Efficient String Sort with Multi-Character Encoding and Adaptive Sampling

Wen Jin , Weining Qian , Aoying Zhou

摘要

排序在计算机科学中扮演着至关重要的角色，它在数据库操作和数据科学任务中具有较为广泛的应用。其中一类重要的排序主要是针对字符串，在现有的字符串排序方法中，基数排序是一种既简单又有效的方法，有很多研究人员通过开展许多相关工作来加速字符串的基数排序，其中一种典型的方法是在每个排序过程中处理多个字符，然而，这种方法会导致基数过大的关键问题。为了解决这一问题，在该论文中引入了一种新的基于多字符编码的方法，该方法可以显著降低基数，这种新的编码方法利用了稀疏字母表在空间上的稀疏性，在该基础上将有效的编码方案与自适应采样过程相结合，进而生成编码，该论文提出的排序算法本质上混合了基数排序与样本排序，并且相对于其他排序方法而言取得了实质性的改进。在真实数据集和合成数据集上的结果表明，该论文中提出的方法比 C++ STL[1]排序算法的平均性能提高了 4.85 倍，在多核环境下的初步测试也表明该方法具有较大竞争力，甚至优于最新的并行字符串排序算法，证明了该方法的可扩展性。

关键词：排序；基数排序；样本排序；字符串

1 引言

排序在计算机科学中起着至关重要的作用，它是许多数据库操作和数据分析问题的核心，高性能排序对于数据库系统中的索引创建、表连接等至关重要，并且更快的算法工具（如排序算法）在一定程度上可使得数据分析系统具有更好的响应性能和可扩展性。除了数字数据外，字符串在实际应用中也很普遍，通常可以使用比较常规的算法（如 Quicksort 或 Mergesort）对字符串进行排序，但是随着字符串长度的增加，这些算法的时间复杂度可能很高。

最著名和最快的字符串排序方法是 MSD（最高有效数字优先）基数排序，基数排序具有良好的渐进性。为加快字符串基数排序的工作，一种典型的方法是在每次排序过程中处理多个字符，但是这种方法会带来基数过大的问题，为解决该问题，引入了一种新的基于多字符编码的方法，该方法可以显著减少基数。

本次复现的论文提出了一种新的排序算法，该算法基于多字符编码（MCE）方案，可以一次处理多个字符，并将它们编码为超级字符（SC），为有效地生成编码，该算法将有效的编码方案与自适应采样过程相结合，本质上是将基数排序与样本排序相结合，从而进一步提高字符串基数排序性能和降低 MCE 的预处理成本。

2 相关工作

根据是否使用成对比较的方式确定待排序元素顺序，可以将排序算法分为比较类算法和非比较类算法，现有的大多数算法属于前者，而后者的典型算法有基数排序和计数排序，理论上，这两种类别都可以用于字符串排序。

2.1 顺序基数排序

McIlroy, Bostic 和 McIlroy[2]针对不同的基数排序进行开创性的研究，在他们的研究中提出了较好的具体实施方案，待排序元素的分布可以通过将该元素复制到第二个临时数组的正确位置，也可以通过使用一种名为“American flag sort”的算法对输入数组中的元素进行置换进而实现排序功能。

Paige 和 Tarjan[3]提出了第一个基于迭代前缀优化的二进制字符串基数排序算法。Daniel 等[4]专注于数据局部性的改进，他们提出了 CC-Radix 排序算法，它在排序之前将数据动态地划分为适合 L2 缓存的子集。

虽然基数排序算法复杂度低，但其具有不规则的内存访问模式，导致缓存丢失，它也不能很好地利用现代计算机体系结构，因为 cpu 比主存快得多，并且字符串的长度可以改变。Ng 和 Kakehi[5]提出了一种名为 CRadix 的缓存变体，它可以一次从字符串中获取 m 个字符，而不是每次采用递归的方式获取需要的单个字符，预获取的字符存储在与字符串指针数组对齐的“缓存”缓冲区中。Kärkkäinen 等[6]对字符串排序进行了全面的实验研究，它包含字符的缓存步骤，在计算步骤中只缓存一个字符，这样在重新分配时就不需要再次读取字符。除此之外，它还利用现代硬件提供的内存延迟隐藏特性，允许处理器同时发出多个独立的读取请求来访问内存，在 for 循环中实现的计数过程包括检索字符和增加字符计数器，它分为两个循环，每个操作一个循环，因此它不需要在读取下一个字符串之前等待计数器更新完成。

Ska 排序[7]利用指令级别的并行性来加快原排列过程，这使得它比 STL 排序更快。Wassenberg 和 Sanders[8]采用为存储桶分配大量虚拟内存的方法，该方法利用了虚拟内存，而这些虚拟内存实际上并没有为其分配物理内存，只有当待排序元素被分配到桶中时，系统才会为其分配内存。

2.2 其他顺序字符串排序算法

LCP-Mergesort[9]通过保存和重用排序子问题中字符串的最长公共前缀 (LCP) 来实现字符串的归并排序，它在最坏情况下时间复杂度是 O(D + n logn)。多字符-快排[10]是快速排序对字符串的应用，当所有字符串都有一个长度为 h 的公共前缀时，它根据公共前缀后面的剩余字符将字符串分割为三个分区 (中间的分区是相等的分区)，在理想情况下其时间复杂度为 O(D + n logn)。样本排序[11]是快速排序的一种泛化形式，它将输入内容分成许多块，也可应用于字符串排序[12]。Super Scalar String Sample Sort (S5)[13]算法通过二叉搜索树将待排序字符串分类并分配到存储桶中，在分类步骤利用了字符串的公共前缀[14]。尽管 std::sort 不是专门为字符串设计的，但它是 c++ STL 库[15]中提供的一种流行的排序工具，它使用了 Introsort[1]，这是一种优化的快速混合排序，当递归深度超过一定程度时可以切换到堆排序，当元素数量很少时切换到插入排序。

