DOI:10.11721/cqnuj20140623

基于空间压缩的外存字符串词典索引算法研究*

曹彦婷
(苏州农业职业技术学院 信息与机电工程系，江苏 苏州 215008)

摘要:随着大数据时代的到来,大规模的数据需要处理,其中字符串词典数据占据着越来越大的比重。现有的字符串词典索引,不是空间占用过大,就是数据的本地引用性能差,无法高效地应用到外存磁盘环境。针对这些问题,提出了一种具有高效 I/O 的压缩索引 SB-trie,该索引不仅解决了以往索引空间占用过大的问题,同时也具有良好的本地引用性能,能够很好地工作于外存磁盘环境。实验结果表明,相比现有的索引,该索引不仅在空间上得到了有效的压缩,同时在磁盘环境下具有高效的查找性能。

关键词:字符串词典索引;外存数据结构;空间压缩;Trie;大数据处理

中图分类号:TP311.12　　**文献标志码:**A　　**文章编号:**1672-6693(2014)06-0107-09

近年来,随着互联网和移动设备的迅猛发展和大数据时代的来临,大规模的数据需要处理,其中文本数据占据着越来越大的比重。字符串词典索引作为文本索引的基础,其应用无处不在,如 RDF 图[1]、IP 数据包分类[2]、网络搜索引擎和生物信息计算等。

面对大规模文本数据的挑战,主要有两个解决思路:一个是设计更加高效的外存数据结构,提高数据的本地引用,把数据放在外存中,每次只读取需要的部分到内存中进行处理,确保高效的 I/O 操作;另一个是对数据进行压缩,使得在相同存储资源的情况下,能存储和处理更多的数据。

在外存索引(思路一)方面,Ferragina 等将 Patricia Trie 和 B+tree 相结合提出了 String B-tree[3],解决了当字符串键过长而导致的性能下降问题。但由于该索引将标签独立存储,每次 B+tree 节点内查找都需要两次读取外存,所以在字符串键的平均长度小于 1 000 B 的情况下,时间性能不如 B+tree。而且由于采用隐式指针的方法来表示 Patricia Trie,其空间占用仍旧较大。Askitis 等将 Burst Trie 扩展到外存,提出了 B-trie[4],该索引采用两层结构,上层根节点采用数组表示的 Trie,存储字符串键的共同前缀,下层叶子节点采用基于折半查找的简单映射结构,存储字符串键去掉共同前缀后得到的后缀。这种方法在所有字符串键都比较接近的情况下,才可能有较好的空间利用率,而大部分情况下空间性能不如 B+tree。在工业界,针对 B+tree 索引空间占用过大的问题,广泛采用 FC(Front coding)编码对节点进行压缩,在一定程度上解决了空间性能问题。但随着大数据的到来,数据规模持续扩大的情况下,其空间占用仍旧太大。Ferragina 等从理论的角度提出了一种缓存无关的索引结构[5],该结构具有良好的本地数据引用,同时空间也得到了压缩。但是由于该索引过于复杂,至今没有相关的实现和实验,验证其实际的时间和空间性能。以上这些索引普遍存在着空间占用过大的问题,而采用 FC 压缩的方法,并不能有效地解决空间占用的问题。

在压缩索引(思路二)方面,Klein 等通过将 FC 和 Huffman 编码结合,提出了一种在压缩状态下直接进行字符串匹配的方法[6]。Grossi 等提出了一种基于路径分割的方法来压缩存储字符串词典[7],Arz 等提出了基于 LZ(Lempel-Ziv)压缩的方法来对字符串集合索引进行压缩[8],Brisaboa 等通过大量的实验[9],对比了 5 种字符串集合索引的时间和空间性能。上面的这些方法虽然在空间性能上面实现了一定的改进,但在大数据的情况下,压缩后的索引仍旧太大,无法全部放入内存,这个时候仍旧需要外存的配合,而这些索引的本地引用性能差,都无法在外存环境下进行高效的 I/O 操作。

* **收稿日期:**2014-03-14　**修回日期:**2014-05-08　**网络出版时间:**2014-11-19 21:49
资助项目:江苏省自然科学基金(No. BK2011281);苏州市应用基础研究计划(No. SYG201241)
作者简介:曹彦婷,女,讲师,研究方向为数据索引和压缩软件工程,E-mail:caoyanting1980@163.com
网络出版地址:http://www.cnki.net/kcms/detail/50.1165.N.20141119.2149.023.html

