一种 *k*-ary 搜索树的快速求交算法

王 珏[1]，包诗琦[2]，宋省身[3†]
(1. 武警警官学院 信息通信系，成都 610213；2. 四川龙桥黑熊救护中心，成都 610500；3. 国防科技大学 前沿交叉学科学院，长沙 410000)

摘 要：*k*-ary 搜索树因其对高速缓存和 SIMD 并行指令集天然的适配性，正在受到越来越多的关注和研究。近年来，它被成功地应用于搜索引擎倒排索引结构中，用于实现高效的查询处理和索引压缩。但基于 *k*-ary 搜索树的查询处理算法目前仍处在一种相对简单基础的应用程度，效率提升有限；而且查询算法仅限于元素搜索，大大限制了其适用范围。基于上述观察，研究了基于 *k*-ary 搜索树的求交算法，并提出了两种优化技术用于压缩搜索范围以提升查询效率。实验证明，结合不同的遍历方式，优化后的求交算法可以极大地提高查询速度，尤其是针对存储海量数据的长倒排链，配合更大的 SIMD 寄存器，*k*-ary 搜索树相比于传统求交算法的优势更为明显。

关键词：信息检索；数据结构；算法优化；求交算法
中图分类号：TP391.3 **文献标志码**：A **文章编号**：1001-3695(2021)09-031-2732-05
doi:10.19734/j.issn.1001-3695.2020.12.0548

Efficient intersection algorithm based on *k*-ary search tree

Wang Jue[1], Bao Shiqi[2], Song Xingshen[3†]
(1. *Dept. of Information & Communication, Officers College of PAP, Chengdu* 610213, *China*; 2. *Chengdu Bear Rescue Centre, Chengdu* 610500, *China*; 3. *Academy for Advanced Interdisciplinary Studies, National University of Defense Technology, Changsha* 410000, *China*)

Abstract: *K*-ary search tree is a promising structure due to its friendliness to cache awareness and SIMD acceleration. Recent years have witnessed its successful application to inverted index for efficient query processing and index compression. However, its query processing methods are either complicated or rudimentary, and its adoption is limited in membership test, which both impair its application in real life. This paper studied the intersection algorithm for *k*-ary search tree, and introduced two early termination techniques collaborating with particular traversal methods to improve its intersection efficiency. The experimental analyses validated the proposed optimizations could greatly improve the query speed, and the optimized *k*-ary search tree suits better for long inverted lists, and when equipped with larger SIMD registers, it is able to beat the state-of-the-art sorted-array based intersection algorithms.

Key words: information retrieval; data structure; algorithm optimization; intersection algorithm

0 引言

在信息检索领域，效率问题是一个相当基础但又至关重要的课题。数据体量的日益增长给搜索引擎带来了巨大挑战，而搜索引擎需要严谨设计的数据结构，以高效地回答大量用户查询。倒排索引是一种被广泛采用的数据结构，通常用于存储和查询处理[1,2]。倒排索引可以看做是一张映射表，将每个唯一的词项映射到对应的倒排链表上，倒排链表存储了与该词项相关的所有文档标志符和其他相关信息，如文档频率和位置信息等。为提升搜索引擎的效率，针对倒排索引的压缩和查询处理技术成为了学术界的研究热点。众多的优化算法在索引的空间占用和查询效率之间取得了很好的折中，近年来的研究开始利用硬件创新提升数据结构的算法效率，如基于 GPU 和 SIMD 指令集的并行算法[3,4]。

搜索引擎需要处理的查询包括布尔查询和排序查询，其中，交集合取查询是信息检索中常见的查询处理操作之一，广泛应用于文档检索或文本检索[5]。求交查询往往作为排序查询的前提条件出现，过滤出绝大多数不太相关的候选文档。

给定一个查询 q，求交查询是指从 $|q|$ 个词项对应的倒排链表中提取公共元素，即包含所有查询词项的文档标志符。由于求交过程必须处理整个链表以提取出包含公共文档的子集，所以它占用了处理用户查询的大部分时长，而对提取出的子集通常小很多，后续的相似度排序耗时要少很多。依靠搜索算法针对一条倒排链上的元素反复遍历另一条链表显然效率低下。

近年来，研究人员开始引入树型数据结构来代替传统排序数组应用于倒排索引[6,7]。其中，*k*-ary 树由于其内在的缓存和 SIMD 友好特性，应用最为广泛[8,9]。*k*-ary 树可以看做一种 B + 树[10]，每个节点由 $k-1$ 个元素组成，这些元素将整个集合均匀地划分为 k 个子树，依次迭代。对于数据密集型的计算任务，性能瓶颈取决于数据在不同层级 cache 的交换速度，而通过将节点大小与 cache 大小相匹配，可以充分利用 cache 数据预读的特性并减少数据交换的频率，因此 *k*-ary 树相比于排序数组优势明显。