2.3 并行排序

pS5[16]是最新的字符串并行排序算法。它是算法 S5 的并行化版本，是为现代多核系统的新特性而专门设计，在现代多核系统中，许多处理器都有各自的缓存级别，但共享 RAM 的内存通道相对较少且速度较慢。PARADIS[17]和 Regions Sort[18]侧重于并行基数排序，前者首先是初始化推测排列，然后是进入修复阶段，以解决在排列时存在的读写依赖关系，而后者依赖于一个图结构来解决依赖关系，它们的实现仅用于对整数排序。

3 本文方法

3.1 本文方法概述

实现字符串快速排序的理想方法是将每个字符串转换为用于排序的数字，为避免存在各字符串长度不相同的情形，最简单的解决方案是在长度较短的字符串的末尾使用“0”进行填充，这样便可使得所有的字符串长度相同，然后每个字符串都将编码成一个（可能是超级大的）整数，很明显每个字符的基数是 256（用一个字节表示），而且由于需要比较的字节数没有减少，所以它并没有提供任何优势[19]。一种有效的方法是将每个字符串编码成一个更小的数字。

在本文中引入了一种新的多字符编码方案（MCE），该方案可以一次性处理多个字符，并将它们编码成一个超级字符（SC），利用后缀字符串的上下限将字符串/子字符串编码到一个紧凑的范围内，然后将多字符编码（MCE）方案应用于基数排序，该方案解决了排序过程中基数过大的问题[20]，并且基于保序性验证了其排序的正确性。然而任何用于排序的编码函数都应当是一个保序编码函数[21]。

3.2 保序编码函数定义

给定一组字符串 S 和一个用于字符串排序的编码函数 Enc，只有当满足以下条件时 Enc 函数才是一个保序编码函数：对 S 中任意的子字符串 S1 和 S2，若按照字典顺序 S1、S2 满足以下关系：S1≤S2，则使用 Enc 编码函数分别对其编码后也应当存在以下关系：Enc（S1）≤ Enc（S2），反之亦然[22]。

要实现对编码后的字符串进行排序，编码方案应当简单，这样有利于快速计算出各字符串的编码值，基本原则是将字符串中的每个字符转换为整数，然后编码函数再对转换后的数字进行一定的操作。该论文首先介绍了一种基础的编码方案 BMCE（the Basic Multi-Character Encoding scheme），然后提出了一种改进的编码方案 EMCE（the Enhanced Multi-Character Encoding scheme）。

3.3 BMCE 编码方案

EMCE（the Enhanced Multi-Character Encoding scheme）是一种直接依赖于字母表大小的编码方案，由于位置 i 处的字母表大小（即来自一组字符串 S 中各子字符串位置 i 处独一无二的字符数量）仅仅是对单个字节为单位进行操作，这使得在基数排序的每个递归阶段引入了固定的成本，并且效率低下。因此可利用字母表的稀疏性来针对某一位置处的字母表大小创建更小的基数，然后，该字符串可以被视为混合基数[23]。在此采取一种新的措施，将唯一字符（不重复的字符）存储在 ULi（一个升序排列的字符列表）中的位置 i 处，首先可以使用函数 hi（ ）将 ULi 中的每个字符转换成该字符所对应的的索引号，即 c→[0,ULi-1], 其中字符 c ∈ ULi 字符列表，特别地：hi (ULi [j]) = j，同样把字符串组 S 中各子字符串位置 i 处唯一字符所构成的各 ULi 升序字符列表的单位值定义如下（同[24]）：

$$Zi = \prod_{t=i+1}^{L-1} | UL_i | \quad 0 \leq i < L-1$$

其中 L 是指 ULi 字符列表的总个数，也是 S 中最长的字符串的长度。特殊地，将最后一个 UL_{L-1} 对应的 Z_{L-1} 定义为 Z_{L-1}=1, 因此将每一个字符的编码值定义如下：

$$G^i(c) = h_i(c) \cdot Z_i \quad c \in UL_i$$

因此给定一组翻译函数 G={G^0，G^1,……,G^{L-1}}, 每个对应于字符串 s 的一个位置，于是可以定义字符串 s 的编码函数为：

$$Enc(s) = op(G^0(s[0]), G^1(s[1]), ...G^{|s|-1}(s[|s|-1]))$$

其中 op 函数是一个对单个字符编码后的数值进行操作的特定函数，且其通常是具有相关权重因子的线性加权求和（例如用于第 i 个位置的权重 wi），在该论文中仅考虑等权求和，即 wi 总是等于 1 的情形。

3.4 EMCE 编码方案

人们期望在对字符串进行编码时使其占用较少的编码空间，并希望可以根据字符串的前缀来区分各个字符串，同时降低 ULi 升序字符列表大小所带的影响，因此，该论文从压缩编码空间的角度来设计一种更紧凑的 MCE。该论文采用的压缩方案利用了两个方向上的信息，即在垂直方向上的唯一字符和水平方向上的后缀字符，除了各字符在对应升序字符列表 ULi 中的有序位置 i 之外，各字符的编码值还需要根据其后缀字符串编码的可能范围来确定，因此将这种方法称之为增强型多字符编码方案（EMCE）。