针对现有索引存在的上述问题,本文设计了 Patricia Trie 的压缩表示,然后将其与 B-trie 相结合,提出了一种新的字符串词典索引 SB-trie (Succinct B-trie),该索引不仅在外存环境具有高效 I/O 的特点,同时解决了空间占用过大的问题。实验表明,相比现有的其他索引,该索引不仅在空间上得到了有效的压缩,同时在磁盘环境下具有高效的查找性能。

本文第 2 节介绍了一些现有的基础数据结构和算法,第 3 节介绍了设计的 Patricia Trie 压缩表示及对应的查找算法,第 4 节介绍了 SB-trie 的结构及其相关的构建和查找算法,第 5 节是实验结果和分析,最后是本文的总结。

1 基础结构和算法

1.1 比特数组

假设 $B[1,n]$是一个长度为 n,由 0 和 1 构成的数组,定义以下操作:

1)$B[i]$:数组 B 中第 i 个元素,$1\leqslant i\leqslant n$。

2)$rank(B, i, b)$:$B[0,i]$中比特 b 出现的个数,$b\in\{0,1\}$,$1\leqslant i\leqslant n$。

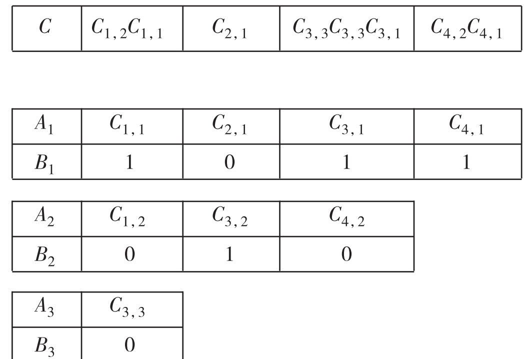
3)$select(B, i, b)$:$B[1, n]$中第 i 个比特 b 出现的位置。

最早由 Jacobson 提出[10],后来经过 Raman 等的改进[11],以上操作可以在 $O(n)$的额外空间下,实现 $O(1)$的时间性能。

1.2 DAC

给定一串连接起来的变长码序列,比如 Huffman 编码,DAC (Directly addressable codes)通过重组该序列[12],实现了对任意变长码的直接读取,而且所花费的代价也非常小。其构造和读取原理如下。

假设 $C[1,n]$是 n 个变长码,首先从各个变长码中取出头部等比特长作为一个符号,存到第一个数组 $A_1[1,n]$,同时另外还需要一个比特数组 $B_1[1,n]$,当第 i 个变长码 C_i 已经全部存储在 $A_1[i]$中了,$B_1[i]$标 0,如果 C_i 还有余下的部分,$B_1[i]$标 1。然后从 C 中去掉存储完成的变长码,重新开始下一级的存储,方法和前面一样。图 1 是一个例子。



C	$C_{1,2}C_{1,1}$	$C_{2,1}$	$C_{3,3}C_{3,3}C_{3,1}$	$C_{4,2}C_{4,1}$

A_1	$C_{1,1}$	$C_{2,1}$	$C_{3,1}$	$C_{4,1}$
B_1	1	0	1	1

A_2	$C_{1,2}$	$C_{3,2}$	$C_{4,2}$
B_2	0	1	0

A_3	$C_{3,3}$
B_3	0

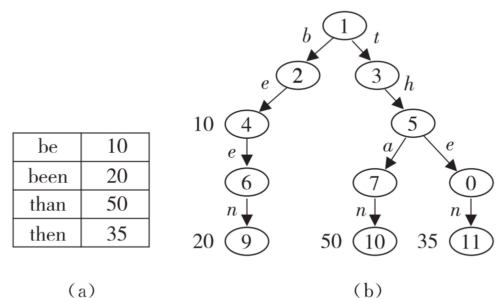
图 1 变长码数组 C 的 DAC 表示

读取的时候,假设要读取第 i 个变长码 C_i,首先取出 $A_1[i]$,同时读取 $B_1[i]$,如果 $B_1[i]$是 0,该变长码已经读取完成,如果 $B_1[i]$是 1,表示还有余下的部分,而且下一段存储在 A_2 的 $rank(B_1, i, 1)$位置,以此类推。以图 1 的例子中查找 C_4 为例,首先在 $A_1[4]$中找到 $A_{4,1}$,然后因为 $B_1[4]$是 1,说明还有余下的部分,$rank(B_1, 4, 1)$为 3,所以第二部分在 $A_2[3]$,因为 $B_2[3]$为 0,读取结束,最后结果 C_4 为 $C_{4,2}C_{4,1}$。