然而，相关的研究仅考虑了在树上进行元素搜索，并没有深入到搜索引擎的常用查询中，这就限制了它的适用性。为此，本文研究了基于 *k*-ary 树的求交算法，并介绍了两种压缩搜索域的优化方法，给出了三种不同的遍历方式进行求交运算。通过实验分析，本文算法在模拟和真实数据集上都有效地提升了计算效率，使 *k*-ary 树的适用性得到了加强。

收稿日期：2020-12-01；**修回日期**：2021-02-01

作者简介：王珏(1987-)，男，四川隆昌人，讲师，硕士，主要研究方向为信息系统与安全；包诗琦(1993-)，女，四川成都人，学士，主要研究方向为科学教育；宋省身(1990-)，男(通信作者)，河南濮阳人，副教授，博士，主要研究方向为信息检索与数据分析(songxingshen@nudt.edu)。

1　相关工作

1.1　倒排索引结构设计

为了提高效率,针对倒排索引的压缩和查询优化技术层出不穷。近年来,研究开始将倒排索引与其他数据结构相结合以获得进一步的改进。Vigna[11] 采用 Elias-Fano 索引构建混合索引,以获得更好的压缩效果;Lemire 等人[12] 研究发现位图压缩在特定场景下比倒排索引更有效;基于机器学习构建的 learned index[13,14] 也因占用空间少、查询处理速度快而受到重视。如今,硬件的发展,特别是 SIMD 并行指令集和层次化存储体系结构的发展,也催生了大量的适配应用[4,5]。领域内普遍认可的观点是针对数据密集型任务,分支预测失误和缓存未命中比冗余算术操作时钟周期的代价更高。因为分支预测失误将刷新指令管道并回滚,缓存未命中将导致数据从低速缓存中重新加载。

树型结构与这些理念不谋而合,其数据的分层组织与多层存储自然契合,通过将节点大小与各级缓存相匹配,可以最大限度地减少数据交换,获得较高的数据吞吐量。此外,通过在节点内安排元素排列,SIMD 指令可以用来并行处理它们。树型结构是数据库常用的底层数据结构。Kim 等人[15] 提出了 FAST,这是一种对缓存结构敏感的索引树布局,其节点根据底层硬件的内存交换页、缓存行和 SIMD 寄存器等架构特征重新排列。FAST 应用于树遍历和搜索操作时,可以极大地消除内存延迟和提升并行效率。Schlegel 等人[10] 首先研究了通过 SIMD 指令进行 k-ary 搜索。通过同时比较 $k-1$ 个元素,它们的算法可以将搜索范围划分为 k 个子区域,从而将时间复杂度降低到 $\log_k N$,并将树结构按级别进行线性化存储,以避免非连续内存访问。Wang 等人[16] 提出了一种基于 k-ary 搜索树的倒排索引压缩算法 MILC,该索引基于动态划分将倒排链分成不同长度的子块,并通过 k-ary 搜索树将倒排项组织起来,实现缓存感知和 SIMD 加速的能力,具有比现行的压缩算法更快的查询速度和更高的压缩率。

1.2　k-ary 搜索树

k-ary 搜索树的每个节点都有 $k-1$ 个元素和 k 个子节点。当一棵树的所有节点都有 $k-1$ 个元素,每个内部节点都有 k 个后续节点,并且每个叶子节点都处于相同的深度,那么它就是一棵完美树(perfect tree),否则叫做完整树(complete tree)。对于后者,唯一的区别在于叶子层,节点在右侧部分消失。最后一层的剩余叶子从左到右排列,一旦被全部移除,上层的节点就会生成一棵高度更小的完美树。树结构如图 1 所示。

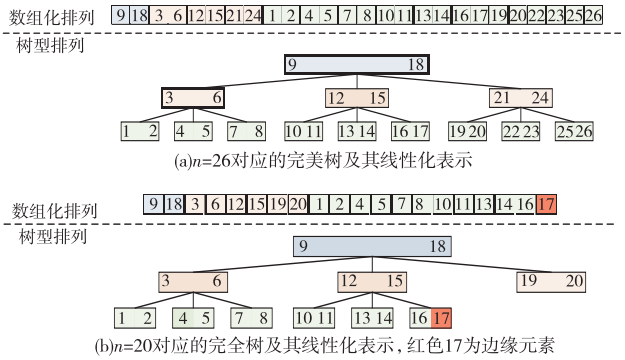


图 1　$k=3$ 对应的完美树和完全树结构

Fig. 1　Perfect and complete tree structure for $k=3$

对于从大小为 n 的排序数组派生的完美树,n 满足 $n=k^H-1$:

$$\underbrace{k-1}_{\text{第1层}}+\underbrace{(k-1)\times k}_{\text{第2层}}+\underbrace{(k-1)\times k^2}_{\text{第3层}}+\cdots+\underbrace{(k-1)\times k^{H-1}}_{\text{第}H\text{层}}=k^H-1$$