对于升序字符列表 ULi 中的每个字符 c，其有两种类型的相关变量，第一种类型是捕获 c 字符后面的字符范围，第二种类型是捕获以 c 字符开头的后缀字符串的范围，对应的记号如下：

- c_ch-/c_ch+：指在任一字符串中紧跟 c 字符的最小或最大字符（在 i+1 位置），并且称 c 为其紧跟字符的父字符；如果 c 字符是字符串的最后一个字符，那么 c_ch-为 0，注意 c_ch+ 不一定是 0，因为其他字符串可以在 c 字符后面。
- $\Omega_i(c)$/$O_i(c)$ 指后缀字符串 Enc(s[i..]) 的下限与上限，其中任意 s∈S, 有 s[i]=c。

EMCE 编码方案的关键思想是对后缀字符串实施正确的下限与上限约束。首先从水平方向上看，假设翻译函数 Gi 已知，则可以建立位置 i 与位置 i+1 之间后缀字符串之间的关系，如下所示：

$$\Omega_i(c) = G^i(c) + \Omega_{i+1}(c\_ch-) \quad for\, c \in UL_i, \quad 0 \le i < L$$

$$O_i(c) = G^i(c) + O_{i+1}(c\_ch+) \quad for\, c \in UL_i, \quad 0 \le i < L$$

上面的公式通过将 G_i(c) 加到以位置 i+1 开始的 c_ch-/c_ch+ 后缀字符串的 G^i(c) 上，得到以位置 i 开始的后缀字符串的下限 Ω_i(c) 或者上限 O_i(c)，这是从位置 i+1 到位置 i 的自然推论。这里需要注意对于 0≤i≤L，允许 i 扩展到 L，因为字节 0 是指字符串的末尾，且规定 Ω_i(0)=0 和 O_i(0)=0。

在垂直方向上，希望确保从位置 i 开始的 ULi[j] 的后缀字符串的下限大于 ULi[j-1] 后缀字符串的上限，这从根本上反映了字典排序的要求，具体形式如下：

$$\Omega_i(ULi[j]) > O_i(ULi[j-1]), \quad for\, 0 < j < |\, UL_i\,|$$

为了满足要求，将 $\Omega_i(ULi[j])$ 定义如下：

$$\Omega_i(ULi[j]) = \begin{cases} \theta_i & j = 0 \\ O_i(ULi[j-1]) + \delta_i & otherwise \end{cases}$$

其中 δ_i 是一个与位置相关的正数，用于指定极限之间的间隙。一般情况下将其设置为 1 就足够了，这里将其设置为一个变量，这样可以根据具体的优化目标进行调整，这同样适用于下限初始化参数 θ_i，

在一般情况下，将其设置为 0 就足够了。

当忽略处理可变长度字符串的要求时，在上述公式中，将 Ω_i(ULi[0]) 设置为 0（假设 θ_i 设置为 0)，以确保任何后缀字符串的编码都是非负的。当字符串长度不一致时，如果在这个位置出现了一个空终止符（即在这个位置有一个字符串终止符），这样就可以保证 Ω_i(ULi[0]) 大于零，这可以解释为，以非零字符开头的后缀字符串的下限应该大于以零字符开头的后缀字符串（它本质上是由空结束符本身组成）。后缀字符串之间的限制关系如下图所示，其中实心箭头表示字符之间在垂直方向上的依赖关系，虚线箭头表示字符之间下限与上限之间的约束关系，总的来说，它是一个迭代过程，从最后一个位置开始，并利用下限或上限之间的约束不断向后迭代计算。

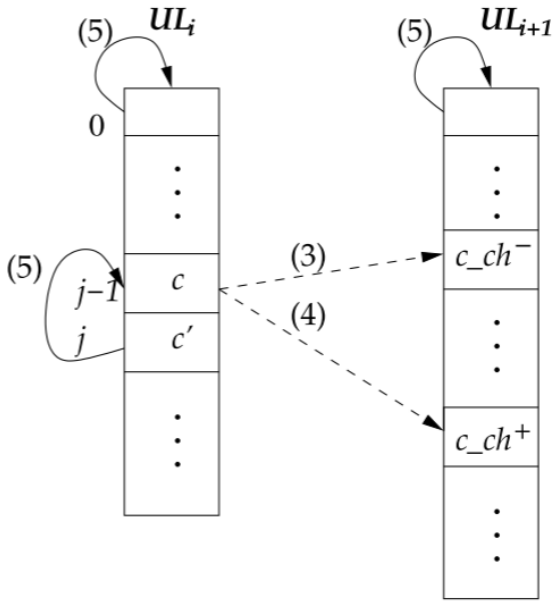

图 1: 后缀字符串限制关系图

4 复现细节

4.1 与已有开源代码对比

该论文的原作者未对相关源码及数据集进行公开，在本次复现过程中所有源代码均由本人编写。复现的主要内容包括：复现基数排序算法、复现 BMCE 和 EMCE 两种编码方案、实现在基数排序算法基础上采用多字符编码 MCE 算法（BMCE 与 EMCE 编码方案），降低排序基数，在原基础上提升字符串排序性能。

4.2 基数排序算法

该部分基数排序算法的伪代码如算法 1 所示，它是一种基于 MSD 的错位基数排序，每次查看一个字符。在伪代码中使用的 CharAt（s , d）操作是指从字符串 s 中提取第 d 个字符，通过调用 RADIXSORT（S , N , d）对字符串数组 S 完成排序，其基数大小为 256。

