长度均保持五千万。可以看到，运行时间仍然基本保持不变，主要的影响仍然是空间消耗。

r 值	数据流长度	运行时间(s)	平均相对误差	最大桶数
4	5*10^7	17.0597	5.12047%	36
4	5*10^6	1.66402	5.1192%	36
4	5*10^5	0.17088	5.11869%	36

上表对数据流长度变化的影响进行了统计，窗口大小 N 均保持为 10^4。可以看到，运行时间和数据流长度成正比，二者是线性关系。我们可以发现，DGIM 算法有良好时间性能，运行时间基本只受数据流长度的影响，r 和 N 的取值对运行时间影响不大，主要影响的是空间消耗。所以对运行时间的改进是可以持续受益的，而对于空间占用，时间戳本质上是在变相记录数据流中 0 的数量，O(logN)的量级想要进一步改善有一定难度，需要从数学上考虑能否进一步降低，但是进行优化是可以的，比如 DGIM 论文中提到的将时间戳只取 2 的倍数，这会增加一定的误差，但是能够节省空间开销。目前还有一些并行化的想法在考虑中，由于时间有限，将留到大作业中进一步考虑。

四、代码说明

程序首先定义了 CircularArray 类用于实现改进后的循环队列。然后定义了 Buckets 类，使用循环队列存储时间戳，并使用 int 型变量维护了一个大小为 R 的 bool 数组来记录每个桶中 1 的数量。主函数中，程序读取数据流中的每个比特，根据出现的 1 更新桶的状态，丢弃最老的桶，并计算估计值和真实值之间的误差。具体详见代码注释。

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <cmath>
#include <chrono>
using namespace std;
#define R 4

int calculate_array_num(double window_size) // 计算窗口中能出现的桶的最大尺寸
{
    double array_num = log((window_size / (R - 1.)) + 1.) / log(2);

    return (int)ceil(array_num);
}

class CircularArray // 自动队列
{
public:
    int arr[R];
    int head;

    CircularArray() : head(0)
    {
        for (int i = 0; i < R; ++i)
        {
            arr[i] = 0;
        }
    }
```

```
    void push(int num) // 入队+出队
    {
        arr[head] = num;
        head = (head + 1) % R;
    }

    int tail() // 获取队尾
    {
        return arr[head];
    }

    void display()
    {
        for (int i = 0, index = head; i < R; i++, index = (index + 1) % R)
        {
            cout << arr[index] << " ";
        }
        cout << endl;
    }

    int nth_num(int n) // 获取队首第 n 个元素
    {
        return arr[(head + R - n) % R];
    }
};

class Buckets
{
public:
    CircularArray *buckets;
    int bucket_num_counter; // bool 数组
    int total;
    int biggest_bucket_size;
    int biggest_bucket_num;
    int biggest_bucket_index;

    Buckets(int window_size) : bucket_num_counter(0), total(0), biggest_bucket_size(1),
biggest_bucket_num(0), biggest_bucket_index(0)
    {
        int array_num = calculate_array_num(window_size);
        buckets = new CircularArray[array_num];
    }

    void arrive_1(int time_stamp) // 出现新的 1 进行更新
    {
        total++;
        bucket_num_counter++;
        if (bucket_num_counter == biggest_bucket_size) // 进位到最大桶的情况
        {
            bucket_num_counter = 0;
            biggest_bucket_num++;
            if (biggest_bucket_num == R + 1) // 最大桶也进位的情况
            {
                biggest_bucket_size *= 2;
                biggest_bucket_index++;
                biggest_bucket_num = 1;
            }
        }

        buckets[0].push(time_stamp);
        for (int i = 1, temp_counter = bucket_num_counter; i <= biggest_bucket_index; i++)
        {
            if ((temp_counter & 1) == 0) // bool 数组末尾是连续的 0 时进行桶合并
            {
                buckets[i].push(buckets[i - 1].tail());
            }
            else
            {
                break;
            }
            temp_counter >>= 1; // 右移一位
        }
    }

    void drop_bucket()
```

```
    {
        total -= biggest_bucket_size;

        if (biggest_bucket_num == 1 && biggest_bucket_size > 1) // 最大桶尺寸下降的情况
        {
            biggest_bucket_size /= 2;
            biggest_bucket_index--;

            if (bucket_num_counter & biggest_bucket_size) // 原本第二大的桶变成新的最大桶
            {
                biggest_bucket_num = R;

                // 把原本最大桶在 bool 数组中对应的值改为 0
                int mask = ~(1 << biggest_bucket_index);
                bucket_num_counter = bucket_num_counter & mask;
            }
            else
            {
                biggest_bucket_num = R - 1;
            }
        }
        else
        {
            biggest_bucket_num--;
        }
    }

    int oldest_time_stamp()
    {
        if (biggest_bucket_num == 0)
        {
            return -1; // 如果没有桶返回-1
        }
        return buckets[biggest_bucket_index].nth_num(biggest_bucket_num);
    }

    int num_of_1() // 1 数量的估计值
    {
        return total - (biggest_bucket_size / 2);
    }
};

int main()
{
    // ifstream input_data("平时作业 3 测试样例/TestDataStream.txt"); // 打开实验数据
    // ifstream input_data("DataStream.txt"); // 打开实验数据
    ifstream input_data("DataStream_10000_50000.txt"); // 打开实验数据
    // ifstream input_label("平时作业 3 测试样例/Groundtruth.txt");
    // ifstream input_label("GroundTruth.txt");
    ifstream input_label("GroundTruth_10000_50000.txt"); // 打开实验数据
    ofstream output_file("error.txt");

    if (!input_data)
    {
        cerr << "无法打开文件! " << endl;
        return 1;
    }

    bool bit;
    int groundtruth;
    int timestamp = -1;
    int window_size = 10000;
    int i = 1, data_num = 0;
    int estimate;
    double error, total_error = 0;
    Buckets buckets(window_size);

    chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();
    while (input_data >> bit)
    {
        timestamp = (timestamp + 1) % window_size; // 时间戳

        if (buckets.oldest_time_stamp() == timestamp && buckets.oldest_time_stamp() != -1)
        {
            buckets.drop_bucket(); // 丢弃末尾的桶
        }
```

```
        if (bit)
        {
            buckets.arrive_1(timestamp); // 出现新的 1 更新桶
        }

        if (i == window_size)
        {
            input_label >> groundtruth;
            data_num++;

            estimate = buckets.num_of_1(); // 窗口内 1 数量的估计值
            error = (double)abs(groundtruth - estimate) / groundtruth * 100.0;
            total_error += error;
            // output_file << fixed << (double)abs(groundtruth - estimate) << endl;
        }
        else
        {
            i++;
        }
    }

    chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = std::chrono::duration_cast<std::chrono::duration<double> >(end -
start); // 计算程序运行时间（秒）
    std::cout << "运行时间: " << duration.count() << " 秒" << std::endl;

    cout << "误差: " << total_error / data_num << "%" << endl;

    input_data.close(); // 关闭文件
    input_label.close();
    output_file.close();
}
```