因此树的高度是 H,在深度 d(从 0 开始计算)有 k^d 个节点和 $k^d(k-1)$ 个元素。为了将一个排序数组转换成一个完美的 k-ary 树,每次算法从当前范围生成一个节点,节点内的元素是从 $k-1$ 分隔符中提取的,该分隔符将当前范围划分为 k 个大小相等的集合。划分过程从整个数组开始构建根节点,然后对应它的子节点。同时,为了节省空间开销,树也被线性化为一个数组,按层级顺序堆叠,而不显式地存储任何树指针。

Schlegel 等人[10] 介绍了一种按顺序将排序数组转换为搜索树的算法。排序数组索引 i 通过计算其相关深度 $d_H(i)=\sum_{x=1}^{H-1}\operatorname{sgn}(i \bmod k^{H-x})$ 和偏移量 $o_H(i)=\lfloor\frac{k-1}{k}\times\frac{i}{k^{H-d_H(i)-1}}\rfloor$,映射到线性化树索引 $f_H(i)$,即

$$f_H(i)=k^{d_H}(i)+o_H(i) \tag{1}$$

但是,输入数组的长度可能不足以形成一个完美的树,因此需要引入额外的操作来构造一个完整树。注意到完整树右下角的元素,称之为边缘元素(fringe entry),它处在线性化数组的结尾 $n-1$ 处,所有小于该元素的值在树中的位置与在同深度完美树中的位置一致,而大于其值的位置则是与深度为 $H-1$ 的完美树一致。通过计算边缘元素 $f_H^*(n)$ 在排序数组的索引,其余元素在完整树中的位置 $g_n(i)$ 由下式给出:

$$g_n(i)=\begin{cases}f_H(i) & i\leqslant f_H^*(n)\\ f_{H-1}(i-o_H^*(n)-1) & \text{其他}\end{cases}$$

SIMD 指令能够同时操作寄存器中的多个数据,特别适合于加速数值计算或计算密集型任务。元素搜索是一项计算简单但数据密集的任务,因为它只需要加载数据并将其与搜索关键字进行比较,然后根据结果重复此过程,直到找到匹配项或达到终点。近年来基于 SIMD 的求交运算研究很多,如 Inoue 等人[17] 利用 SIMD 指令过滤掉代价高昂的分支预测失误,通过每次只加载比较数值的部分比特,能够同时比较更多的元素;Song 等人[18] 探索了 SIMD 指令在多倒排链求交中的应用,并提出了一种灵活的搜索算法来平衡非 SIMD 和 SIMD 指令数量以提高效率。

虽然这些基于 SIMD 的算法实现了相当大的加速,但它们主要用在排序数组上[19,20]。而结合到 k-ary 树时,整个过程变得更加简单和高效,节点大小与寄存器对齐,节点内元素将搜索范围分为 k 个等大小的子域,一个节点内的比较操作只需一条指令即可完成,通过获取的节点内偏移量 $m(m\in[0,k]$ 是不大于目标值的最大值位置),可以直接寻址到对应的子节点上。当前基于树型结构的应用集中在数据库和图计算中,局限于值查找或者简单的布尔查询,而应用于倒排索引的求交查询研究较少,通常是简单地将参与运算的两棵树顺序搜索,没能利用元素的分布特性,这显然很低效。

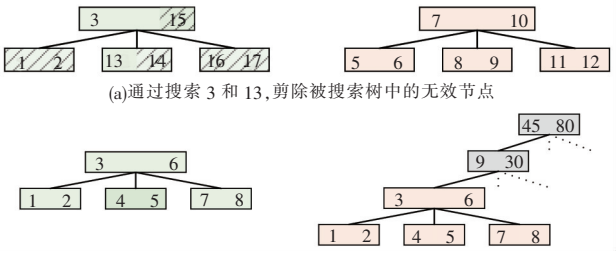
2　基于 k-ary 搜索树的求交算法

在给定两棵 k-ary 树的情况下,计算交集的一个隐式假设是将较小的线性化树视为未排序数组,忽略其树结构,然后使用基于 SIMD 的搜索算法在另一棵树上按顺序搜索其元素。在下文中将此方法称为 sequential。该方法没有充分利用 k-ary 搜索树结构的数值分布,因为它忽略了元素之间的内在关系,并且每次搜索都从根节点开始。

通常,潜在的交集元素会遍布整个树,一种更明智的方法是通过剪枝跳过无效的子树而不是穷举遍历。这里无效子树的概念包含了两个含义:首先,从较小的树(被搜索树)中提供一个元素 ε,当搜索过程到达另一个树的最左边或最右边的叶节点时,在较大的树(搜索树)中已经没有元素小于或大于 ε,那么元素 ε 的左子树或右子树即可以被剪除,因为搜索树中已