第 4 行至第 6 行指计算深度 d 处每个字符出现的频率，第 7 行生成 bucket 桶索引，用于标识每个桶在排序数组中的起始偏移量，第 8 行至第 10 行将字符串分发到辅助数组 aux 中，以便对第 d 个字符进行排序，然后复制回 S，第 11 行至第 13 行：如果分发后的桶中存在多个元素，则根据下一个字符递归地对桶的“子桶”进行排序。

Procedure 1 Pseudo-code for out-of-place Radix Sort.→ RadixSort(S, N, d)

Input: A string array S, size N, d (the d-th character to sort on)
Output: S (sorted)
Initialize per-bucket counter C = {0, . . . , 0}
for $s \in S$ **do**
    $C[CharAt(s, d)] + +;$
$Compute\ prefix\ sum\ on\ C\ to\ get\ per-bucket\ starting\ indices\ and\ store\ those\ in\ array\ bIndex$
end
for *i = 0 To N − 1* **do**
    $aux[bIndex[CharAt(S[i], d)] + +] = S[i] \rightarrow distribute\ to\ tem-porary\ array\ aux$
$Copy\ array\ aux\ back\ to\ S$
end
for *r = 0 To R − 1* **do**
    if *C[r] > 0* **then**
        $RadixSort(S\_r, C[r], d + 1) \rightarrow S\_r\ is\ the\ sub-array\ of\ S\ for\ bucket\ r$
    end
end

4.3 BMCE 编码与 EMCE 编码

（1）BMCE 编码算法：针对待排序的字符串，首先将各字符串位置 i 处的唯一字符（不重复的字符）存储在对应的 ULi（升序排列的字符列表）中，其中 ULi 的数量取决于待排序字符串的最大长度，然后再根据公式计算各个 ULi 升序字符列表的单位值，并规定最后一个 ULi 的单位值设置为 1，根据各个字符在 ULi 列表中的位置（从 0 开始）和 ULi 的单位值即可计算出单个字符的编码值，然后将待排序字符串的各单个字符的编码值进行累加计算出整个字符串的编码值，以上过程用到的所有计算公式如第 3.3 节中所示。

当所有待排序字符串长度均相同时，这种编码方式可以有效实现字符串的排序。但是 String 类型是一种可变长度的数据类型，如果仍然采用相同的计算方法，一般情况下会违反保序性，会存在如字符串“abg”与字符串“abgk”计算得到相同的编码值，而实际上字符串“abgk”的编码值至少要比字符串“abg”大 1，针对这种情况，一种快速的解决方法是假设空终止符（字节为 0）存在于每一个位置，因此对于任何非零字符 c 来说，其 hi（c）的值总是大于 0。这样一来，就可以保证两个字符串（其中一个字符串是另一个字符串的前缀）始终具有不同的编码值（长的字符串至少比短的字符串大 1），因此，对于一个特定的字符串，只需要跟踪是否有任何字符串终止在一个位置并且只增加该位置的基数，并将空终止符作为一个额外的字符添加到 ULi 中。

（2）EMCE 编码：当忽略处理可变长度字符串的要求时，借助第 3.4 小节中 EMCE 编码的相关公式可计算出各字符串的编码值，首先要将 Ωi(ULi[0]) 设置为 0（假设 θi 设置为 0），以确保任何后缀字符串的编码都是非负的。当字符串长度不一致时，如果在这个位置出现了一个空终止符（即在这个位置有一个字符串终止符），这样就可以保证 Ωi(ULi[0]) 大于零，这可以解释为，以非零字符开头的后缀字符串的下限应该大于以零字符开头的后缀字符串（它本质上是由空结束符本身组成）。对于这种情况，采用与 BMCE 编码相同的措施，将 NULL 结束符添加到 ULi 升序字符列表中，使得第一个非 NULL 字符总是有一个较大的下限。

EMCE 编码方案的伪代码如算法 2 所示，它由两个循环组成，第一个循环处理字符串集的每个位

置，第二个循环处理该位置上的每个唯一字符。在每个位置迭代开始时（第 4 行），将下一个位置的 NULL 结束符的下限与上限都初始化为 0，以建立迭代过程的基础，第 5 至第 6 行是指根据初始下限 θ_i 计算列表中第一个字符 c 的编码值，因为知道这个位置的 $\Omega_{i+1}(c\_ch-)$ 的值（对于最后一个位置，它总是为 0），在此基础上，有了字符 c 的转换值，就可以利用前文中的公式计算它的上限值，并进一步计算其后继字符的下限。总的来说，它是一个迭代过程，从最后一个位置开始，并利用下限或上限之间的约束不断向后迭代计算。

Procedure 2 EMCE

Input: Unique lists ULi , Lower limit initialization constants θi ,Limits gap constants δi , for $0 \le i < L$

Output: Translation function Gi , for $0 \le i < L$

for *each position i in [L − 1 .. 0]* **do**

    $Initialize\ \Omega_{i+1}(0)\ and\ O_{i+1}(0)\ to\ 0$

    $Get\ the\ first\ character\ c\ from\ ULi$

    $Set\ G^i(c)\ to\ \theta_i – \Omega_{i+1}(c\_ch-)$

    while *Move To Next (ULi , c′)* **do**

        $Set\ \Omega_(c')\ to\ G^i(c) + O_{i+1}(c\_ch+) + \delta_i$

        $Set\ G^i(c')\ to\ \Omega_i(c') - \Omega_{i+1}(c'\_ch-)$

        $Update\ predecessor\ c\ with\ c'$

    end

end

4.4 将 MCE 应用于基数排序

为了将 MCE 编码方案（BMCE 与 EMCE）应用于基数排序[25]，可以先将每个字符串分成在位置上连续且以字符为单位的组，每个组被编码为一个由编码值表示的 SC，其基数应当限制在一个预定义范围内。当使用 BMCE 编码方案时，基数大小的计算公式为 $\prod_{i=u}^{v} | UL_i |$ 其中 [u,v] 是当前组中字符的位置范围。对于 EMCE 编码方案，基数大小由以下方式确定：对于字符串数组 S 和任意的 s 属于 S[u,v]，存在 Enc（s）<$O_0(UL_0[|\ UL_0\ | -1]) + 1$ 这表明此时可以使用位置 u（加 1）的最大字符的上限作为基数大小，注意使用此基数大小时，EMCE 仅应用于索引范围 [u,v] 中的字符。