1.3 Trie

Trie 是一种分支有序的多路树,每个节点包含一个值和多个分支,每个分支对应一个不同的字符标签,从根节点到某个节点路径上的标签构成一个字符串键,与该字符串键相关联的值存储在该节点的值域中。

图 2 是一个典型的 Trie,通过观察发现 Trie 的一个重要的特点是,具有共同前缀的字符串键聚合在一起,共享前缀节点,这样减少了节点的数量,从而减少了空间占用,其效果类似于 FC 编码的压缩。与 FC 编码不同的是,Trie 支持快速查找。基于折半查找的结构,时间复杂度为 $O(|P|\lg n)$,查找性能与集合中的字符串键的个数有关,而 Trie 查找的时间复杂度为 $O(|P|)$,查找只与字符串键的长度相关,所以比折半查找更优。



be	10
been	20
than	50
then	35

图 2 (a)一个简单的字符串词典;(b)对应的 Trie

给定一个字符串键 K_i,K_i可能是另一个字符串键 K_j 的前缀,所以与字符串键相对应的值可能出现在 Trie 树的任意一个节点中。本文定义尾节点如图 2 所示。

定义 1 Trie 中某个字符串键 K 对应的路径的最后一个节点,即与该字符串键相对应的值所在的节点,称

为 K 在 Trie 中的尾节点。

Trie 的准确查找如算法 1 所示。第 3)～7)步的循环变量为 i,P 的前缀 $P[1, i]$存在于 Trie 中,且 node 是字符串 $P[1, i]$的尾节点。第 8)步通过访问节点的值域确定 P 确实存在,还是其他字符串键的一个前缀。这里以图 2 中的 Trie 举例,假设 P=“than”,node 依次为 1,3,5,7,10 节点,最后在节点 10 中找到与 P 相关联的值。该算法的时间复杂度为 $O(|P|)$。

算法 1　准确查找 P,返回 P 关联的值。

输入:Trie 树根节点 *root*,要查找的模式 P。

输出:Trie 中与 P 关联的值,如果不存在,返回 null。

1)*node*:=*root*;

2)i := 1;

3)WHILE $i <= |P|$

4)*node*:=找到 node 中对应 $P[i]$的孩子节点

5)IF *node* is null, RETURN null

6)i:= $i + 1$

7)END WHILE

8)RETURN node 节点中的值

由于 Trie 中每个节点对应的字符串是确定的,通过 Trie 中的某一个节点可以找到以该节点为尾节点的字符串,所以可以用一个节点来标识一个字符串。通过节点找到对应的字符串的算法如算法 2 所示。其中第 3)～8)步循环变量是 *node*,节点 tail 到节点 node 路径上的字符串为 *key*,node 节点到 root 节点是还没有收集到的字符串。以图 2 中的 Trie 为例,假设 tail 是 11 号节点,那么 node 依次经过 11、8、5、3、1,期间将路径上遇到的字符依次添加到 *key* 的末尾,最终返回 P=“then”。和算法 1 类似,该算法的时间复杂度是 $O(|P|)$。

算法 2　ComputeKeyString

输入:Trie 的根节点 *root* 和其中某一个节点 *tail*。

输出:Trie 中与 tail 节点相对应的字符串。

1)*node*:= *tail*

2)*key*:= “”

3)WHILE *node* ! =*root*

4)*parent*:= *Parent*(*node*)

5)*ch*:= *parent* 的分支中与 node 节点相关联的字符标签

6)*key*:= *key* + *ch*

7)*node*:=*parent*

8)END WHILE

9)RETURN *key*

1.4 Trie 的 LOUDS 表示

传统的基于指针的 Trie 表示方法,空间占用过大,尤其是在 64 位机上。下面介绍一种新的 Trie 的压缩表示,LOUDS (Level-order unary degree sequence)[13]。假设一个 Trie 节点有 3 个分支,该节点可以用码 1 000 表示,3 个 0 表示 3 个分支,前面的 1 表示该节点的开始。对 Trie 进行宽度优先遍历,将每个节点的码表示连接起来,形成的一个二进制序列就是该 Trie 的 LOUDS 表示。为了计算的方便,一般给 Trie 的根节点添加一个父节点,作为超级根节点,所以对应的 LOUDS 序列的最前面为超级根节点添加 10。

LOUDS 仅仅记录了 Trie 的树结构信息,另外还需要以下辅助结构:

1)*Labels*:字节数组,以宽度优先遍历的顺序存储每个节点各个分支对应的字符标签。

2)*Values*:整数数组,以宽度优先遍历的顺序记录节点值域内的信息。

图 3 是图 2 中的 Trie 对应的 LOUDS 表示。本文约定采用节点码的第一个比特 1 在 LOUDS 中的位置来标识该节点。利用 LOUDS 的性质,可以方便地在 $O(1)$的时间复杂度下实现树的基本操作。

节点编号			1			2		3		4		5			6		7		8		9	10	11
Values			*N*			*N*		*N*		*N*		*N*			*N*		*N*		*N*		20	50	35
Labels				*b*	*t*		*e*		*h*		*e*		*a*	*e*		*n*		*n*		*n*			
LOUDS	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	1	1

图 3 前面图 2 中的 Trie 树对应的 LOUDS 表示,*N* 表示 null,空值

1)$Branch(id, i)$,节点 id 的第 i 个孩子节点:$louds.select1(louds.rank0(id + i))$。

2)$Parent(id)$,节点 id 的双亲节点:$louds.select1(louds.rank1(louds.select0(louds.rank1(id) - 1)))$。

3)$Degree(id)$,节点 id 的度:$louds.select1(louds.rank1(id) + 1) - id - 1$。

2 Patricia Trie 的压缩表示

本节首先介绍了 Patricia Trie 的准确查找和下界查找算法,这两个算法将在第 4 节介绍的 SB-trie 查找算法中用到。然后介绍了 Patricia Trie 的压缩表示,以及在压缩表示的情况下分支节点匹配算法。

2.1 Patricia Trie

Patricia Trie 是一种特殊的 Trie,它是针对原始 Trie 空间占用过大问题的一种改进。原始的基于指针表示的 Trie,存在着许多只有一个分支的节点,这些节点本身不起到索引的作用,却含有大量的指针,占用了大量的空间。Patricia Trie 便是将原始 Trie 中的只有一个分支的节点去掉,同时将分支标签从原来的一个字符推广到字符串得到的。图 4 是前面图 2 的 Trie 对应的 Patricia Trie。

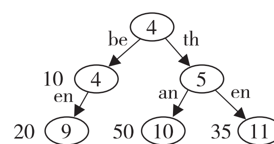


图 4 前面 Trie 对应的 Patricia Trie

Patricia Trie 的准确查找算法和 Trie 类似,区别就是原来的字符标签变成了字符串标签,其算法如算法 3 所示。算法从根节点开始,用 P 逐个匹配 Patricia Trie 中的节点,其中的循环变量为 i,$P[1, i-1]$与 Patricia Trie 中 node 节点所对应的字符串匹配。以图 4 的 Patricia Trie 为例,假设 P 为“than”,那么 node 依次为 1、5、10 节点,最后在 10 节点找到 P 对应的值。该算法的时间复杂度为 $O(|P|)$。

算法 3 ExactMatch

输入:根节点 *root*,要查找的模式 P。

输出:Patricia Trie 中与 P 相关联的值。

1)$node := root$

2)$i := 1$

3)WHILE $i < |P|$

4)$node, i := MatchChild(node, P, i)$

5)IF *node* is null, RETURN null

6)END WHILE

7)RETURN *node*

除了准确查找外,本文下面还需要用到下界查找 *LowerBound*。首先看一个与 Trie 的下界查找密切相关的性质。

性质 1 深度优先遍历 Trie 输出的字符串键序列是以词典序递增排列的有序序列。

证明 由于 Trie 是一种分支有序的多路树,Trie 树每个节点下的多个分支都是有序的,Trie 树建立的过程即是排序的过程,所以不管图 2(a)输入顺序如何,建立的 Trie 树是一个固定的形状,如图 2(b)所示。深度优先遍历图 2(b)的 Tire 树得到的顺序是:be、been、than、then,是以词典序递增排列的有序序列。

然后下界查找定义如下:

定义 2 给定一个 Trie,内包含的字符串集合 $S=\{s_1, s_2, \cdots, s_n\}$,这里的 S 按照词典序非递减有序排列。给定任一关键字 P,在 Trie 中查找第一个不小于 P 的键 s_i,其中 i 是该键在 S 中的排名。返回 s_i 的键值对。

以上针对 Trie 的定义和性质对于 Patricia Trie 同样适用。利用 Patricia Trie 深度优先遍历的有序性,Patricia Trie 的下界查找的算法如算法 4 所示。该算法前面部分与准确匹配一致,循环第 3)~7)结束后 node 节点是 Patricia Ttrie 中找到的最长匹配 P 的节点,此时该 Patricia Trie 被分成 3 个部分,以 node 为根的子树为第一部