经不存在匹配,这里无效子树位于被搜索树中。其次,希望通过上一次搜索来缩小当前的搜索范围,即在搜索树上找到包含潜在匹配的最小子树,而不是每次都从根节点搜索整棵树,此时无效子树位于搜索树上。因此,通过合理设计最大化剪除这些无效子树,对于提升效率至关重要。

在图 2 中说明了上述两种剪枝情况,其中,左侧为被搜索树,右侧为搜索树,每次从左侧被搜索树提取一个元素在右侧搜索树中进行查找。在图 2(a)中,给定一个元素 3,搜索指针就会到达最左边的叶节点,但仍没有找到匹配项,然后可以立即删除元素 3 包含 1 和 2 的左子树。同样,当搜索元素 13 时,指针将到达最右边的叶节点,然后元素 13 的右侧同级元素和兄弟节点(分别为 14 和 16,17),甚至其父级元素 15 都可以安全地删除。在图 2(b)中,可以清楚地看到所有匹配项都位于左下角的子树中,其余节点并不相关,并且阻碍了搜索过程,将搜索区域限制在最小子树中就可以避免从根节点开始进行不必要的比较。



(a)通过搜索 3 和 13,剪除被搜索树中的无效节点

(b)搜索树中的匹配值均在左下角子树,其他灰色元素无须重复比较

图 2 k-ary 搜索树求交时的两种剪枝方法

Fig. 2 Two pruning strategies for the intersection of k-ary seach tree

接下来提出了两种剪枝算法来实现上述概念,并分别命名为 skip 和 narrow,而搜索过程仍可使用 SIMD 指令同时比较整个节点,并行效率保持不变。为了便于理解,算法以完美树求交作为前提,但并不影响在完整树上使用。

2.1 skip:针对被搜索树的剪枝

给定一个节点,使用它所处的深度和在当前层级的索引来标志它在树中的位置,例如一棵深度为 2 的完美树,其根节点表示为(0,0),三个子节点分别是(1,0)、(1,1)和(1,2)。给定任意点(d,r),它在线性化表示中的索引为 $o_{(d,r)}=k^d-1+r(k-1)$。

当从搜索树中搜索元素时,增加两个参数来标志搜索范围,称做门卫节点 l 和 r,它们的标志分别为(d_l,r_l)和(d_r,r_r)。当求交开始时,左右门卫分别是搜索树最左侧和最右侧的叶子节点,左门卫的坐标终会是$(d,0)$,而右门卫则取决于搜索树是否是完美树。当搜索完被搜索树的目标元素时,无论找到与否,搜索算法都会返回一个坐标值(d',r'),对应不小于目标元素的最大值。此时,搜索范围被切分为两块,对于目标元素左子树上的节点,其搜索范围由(d_l,r_l)和(d',r')构成;对于目标元素右侧兄弟元素和右子树上的节点,其搜索范围由(d',r')和(d_r,r_r)构成。很显然,当任一搜索范围变为空时,被搜索树上的子节点可以直接跳过,无须重复计算交集。

算法 1 展示了 skip 的原理。在第 11 行中,当新找到的位置和左右门卫分别重合时,后续的搜索会在第 4 行处终止,并影响到更深层的子节点。值得注意的是,在指针 p_2 从未更新过,每次搜索都要从搜索树的根节点启动;被搜索树获取元素的方式为顺序获取,即 sequential,这样处理的代价就是需要为每个节点维护门卫节点表,等指针 p_1 移动到对应节点时才能使用。在后续的讨论中,会介绍另一种遍历方式,无须维护节点表的同时提供一种更紧致的搜索域,从而进一步提升剪枝的效率。

算法 1 k-ary 搜索树求交算法

输入:指向两棵树 T_1、T_2 线性化数组根节点的指针 p_1 和 p_2。

输出:指向交集数组头部的指针 p。

```
1  初始化在 T2 上的门卫节点(dl,rl)和(dr,rr)
2  取出被搜索元素 ε = *pl
3  while ε 不为 EOF do
4    if 当前搜索域的左右门卫重合 then
5      return
6    end if
7    从 p2 指向的位置开始沿着 T2 搜索 ε,并返回一个不小于 ε
     的元素 ε°及其位置(d°,r°)
8    if ε° = ε then
9      *p++ = ε
10   end if
11   调整 T1 中 p1 指向元素左子树的右门卫节点、右子树及右侧
兄弟节点的左门卫节点为(d°,r°)
12   ε = *(++p1)
13 end while
```