通常，对于 BMCE 和 EMCE 编码方式都可以通过每次延长一个位置来逐步形成一个组。算法 3 $Create\ Next\ Group$ 描述了 EMCE 编码方式形成组的伪代码，在开始排序之前，可以为 S 生成所有组，但这不太有效，因为要排序的字符串只有在新一轮排序开始时才在当前“桶”中，所以应该根据要排序的字符串进行编码，这可以通过在算法 1 中 Radix Sort 的开头插入算法 3 $Create\ Next\ Group$ 进而实现，相应地将 CharAt 函数更改为使用 SC 的编码值。

Procedure 3 CreateNextGroup

Input: A string array S, start position u, radix limit R′, batch size b

Output: radix size R, group ending position v, translation function Gi , for u ≤ i ≤ v

radix = 1; t1 = u;

while *true* **do**

  $t2 = t1 + b - 1$

  $Collect\ ULj\ and\ the\ corresponding\ c\_ch-\ and\ c\_ch+\ from\ S\ for\ j\ \in [t1...t2]$

  for *i = t1 to t2* **do**

    Invoke Procedure. 2 on S[u..i] and update radix

    if *radix > R′* **then**

      $v = i-1\ and\ break\ the\ While\ loop$

    end

    t1 = t2 + 1

  end

  Invoke Procedure.2 on S[u..v] for G^i and assign radix to R

end

5 实验结果分析

5.1 BMCE 编码方案

（1）针对所有待排序字符串长度等长的情形，假设存在一个字符串组 S 为“abf”,“mrn”,“aps”,“cuc”,“tdf”，“abg”，表示为 s0，…,s5，其中 L=3，对应升序字符列表 ULi 如下图所示：

Position	Unique List (UL_i)	Radix
0	{a, c, m, t}	4
1	{b, d, p, r, u}	5
2	{c, f, g, n, s}	5

图 2: ULi 升序字符列表示意图（BMCE）

接着计算得到每个 ULi 的 Zi 值分别为：Z0=25，Z1=5，Z2=1，然后再根据公式计算得到每个字符的编码值如下图所示：

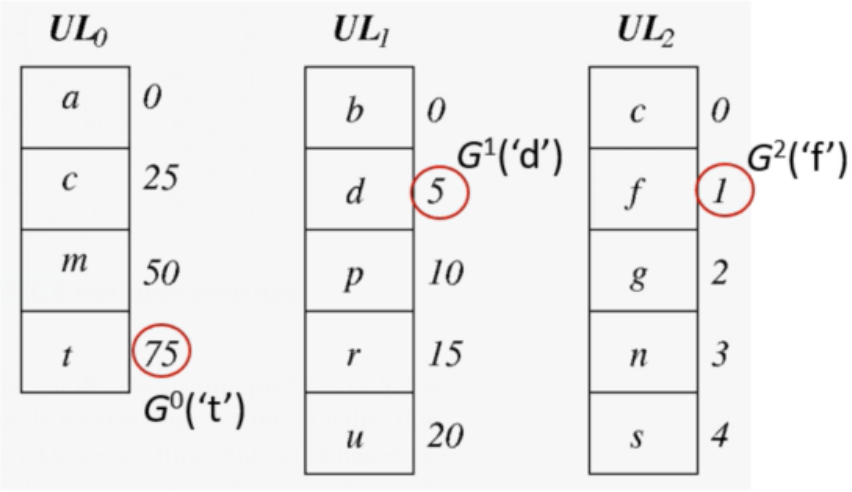

图 3: 各字符编码值示意图（BMCE）

例如字符串“tdf”的编码值为 Enc（“tdf”）=75+5+1=81；字符串组 S 各字符串 si 的编码值

分别为：1，68，14，45，81，2，然后对其编码值进行排序，最终得到对应排好序后的字符串序列为：“abf”，“abg”，“aps”，“cuc”，“mrn”，“tdf”。基数大小 radix 为 100，即一次性编码 3 个字符其编码区间为 [0,99]。

（2）针对不等长的字符串，假设存在以下字符串组 S 为：“mrn”,“aps”,“tdf”,“abg”,“abf”,“cuc”,“abgk”，首先收集字符串中每个位置上的 unique 唯一字符并存储在对应 ULi 升序字符列表中（在每个字符串的终止位置添加一个空终止符），最终得到的 ULi 如下图所示：

m	r	n	
a	p	s	
t	d	f	
a	b	g	
a	b	f	
c	u	c	
a	b	g	k

i	Unique List(UL_i)	Radix	Unit Value (Z_i)
0	{a, c, m, t}	4	50 (5*5*2)
1	{b, d, p, r, u}	5	10 (5*2)
2	{c, f, g, n, s}	5	2
3	{Ø, k}	2	1