分,余下的部分被从 node 到 root 节点的路径分割成左右两个部分,左侧部分 L 全部小于 P,右侧部分 R 全部大于 P。接下来有以下几种情况:

1)$i > |P|$,即 P 已经全部匹配完成,这个时候只要通过 *LeftMostNode* 直接找到以 node 为根节点的子树中的最小键的节点即可。

2)$i <= |P|$,即前缀 $P[1, i-1]$匹配成功,node 节点是第一个无法匹配的节点。这个时候,通过 *ChildLowerBound* 在 node 节点的分支中找到第一个不小于 $P[i,|P|]$的分支 next,如果找到,接下来找到以 next 节点为根节点的子树中的最小键节点即可。如果没有找到,说明以 node 为根的子树中的字符串都小于 P,这时要找的就是右侧部分 R 中的最小键。通过 *GeneralizedSibling* 找到 R 中最小字符串的尾节点,如果没有找到,说明 R 是空的,即要找的键值对不存在。如果查找成功,只要在找到的节点的子树中找到最小键节点即可。

算法 4　LowerBound,Patriacia Trie 下界查找

输入:根节点 *root*,要查找的模式 P。

输出:Patricia Trie 中第一个不小于 P 的键值对的值。

1)*node*:= *root*

2)i:= 1

3)WHILE $i <= |P|$

4)*next*,i:= *MatchChild* (*node*, P, i)

5)IF *next* is null, BREAK

6)*node*:= *next*

7)END WHILE

8)IF $i <= |P|$

9)*next*:=*ChildLowerBound*(*node*, $P[i, |P|]$)

10)IF *next* is null

11)*next*:= *GeneralizedSibling*(*node*)

12)IF *next* is null, RETURN null

13)*node*:= *next*

14)*node*:= *LeftMostNode*(*node*)

15)RETURN node 节点中的值

2.2 PatriciaTrie 的压缩表示

假设 Patricia Trie 中某个节点对应的字符串标签为 $s=as'$,其中 a 是一个字符,s'是除去 a 余下的部分。将 Patricia Trie 中各个节点的字符串标签的 s'部分忽略,Patricia Trie 便成了原始的 Trie,可以采用前面 1.3 节介绍的 LOUDS 表示法进行存储。另外的 s'的集合便构成了一个新的字符串集合,可以采用 1.3 节的方法,存储在另外一个 Trie 中,该 Trie 采用 LOUDS 表示进行存储,这样只需要在 Patricia Trie 的节点中保存 s'在 Trie 中的尾节点地址即可。

由于将 Patricia Trie 的分支标签分成了两部分保存,所以匹配孩子节点的时候要分成两步,第一步和 Trie 一样,找到当前节点中匹配首字符的孩子节点,第二步匹配孩子节点中保存的余下的字符串标签,伪代码如算法 5 所示。

算法 5　MatchChild,匹配 Patricia Trie 中孩子节点

输入:当前节点 *id*,模式 P,当前要匹配的 P 的后缀的开始位置。

输出:节点 *id* 中与 P 中 *id* 开始的字串匹配的孩子节点,下次要匹配的 P 的后缀的开始位置。

1)*offset*:= *louds.rank*0(*id*)

2)*degree*:= *Degree*(*id*)

3)*rank*:= 在字符数组 $Labels[offset, offset + degree - 1]$中找到 $P[i]$的位置

4)IF *rank* is null, RETURN null, i

5)*child*:= *Branch*(*id*, *rank*)

6)*label*:= 计算 *child* 节点标签的 s'部分

7)IF *label* is prefix of $P[i+1, \cdots]$

8)RETURN *child*, i + length(*label*)

9)RETURN null, i

节点中下界查找 ChildLowerBound 算法如算法 6 所示,前面类似算法 5,后面第 7)～10)步确保分支字符串标签不小于 P,如果小于 P,找到该孩子节点右侧的第一个兄弟节点。

算法 6 ChildLowerBound 在 Patricia Trie 节点中找到第一个不小于 P 的孩子节点

输入:节点 *id*,模式 P。

输出:节点 *id* 中第一个不小于 P 的孩子节点。

1)*offset*:= *louds.rank*0(*id*)

2)*degree*:= *Degree*(*id*)

3)*rank*, *ch*:= 在 *Labels*[*offset*, *offset* + *degree*−1]中找到第一个不小于 $P[1]$的位置,并返回该字符

4)IF *rank* is null, RETURN null

5)*child*:= *Branch*(*id*, *rank*)