2.2 narrow:针对搜索树的剪枝

基于两个门卫节点限定的范围,寻找到最小子树,而不是从根节点开始,则需要执行的比较次数就会少得多。考虑到数据流水线中分支预测失误导致的数据重载代价很高,最小子树优势就更加明显。而寻找最小子树是图论中的一个经典问题,即最小共祖先问题(lowest common ancestor,LCA):给定树结构中任意两个节点 v 和 w,LCA 是 v 和 w 同时作为其子节点的层级低的节点。根据左右门卫,使用 LCA 代替根节点,自然就找到了包含潜在交集的最小子树。

找到任意两个节点 LCA 的一种简单方法是沿着树反溯,直到找到一个公共节点,但速度太低,反而阻碍了算法效率。本文使用的方法是将 LCA 问题简化为范围最小查询问题(range minmum query,RMQ),即在给定数组中查找在任意个索引之间最小值的索引位置。具体的方法是:首先在搜索树上建立一个深度优先的 Euler 图,并将路线中的节点按照遍历顺序记录其在线性化数组中的索引值,形成数组 E。在 E 中,由于 Euler 回路的存在,每个节点会被恰好访问两次,其长度恰好为 $\lceil 2n/(k-1)\rceil$。除此以外,还有一个同样长度的数组 L 用于记录 E 中每个节点的深度,以及长度为 n 的数组 H,$H[i]$表示线性化数组中节点 i 在 E 中首次出现的位置。这样,给定任意节点 v 和 w,本文首先查找它们在 H 中出现的位置 $H[v]$和 $H[w]$,随后查找 L 中对应于 $L[H[v]]$和 $L[H[v]]$的最小值,也就是最小的深度值,最后返回 E 中相同位置的值即可,也就是 $LCA(v,w)=E[RMQ_L(H[v],H[w])]$,其中 RMQ_L 即为第二步查找最小深度值的操作。使用顺序查找的时间复杂度为 $O(n)$,为提升效率本文基于动态规划构建稀疏表,存储 L 中任意长度为 2^k 的区间 RMQ_L,稀疏表实际上是一个二维数组 $M[0,n-1][0,\log n]$,$M[i][j]$中是起点为 i 长度为 2^j 的区间 RMQ_L。给定任意区间$[i,j]$,分别从头尾找到该区间最大的 2 次幂长度,RMQ_L 即为这两个区间的最小值,令 $k=\lfloor \log(j-i-1)\rfloor$,则

$$RMQ_L(i,j)=\begin{cases} M[i][k] & M[i][k]\leqslant M[j-2^k+1][k] \\ M[j-2^k+1][k] & \text{otherwise} \end{cases}$$

这样一来,RMQ_L 可以在 $O(1)$时间内得出,而每次搜索的平均比较次数也从 H 下降为 $H/2$。注意这些数组可以在搜索树时同步建立,不会影响到查询效率,代价是这些辅助索引结构会造成额外的空间占用。为了保证查询速度,本文采用未压缩的原始结构,针对辅助数据结构的压缩可以参考文献[11,15]。

这两种剪枝算法同样适用于完全树,无论是 skip 中判断门卫节点是否重合,还是 narrow 中计算 LCA,对任意两个位置的节点,始终能够找到一条路径将两个节点连接起来。而 LCA

则是最短路径上,层级最高的节点;当路径长度为 0 时,即为节点重合。因此,剪枝算法对搜索树的叶子层并无额外要求。加入了 narrow 以后,算法 1 仅需要修改第 7 行的位置,即根据当前左右门卫节点,令 $p_2 = LCA((d_l, r_l), (d_r, r_r))$ 即可使搜索 T_2 时不再从根节点开始,达到提升效率的目的。

3　求交算法的遍历方式

为了最大限度地发挥两种剪枝算法所带来的增益,被搜索树上的每个元素搜索域应该尽量准确。而门卫节点的更新发生在上一个节点的搜索之后,因此从被搜索树上选择元素的顺序就变得尤为重要。本文将介绍三种遍历方式,如图 3 所示。

图 3　被搜索树求交时的三种遍历方式

Fig. 3　Three traversal techniques for the tree to be searched

第一种是 sequential,如前所述,它忽略了它的树结构,像对待数组一样逐层取值。之前的研究中都是使用这种方法,但使用剪枝优化时,就需要维护一个临时表来记录从上层元素继承的门卫节点位置。除此以外,sequential 返回的搜索域并不是最优的,因为它是逐节点遍历的,即节点内第一个元素完成搜索后,其子节点和兄弟节点搜索域都被更新为同一组门卫节点。兄弟节点的搜索不会更新其左子节点的搜索域,但如果先搜索子节点元素,它可以进一步压缩兄弟节点的搜索域。以图 3(a)为例,当搜索完元素 3 时,其左子节点被排除,sequential 接下来会搜索元素 15,但如果先搜索元素 13,就可以排除掉整个搜索域,进而提前结束求交。