图 4: ULi 升序字符列表示意图

可以发现，通过在字符串结束位置添加空终止符后使其编码基数 radix 增加了，从原先的 100 增加到 radix 为 4*5*5*2=200，各个 ULi 的 Zi 值也做出了相应的调整，并且各个字符串的编码值范围增大为 [0,199]。

5.2 EMCE 编码方案

假设存在一个字符串组 S 为“abf”，“mrn”，“aps”，“cuc”，“tdf”，“abg”，表示为 s0，…,s5，其中 L=3，首先分析各字符在竖直方向上的约束问题，以 b、d 字符为例，其中 Ω0(a)=Ω1(b)=Ω2(c)=0，Ω1(d)=O1(b)+ δi。对应升序字符列表 ULi 及竖直方向上字符之间的约束关系如下图所示：

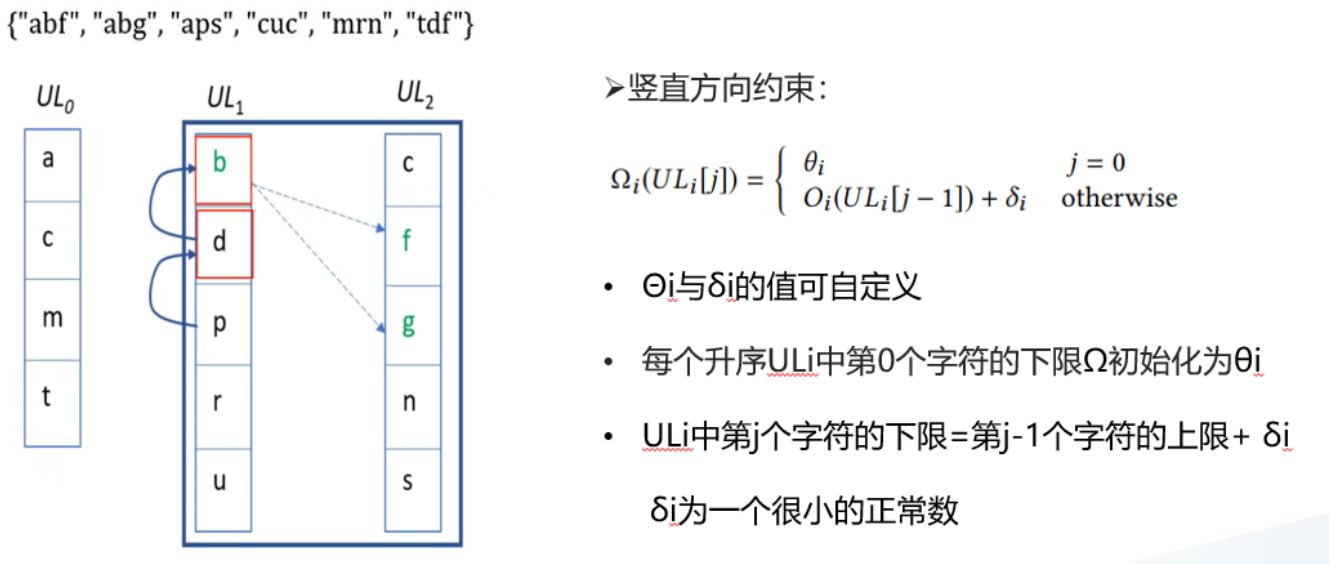

图 5: 竖直方向约束关系图

在水平方向上的约束关系如下图所示，以 UL1 升序字符列表中的字符 b 为例，b 字符的后缀 b_ch-为 f, 且其后缀 b_ch+ 为 g，因此有如下关系式：

Min(Enc(“b*”))=G1(‘b’)+Min(Enc(“f*”));

Max(Enc(“b*”))=G1(‘b’)+Max(Enc(“g*”));

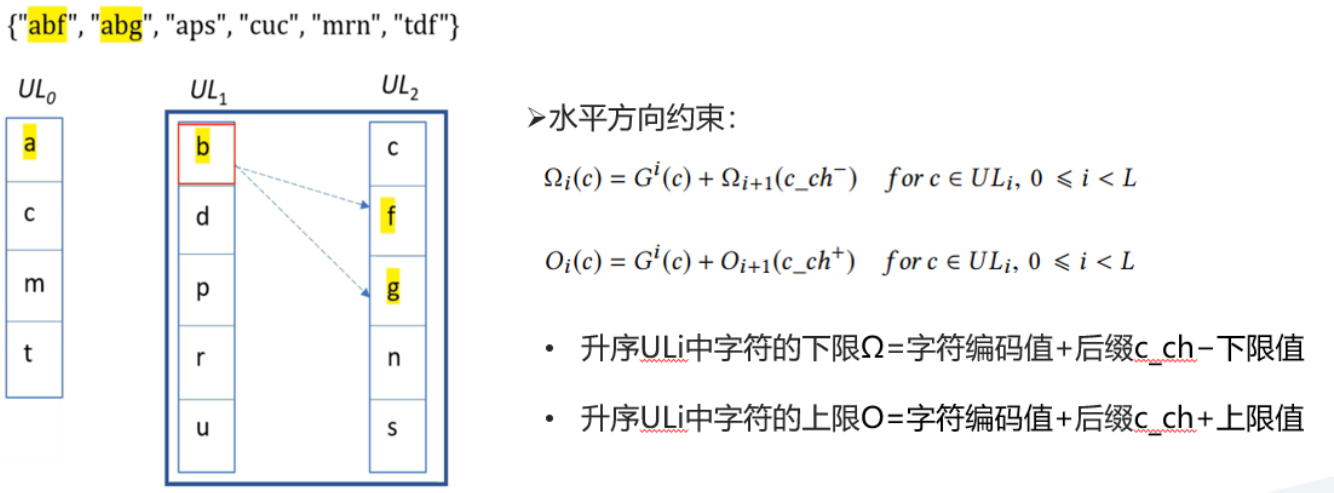

图 6: 水平方向约束关系图

通过分析 ULi 升序字符列表中各个字符在竖直和水平方向上的约束关系后，可利用前文描述的 EMCE 编码方式对应的公式计算出各个字符的上限值与下限值，进而计算出各个字符的编码值，其中以字符 b 和字符 d 为例计算其对应编码值，如下图所示：