6)*label*:= 计算 *child* 节点标签的 *s'*部分

7)*label*:= *ch* + *label*

8)IF P <= *label*

9)RETURN *child*

10)RETURN *Sibling*(*child*)

2.3 进一步压缩

Patricia Trie 中有的节点的 *s'*是空字符串,所以保存 *s'*信息的整数数组 *Links* 是一个稀疏数组,可以进行压缩。本文采用一个支持 rank 操作的比特数值 *HasLink* 和另外一个整数数值 *ValidLinks* 表示。这样在访问第 i 个整数的时候,可以先查看 *HasLink*[i]的值,如果是 0,表示 *s'*为空,如果为 1,*ValidLinks*[*HasLink.rank*1(i)]就是 *s'*在 Trie 中的位置。

由于保存 *s'*信息的 Trie 采用 LOUDS 表示进行存储,Trie 中节点的编号是采用宽度优先的顺序进行的,较短的字符串键的标识也较短。同时 Patricia Trie 中各个节点的 *s'*长度普遍较短,所以保存位置信息的 *ValidLinks* 里的整数,大部分都数值较小,所以可以用 DAC 对其进行进一步压缩。

同样的方法可以应用到 *Values* 稀疏数组。采用这种方式进行压缩后的,索引的空间占用得到了有效较少,而且不影响 Patria Trie 的查找。

3 SB-trie

本节介绍如何将前面描述的 Patricia Trie 的压缩表示和相关的分支匹配算法与 B-trie 相结合,实现 SB-trie 的压缩索引。

3.1 外存模型

为了更好地描述下文中外存算法的性能,首先简单介绍一下外存模型(External memory model)[14]。图 5 是一个简单的外存模型示意图。外存模型是用于分析外存算法时,对计算机的一种抽象模型,主要包括一个无限空间的外存磁盘,一个空间为 M 的内存和一个 CPU。每次 I/O 读写操作都在外存和内存之间传输连续的数据 B,B 是内存和外存进行数据传输的最小单位块的大小,明显 $1 \leqslant B \leqslant M$。算法的时间性能以 I/O 的数量来度量,不考虑 CPU 计算和访问内存的时间。

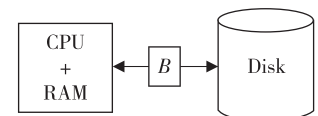


图 5 外存模型 SB-trie 的构建

3.2 SB-trie 的构建

SB-trie 的结构主要分成两个部分,外部结构采用类似的 B-trie 结构,内部结构即节点内部采用压缩表示的 Patricia Trie,同时将节点 Patricia Trie 的字符串标签集中存储到单独的一个 Trie 中,所有的 Patricia Trie 和 Trie 采用 LOUDS 表示法进行压缩存储。SB-trie 采用类似 B+tree 批量构建的方法构建外部结构。假设输入字符串键集合为 $K=\{K_1, \cdots, K_n\}$,K 以词典序递增的顺序排列。首先对 K 进行分组,每组包含相邻的字符串键,每个组构造一个 Patricia Trie,并且确保每个小组占用的空间小于等于 B,把这些小组作为 SB-trie 的叶子节

点。然后取出每个小组内最大的键作为新的键,该小组在文件中的位置作为新的值,形成一个新的键值对的集合 K',为 K' 构造一个 Patricia Trie 作为 SB-trie 的根节点。图 6 是一个大致的 SB-trie 布局示意图。

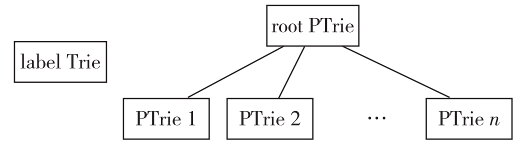


图 6　SB-trie 结构布局

3.3 SB-trie 的查找

SB-trie 的查找算法如算法 7,首先第 1)步与 K 中的最大字符串键 K_n 比较,如果 P 大于 K_n,则该 SB-trie 中不存在 P,返回空值。第 2)步中的 *LowerBound* 是从 node 节点中找到 P 所在的那个叶子节点的位置,如果没有找到,则查找失败。如果找到,从文件读取叶子节点,最后 *ExactMatch* 在叶子节点中准确查找 P,如果 P 存在,返回和 P 相关联的值,否则返回 null。因为 SB-trie 只有一层索引结构,每次查找只需要读取磁盘一次,所以本算法在外存模型上的时间复杂度为 $O(1)$。