为此,本文引入一种新的遍历,称为 hierarchical,给定一个节点的 $k-1$ 个元素和 k 个子节点,本文以交错的方式遍历其元素和子节点,如图 3(b)所示,每次先访问一个父节点元素,随后下沉访问左子节点,之后访问父节点右侧兄弟元素,再次访问该元素左子节点,依次往复。注意第 $k-1$ 个子节点已经没有父级元素,结束后直接跳转到第 k 个子节点。这种遍历方式可以最大限度地利用节点元素之间的依存关系排除无效子树。

第三种遍历树的方法称为 sorted,即以元素数值大小升序求交,其结果集也自然是排序的,而前两者必须依赖额外的操作来对结果集进行排序。同时,sequential 和 hierarchical 可以看做是一种双边搜索,因为它们保留了两侧门卫节点,而 sorted 实际上是一种单边搜索,因为它只能从左侧更新节点。对 sorted 而言,skip 不再可用,而 narrow 仍然适用,但实验中发现提升并不明显,因为造成延迟的主要原因在获取元素的方式上。

同样,三种遍历方式也需要对算法 1 进行相应的修改。其中 sequential 是算法 1 的默认遍历方式,而针对另外两种遍历方式,需要按照修改第 12 行中更新 p_1,也就是获取 T_1 中待搜索元素的顺序。其中,sorted 需要通过式(1)的逆函数获取数值顺序。而 hierarchical 则相对复杂,需要通过递归的方式逐层按照先左子树,后右侧兄弟节点和右子树的方式获取元素。

4　实验与分析

本章将设计实验测试上一章中针对 k-ary 搜索树的求交优化算法。首先,测试不同的遍历方式对效率的影响,随后结合两种剪枝算法观察效率的变化,最后测试算法针对不同节点大小的 k 效率变化。

本文使用的数据集是 TREC GOV2 数据集,由 2 500 万篇文档和 1 500 万个词项构成,未压缩状态约 426 GB。本文从中提取出词项及其对应的倒排链,并将倒排项中文档标志符按照 URL进行重排序。算法由 C ++ 进行编写,使用 G ++ 5.4.0 和-O3 优化编译。相关的代码已经开源在了 GitHub 上(https://github.com/Sparklexs/k-ary-search-tree)。

测试平台为一台 8 核 Intel i7-9700 CPU,3.30 GHz 主频和 32 GB RAM,操作系统为 CentOS 7。所有的测试结果取连续执行 20 次的平均值。

1)遍历方式　在测试剪枝技术之前,本文首先通过三种未优化的遍历方法来执行求交。这里固定 $k=3$,由于节点大小不同而导致的性能差异将在后续实验中比较。如图 4 所示,在所有情况下,sequential 都比其他两种方法表现出相当大的优势,减少了 25% ~75% 的时间。注意横坐标代表两棵求交树之间的长度比例,本文将被搜索树大小固定为 1 024 个元素,并不断倍增另一棵的大小观察差异。计算时间随着数据集的增长而延长,并呈现出近似线性关系。当长度比例增大时,这三种方法减缓的趋势相同,sequential 和其他两种方法之间的时间差几乎没有变化,这是因为这三种遍历方法在被搜索树中没有消除任何子树,计算效率主要取决于被搜搜树中的元素个数。给定一个被搜索元素,搜索过程总是从搜索树的根开始,其平均时间复杂度为 $O(\log_k n)$,被搜索树的长度固定为 m,则交集的时间复杂度为 $O(m \times \log_k n)$。这些方法之间的差距主要是由于它们获取元素的效率不同,sequential 触发的缓存未命中最少,而另外两种方法的时间成本则相对增加。在这种情况下,交集的计算时间没有使用任何优化,唯一影响效率的是求交树的大小。

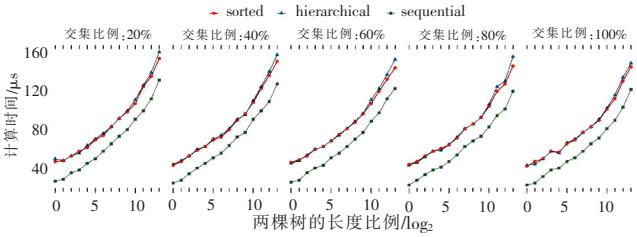


图 4　三种不同的遍历方式对 k-ary 搜索树求交效率的影响

Fig. 4　Intersection efficiency of three traversal techniques

2)剪枝优化　接下来,本文加入剪枝算法。如图 5 所示,保留 sequential 作为基准,优化后的方法标记为"_opt"。可以看出结合剪枝算法后,效率提升还是很明显的,减少了 20% 左右的执行时间。而随着树长度的增大,性能差异也越来越大。这三种优化方法几乎没有区别,其中 hierarchical 最快,甚至略优于 sequential。注意到求交时间对交集比例不敏感,本文期望在 skip 的帮助下,小的交集比例可以更快地导致提前终止,但是,算法为了确认所有可能的匹配,仍然不可避免地需要进行必要的比较。