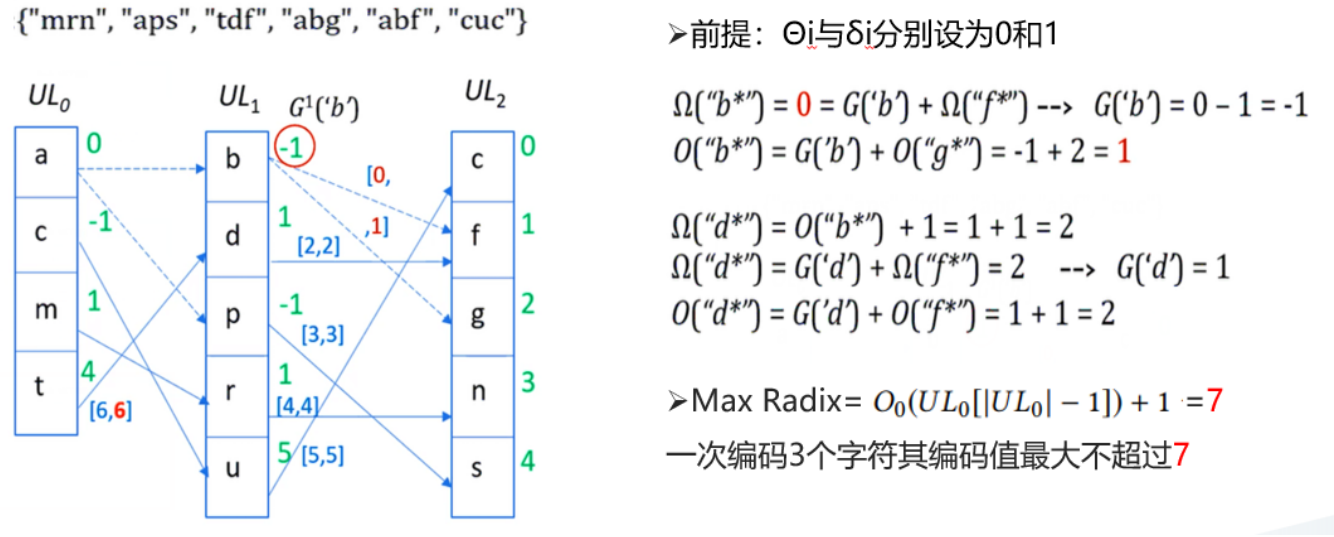

图 7: EMCE 编码示意图

同理将各字符串中的单个字符的编码值进行累加即可得到对应字符串的编码值，以字符串“aps”为例，其编码值为 Enc（aps）=0+（-1）+4=3。在该例子中采用 EMCE 编码方式一次性编码 3 个字符其编码值最大不超过 7，在计算出各字符串的编码值之后，即可使用基数排序算法对各字符串进行排序。

通过对比 BMCE 与 EMCE 两种不同的编码方案可以发现：使用 BMCE 编码方案可以为编码值生成大整数，即使是对于少量的短字符串，而且它对每个位置的升序字符列表的大小也非常敏感，最终对各字符串编码的数值也限定在一个较大的区间范围内，除非有意设计，否则字符串几乎不会覆盖 BMCE 计算范围内的每个编码值。EMCE 编码采用的压缩方案利用了两个方向上的信息，即在垂直方向上的唯一字符和水平方向上的后缀字符，除了各字符在对应升序字符列表 ULi 中的有序位置 i 之外，各字符的编码值还需要根据其后缀字符串编码的可能范围来确定，最终可以使得各字符串的编码值限

定在一个较小的区间内部，相比于 BMCE 编码方案，EMCE 编码在一定程度上降低了排序基数，并压缩了字符串的编码区间。

6 总结与展望

本文主要针对的工作是关于字符串的排序问题，在现有的字符串排序方法中，基数排序是一种既简单又有效的方法，有很多研究人员通过开展许多相关工作来加速字符串的基数排序，其中一种典型的方法是在每个排序过程中处理多个字符，然而，这种方法会导致基数过大的关键问题。为了解决这一问题，在该论文中引入了一种新的基于多字符编码的方法（BMCE 编码与 EMCE 编码），该方法可以显著降低基数，这种新的编码方法利用了稀疏字母表在空间上的稀疏性，最终可以使得各字符串的编码值限定在一个数值区间内部。

BMCE 编码原理是将每个字符串编码成一个大整数，这种方式对每个位置的升序字符列表的大小非常敏感，最终对各字符串编码的数值也限定在一个较大的区间范围内。而 EMCE 编码方式可以说是 BMCE 编码的改进，EMCE 编码采用了压缩方案，利用水平和垂直两个方向上的信息，除了各字符在对应升序字符列表 ULi 中的有序位置 i 之外，各字符的编码值还需要根据其后缀字符串编码的可能范围来确定，最终可以使得各字符串的编码值限定在一个较小的区间内部，相比于 BMCE 编码而言，其进一步缩小了编码区间并降低了基数。根据最终的测试结果可以发现，BMCE 编码方式与 EMCE 编码方式都能很好地运用于字符串的排序问题上，能够明显降低排序基数，具有较好的可扩展性。