算法 7　Find,在 SB-trie 查找 P

输入:SB-trie 的根节点 *root*,SB-trie 中存在的最大的字符串键 K_n,搜索的模式 P

输出:SB-trie 中与 P 相关联的值,如果不存在,返回空值。

1)if $P > K_n$, RETURN null

2)$offset := LowerBound(root, P)$

IF *offset* is null, RETURN null

3)$leafNode$:= 读取 *offset* 位置的节点

RETURN $ExactMatch(leafNode, P)$

4 实验

本节通过将本文实现的 SB-trie 与其他现有的字符串词典索引算法实现进行对比实验,说明 SB-trie 的实际时间和空间性能。其他的索引算法实现有:

B+tree,最经典的外存索引算法,主要用于整数等定长键的索引。在不超过外存块大小的情况,本文通过将变长的字符串键尽可能多地存储到节点中,采用文献[4]中的节点布局方法实现了一个简单高效的字符串键 B+tree。

FCB+tree,采用增量编码(Front coding)的 B+tree。在前面实现的字符串键 B+tree 的基础上,对节点内的字符串集合采用增量编码进行压缩,查找采用了文献[6]提出的查找算法。

String B-tree,Ferragina 等针对传统 B+tree 在字符串键长度过长情况下性能下降的问题提出的字符串词典索引算法。本文实验采用一个静态的开源实现[15]。

Leveldb,Google 的轻量级键值对存储的开源库,目前广泛使用的键值对存储,所以文本将其作为工业级字符串词典索引的代表,进行对比实验。

另外本文的 SB-trie 实现的比特数组采用了 sdsl 库的 RRR 算法实现,DAC 实现部分,本文在文献[12]提供的 C 实现的基础上进行的算法方面的优化。以上所有程序都采用 C++实现。

4.1 实验环境

本文的实验平台 CPU 是 Intel Core 2 Duo 2.4 GHz,内存 2 GB,硬盘日立 160 GB,5 400 rpm。操作系统为 Ubuntu 12.04 LTS 64 位,文件系统是 FAT32。所有 C++代码都用 g++4.6.3,以-O3 参数编译。

实验数据采用的是 2012 年 12 月 1 日的英文版维基百科的所有条目的条目名,总共 9 864 458 个条目,占用空间 201.7 MB,实验中近似表示为 10 000 千条数据集,其他较小的数据集是通过截取该数据集前面的若干条数据生成的。

4.2 实验结果和分析

本文分别对 5 种索引在不同数据规模下进行了查找操作的对比实验,每次程序运行前都重新挂载实验分区,并运行辅助程序以最大限度排除缓存对实验的干扰。本文用原始数据来对索引运行查找操作,每隔 1 000 个数据查找一次,连续查找一定次数后的总时间取平均值,得到图 7。图 8 是各个索引的占用空间与原始数据的对比图。下面分别对各个索引进行分析如下。

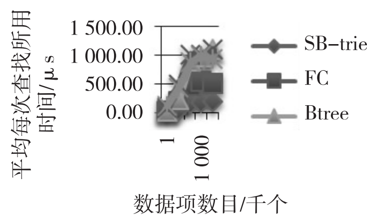


图7 查找时间对比

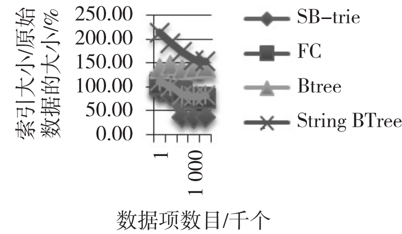


图8 索引的空间占用对比

B+tree,由于直接将字符串键保持在节点内部,没有进行任何的压缩,所以从实验结果可以看到,其空间占用基本维持在原始数据大小的140%左右,而且与数据量的大小没有明显的关系。而查找时间性能方面,由于采用的是多级索引方式,在数据量较小的时候,相比其他索引更快,但随着数据量的增大,花费的时间变长。

FC B+tree,由于在B+tree的基础上对节点内的字符串键采用增量编码进行了一定的压缩,所以可以看到随着数据量的增大,相应的空间占用比越来越小。查找时间性能方面,虽然计算的复杂性随着压缩的加入增加了,但是由于空间的减少,所以需要读取外存的次数也相应减少。从实验结果看出,减少的外存读取时间补偿了计算复杂度的增加,所以最终整体的查找性能得到了提高。

String B-tree,和B+tree一样采用多级索引结构,而且节点内采用指针表示的Trie进行存储,所以实际的空间性能比B+tree更差。而查找时间性能方面,虽然采用Trie作为节点内的结构,相应的计算复杂度比采用折半查找的方式更快,但由于空间占用增大,导致读取外存的次数增加,总的查找性能比其他索引差。因为String B-tree主要是针对B+tree在字符串键很长的情况下所作的优化,而实验采用的数据的字符串键普遍较短,所以出现这个结果也是可以意料的。