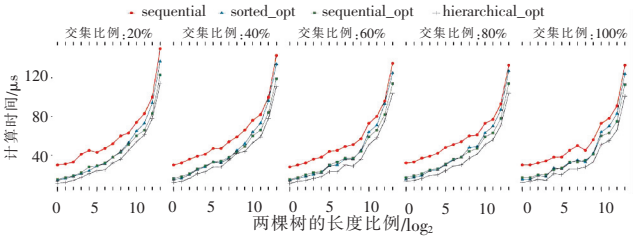


图 5　加入剪枝优化的三种遍历方式对 k-ary 搜索树求交效率的影响

Fig. 5　Intersection efficiency of three traversal techniques with pruning optimizations

表 1 中,将 skip 和 narrow 的性能提升分开,无剪枝表示没有优化的原始方法,opt 是结合两种剪枝算法后的结果。sorted 中的空白是由于它是单边算法,skip 并不适用。可以看到,skip 和 narrow 都有助于提高效率,narrow 比 skip 更明显,这也与图 5 中的结果一致。这种差异可以归因于 skip 在被搜索树中截断子树,而 narrow 作用在搜索树中为每次搜索计算 LCA,而缩短搜

索树的长度比搜索树更重要。注意到算法的平均时长小于 100 μs,这与 GOV2 中一半的列表长度小于 2^{10} 这一事实一致,即网页数据集的数分布中,短文档的长度总是远大于长文档,通过关注短倒排列表,优化算法总是可以获得较好的平均结果。

表 1 k-ary 搜索树在不同剪枝优化下的平均求交时间

Tab. 1 Average intersection time of k-ary tree with optimizations /μs

遍历方式	无剪枝	skip	narrow	opt
hierarchical	84.14	57.54(-21.5%)	41.13(-43.9%)	39.39(-46.3%)
sequential	73.34	77.20(-8.2%)	63.44(-24.6%)	56.57(-32.8%)
sorted	86.47	-	63.53(-26.5%)	-

3)总体比较 最后分析了节点大小 k 引起的性能差异,并与另一种基于 SIMD 的求交算法进行了比较,而它也是当前在排序数组上最快的求交算法 svs_opt[18]。该算法使用了一种双尺度搜索算法,能够根据求交的数组大小自适应调整搜索窗口以获得最佳效率。为确保公平性,同样使用 C++ 编译并在相同的实验环境进行测试。结果如图 6 所示。

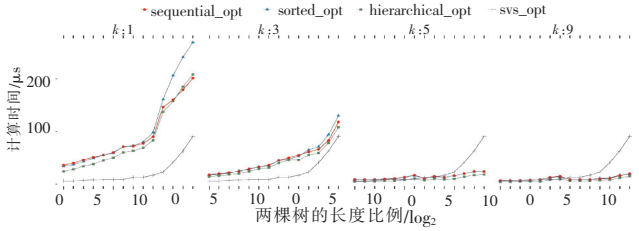


图 6 不同的节点大小(k)对 k-ary 搜索树求交效率的影响

Fig. 6 Intersection efficiency on k-ary tree with different node size k

不同的 k 也对应着一次载入 SIMD 寄存器的元素个数。由于硬件限制,实验条件中的寄存器最多可容纳 256 位,即 k 为 9,寄存器将同时载入并比较输出 k 个元素。令 $k=1$ 表示不使用 SIMD 并行指令,即在每次仅读入比较 1 个元素。注意 svs_opt 可以动态选择所使用的 SIMD 寄存器,它与 k 无关,因此在四个子图中形状保持不变。而对于 k-ary 搜索树而言,SIMD 指令可以显著加快算法速度。在一开始,svs_opt 性能优于 k-ary 搜索树求交,当 $k>5$ 时,svs_opt 开始失去了优势,特别是对于大于 2^{10} 的长度比例,其计算时间急剧增加。而 k-ary 搜索树在增大 k 值时计算时间明显下降,也侧面证明 k-ary 搜索树具有 SIMD 友好性。但在 $k=5$ 和 $k=9$ 之间,差异并不明显,其原因可以归结于:当搜索深入到较深层次时,树枝的分叉过多,导致的缓存未命中将不可避免,并超过了 SIMD 带来的增益。另外,考虑到被搜索树的大小固定为 2^{10},其相应的树结构看起来更平坦,k 值越大,树的深度就越小。当被搜索树继续变大时,$k=5$ 和 $k=9$ 之间的性能差距将再次增大。综上所述,k-ary 搜索树的优势在于求交数据集差异较大和 $k>5$ 时,甚至超越了 svs_opt。

5 结束语

本文研究了针对 k-ary 搜索树求交的优化算法,具体来说,本文提出了 skip 和 narrow 两种剪枝方法分别用于剪除两棵求交树中的无效节点,并结合三种遍历方式对剪枝效果进行了测试。实验证明,hierarchical 的遍历方式结合剪枝算法,可以减少近一半的执行时间,而配合更大的 SIMD 寄存器和倒排链长度,其性能甚至远优于传统的排序数组求交算法。为提高 k-ary 搜索树的适用性,引入压缩算法减少其空间占用,研究高效的范围查询和排序查询算法也是值得关注的课题。

参考文献:

[1] Zobel J, Moffat A. Inverted files for text search engines[J]. ACM Computing Surveys, 2006, 38(2): 6.