本文以上两种编码方式所采用的编码函数都是仅考虑了等权求和的方式，即 wi 总是等于 1 的情形（主要是针对 op 函数，op 函数是一个对单个字符编码后的数值进行操作的特定函数，且其通常是具有相关权重因子的线性加权求和，例如用于第 i 个位置的权重 wi），今后的研究工作可以在探索非均匀权重编码函数方面进一步展开研究，考虑非等权求和的方式，分析其与均匀权重编码方式之间的差异性，以及是否具有更有效的排序性能等。

参考文献

[1] MUSSER D R. Introspective Sorting and Selection Algorithms[J]. Software Practice and Experience, 1997, 27(8).

[2] MCILROY P M, BOSTIC K, MCILROY D. Engineering Radix Sort[J]. Computing systems, 2002, 6(1): 5-27.

[3] PAIGE R, TARJAN R E. Three Partition Refinement Algorithms[J]. SIAM Journal on Computing, 1987, 16(6): 973-989.

[4] JIMENEZ-GONZALEZ D, NAVARRO J, LARRIBA-PEY J L. CC-Radix: a cache conscious sorting based on Radix sort[C]//Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings. 2003: 101-108.

[5] NG W, KAKEHI K. Cache Efficient Radix Sort for String Sorting[J]. IEICE Trans. Fundam. Electron. Commun. Comput. Sci., 2007, E90-A(2): 457-466.

[6] KÄRKKÄINEN J, RANTALA T. Engineering Radix Sort for Strings[C]//AMIR A, TURPIN A, MOFFAT A. String Processing and Information Retrieval. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009: 3-14.

[7] BINGMANN T, EBERLE A, SANDERS P. Engineering Parallel String Sorting[J]. Algorithmica, 2014, 77: 1-52.

[8] WASSENBERG J, SANDERS P. Engineering a Multi-core Radix Sort[C]//JEANNOT E, NAMYST R, ROMAN J. Euro-Par 2011 Parallel Processing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011: 160-169.

[9] NG W, KAKEHI K. Merging String Sequences by Longest Common Prefixes[J]. Information and Media Technologies, 2008.

[10] BENTLEY J L, SEDGEWICK R. Fast algorithms for sorting and searching strings[C]//Eighth Acm-siam Symposium on Discrete Algorithms. 1997.

[11] AXTMANN M, WITT S, FERIZOVIC D, et al. In-place Parallel Super Scalar Samplesort (IPSSSSo) [C]//. 2017.

[12] SANDERS P, WINKEL S. Super Scalar Sample Sort[C]//ALBERS S, RADZIK T. Algorithms – ESA 2004. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004: 784-796.

[13] BINGMANN T. Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools[Z]. 2018.

[14] BINGMANN T, SANDERS P. Parallel String Sample Sort[C]//BODLAENDER H L, ITALIANO G F. Algorithms – ESA 2013. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013: 169-180.

[15] KRISTO A, VAIDYA K, ETINTEMEL U, et al. The Case for a Learned Sorting Algorithm[C]// SIGMOD/PODS '20: International Conference on Management of Data. 2020.

[16] BINGMANN T, EBERLE A, SANDERS P. Engineering Parallel String Sorting[J]. Algorithmica, 2014, 77: 1-52.

[17] CHO M, BRAND D, BORDAWEKAR R, et al. PARADIS: An Efficient Parallel Algorithm for in-Place Radix Sort[J]. Proc. VLDB Endow., 2015, 8(12): 1518-1529.

[18] OBEYA O, KAHSSAY E, FAN E, et al. Theoretically-Efficient and Practical Parallel In-Place Radix Sorting[C]//SPAA '19: The 31st ACM Symposium on Parallelism in Algorithms and Architectures. Phoenix, AZ, USA: Association for Computing Machinery, 2019: 213-224.

[19] BLELLOCH G E, FINEMAN J T, GIBBONS P B, et al. Sorting with Asymmetric Read and Write Costs [Z]. 2016.

[20] FRANCESCHINI G, MUTHUKRISHNAN S, PĂTRAŞCU M. Radix Sorting with No Extra Space[C]//ARGE L, HOFFMANN M, WELZL E. Algorithms – ESA 2007. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007: 194-205.

[21] GU Y, SUN Y, BLELLOCH G E. Algorithmic Building Blocks for Asymmetric Memories[C]//AZAR Y, BAST H, HERMAN G. Leibniz International Proceedings in Informatics (LIPIcs): 26th Annual European Symposium on Algorithms (ESA 2018): vol. 112. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018: 44:1-44:15.

[22] KRISTO A, VAIDYA K, ÇETINTEMEL U, et al. The Case for a Learned Sorting Algorithm[C]// SIGMOD '20: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland, OR, USA: Association for Computing Machinery, 2020: 1001-1016.

[23] RAFIEV A, MOKHOV A, BURNS F P, et al. Mixed Radix Reed-Muller Expansions[J]. IEEE Transactions on Computers, 2012, 61(8): 1189-1202.

[24] ARNDT J. Mixed radix numbers[M]//Matters Computational: Ideas, Algorithms, Source Code. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011: 217-231.

[25] RAHMAN N, RAMAN R. Adapting Radix Sort to the Memory Hierarchy[J]. ACM J. Exp. Algorithmics, 2002, 6: 7-es.