[2] Manning C D, Raghavan P, Hinrich S. Introduction to information retrieval[M]. Beijing: People's Posts and Telecommunications Press, 2010.

[3] Fox J, Green O, Gabert K, *et al*. Fast and adaptive list intersections on the GPU[C]//Proc of IEEE High Performance Extreme Computing Conference. Piscataway, NJ: IEEE Press, 2018.

[4] Zhang Jiyuan, Lu Yi, Spampinato D G, *et al*. FESIA: a fast and SIMD-efficient set intersection approach on modern CPUs[C]//Proc of the 36th IEEE International Conference on Data Engineering. Piscataway, NJ: IEEE Press, 2020: 1465-1476.

[5] Lemire D, Boytsov L, Kurz N. SIMD compression and the intersection of sorted integers[J]. Software: Practice and Experience, 2016, 46(6): 723-749.

[6] Awad M A, Ashkiani S, Johnson R, *et al*. Engineering a high-performance GPU B-tree[C]//Proc of the 24th Symposium on Principles and Practice of Parallel Programming. 2019.

[7] Wu Wei, Li Bin, Chen Ling, *et al*. k-ary tree hashing for fast graph classification[J]. IEEE Trans on Knowledge and Data Engineering, 2018, 30(5): 936-949.

[8] Song Chen, Hao Ruixia. Antimagic orientations for the complete k-ary trees[J]. Journal of Combinatorial Optimization, 2019, 38(4): 1077-1085.

[9] Sprenger S, Schäfer P, Leser U. BB-Tree: a main-memory index structure for multidimensional range queries[C]//Proc of the 35th IEEE International Conference on Data Engineering. Piscataway, NJ: IEEE Press, 2019: 1566-1569.

[10] Schlegel B, Gemulla R, Lehner W. k-ary search on modern processors [C]//Proc of International Workshop on Data Management on New Hardware. New York: ACM Press, 2009.

[11] Vigna S. Quasi-succinct indices[C]//Proc of International Conference on Web Search and Data Mining. New York: ACM Press, 2013: 83-92.

[12] Lemire D, Kaser O, Kurz N, *et al*. Roaring bitmaps: implementation of an optimized software library[J]. Software: Practice and Experience, 2018, 48(4): 867-895.

[13] Kraska T, Beutel A, Chi E H, *et al*. The case for learned index structures[C]//Proc of SIGMOD. 2018: 489-504.

[14] Oosterhuis H, Culpepper J S, De Rijke M. The potential of learned index structures for index compression[C]//Proc of the 23rd Australasian Document Computing Symposium. 2018: 1-4.

[15] Kim C, Chhugani J, Satish N, *et al*. FAST: fast architecture sensitive tree search on modern CPUs and GPUs[C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2010.

[16] Wang Jianguo, Lin Chunbin, He Ruining, *et al*. MILC: inverted list compression in memory[J]. Proceedings of the VLDB Endowment, 2017, 10(8): 853-864.

[17] Inoue H, Ohara M, Taura K. Faster set intersection with SIMD instructions by reducing branch mispredictions[J]. Proceedings of the VLDB Endowment, 2014, 8(3): 293-304.

[18] Song Xingshen, Yang Yuexiang, Li Xiaoyong. SIMD-based multiple sets intersection with dual-scale search algorithm[C]//Proc of ACM Conference on Information and Knowledge Management. New York: ACM Press, 2017.

[19] 闫宏飞,张旭东,单栋栋,等. 基于指令级并行的倒排索引压缩算法[J]. 计算机研究与发展, 2015, 52(5): 995-1004. (Yan Hongfei, Zhang Xudong, Shan Dongdong, *et al*. SIMD-based inverted index compression algorithms[J]. Journal of Computer Research and Development, 2015, 52(5): 995-1004.)

[20] 宋省身,杨岳湘,江宇. 基于单指令级并行的快速求交算法[J]. 山东大学学报: 理学版, 2018, 53(3): 54-62. (Song Xingshen, Yang Yuexiang, Jiang Yu. Efficient multiple sets intersection using SIMD instructions[J]. Journal of Shandong University: Natural Science, 2018, 53(3): 54-62.)