Leveldb,由于采用了Snappy通用压缩库对存储数据进行了一定的压缩,所以随着数据量的增大其空间占用比明显减小,其效果接近FC B+tree。查找时间性能方面,由于其完全地支持插入删除等动态操作和其他复杂功能,所以其查找时间性能也相应差一些。

最后是本文的SB-trie。实验结果显示SB-trie在空间占用上远远小于其他索引,当数据量大于10万条时,空间占用稳定在原始数据的36%左右。201.7 MB的原始数据,其索引的空间占用只有73.1 MB。显著的空间压缩,主要有两方面的原因,一是采用了压缩表示的Patricia Trie。Patricia Trie本身通过共享相邻字符串键之间的共同前缀,在一定程度上减少了空间占用,然后采用LOUDS表示法对Patricia Trie进行压缩表示,LOUDS采用二进制数组表示Patricia Trie的拓扑结构,从而避免存储大量指针带来的空间消耗,大大减少了空间占用。二是将所有的SB-trie节点内Patricia Trie的字符串标签统一存储在另外一个LOUDS表示的标签Trie中,多个相同或者共享前缀的标签可以只存储一份,更进一步实现了压缩。

查找时间性能方面,虽然采用压缩表示的Patricia Trie增加了计算复杂度。但由于空间得到有效压缩,叶子节点数目相应减少,使得采用一层索引成为可能,这样每次查找只需要一次读取磁盘的操作。外存读取减少所节省的时间大大补偿了CPU计算复杂度提高增加的时间,所以当数据量大于10 000条时,查找效率明显优于其他索引。

5 结束语

大数据时代海量的文本数据对字符串词典索引算法提出了新的挑战。现有的字符串词典索引,不是空间占用过大,就是在外存环境下无法高效地进行I/O操作,无法有效地应对大数据的挑战。针对这个问题,本文设计了Patricia Trie的压缩表示,然后将其与B-trie相结合,提出了一种新的字符串词典索引SB-trie,该索引不仅解决了以往索引空间占用过大的问题,同时也具有良好的本地引用性能,能够很好地工作于外存磁盘环境。实验表明,相比现有的索引,该索引不仅在空间上得到了有效压缩,同时查找时间性能也更高。但是该索引只是静态索引,不支持动态添加和删除操作,所以如何对其进行扩展,以支持动态操作是更具挑战的问题。

参考文献:

[1] 杜方,陈跃国,杜小勇. RDF数据查询处理技术综述[J]. 软件学报,2013,24(6):1222-1242.

Du F, Chen Y G, Du X Y. Survey of RDF query processing techniques[J]. Journal of Software, 2013, 24(6): 1222-1242.

[2] 尚凤军，潘英俊，潘雪增. 基于随机分布的多比特 Trie 树 IP 数据包分类算法研究[J]. 通信学报，2008，29(7)：109-117.
Shang F J, Pan Y J, Pan X Z. Research on a stochastic distribution multibit Trie tree IP classification algorithm[J]. Journal on Communications, 2008, 29(7): 109-117.

[3] Ferragina P, Grossi R. The string B-tree: a new data structure for string search in external memory and its applications[J]. Journal of the ACM (JACM), 1999, 46(2): 236-280.

[4] Askitis N, Zobel J. B-tries for disk-based string management[J]. The VLDB Journal, 2009, 18(1): 157-179.

[5] Ferragina P, Grossi R, Gupta A, et al. On searching compressed string collections cache-obliviously[C]// Principles of database systems (PODS'08). [S. l.]: ACM press, 2008: 181-190.

[6] Klein S T, Shapira D. Compressed matching in dictionaries[J]. Algorithms, 2011, 4(4): 61-74.

[7] Grossi R, Ottaviano G. Fast compressed tries through path decompositions[J]. ALENEX, 2012: 65-74.

[8] Arroyuelo D, Navarro G, Sadakane K. Stronger lempel-ziv based compressed text indexing[J]. Algorithmica, 2012, 62(1-2): 54-101.

[9] Brisaboa N, Canovas R, Claude F. Compressed string dictionaries[J]. Experimental Algorithms, 2011: 136-147.

[10] Jacobson G. Space-efficient static trees and graphs[C]// Proceedings of the 30th annual symposium on foundations of computer science. NC: Research Triangle Park, 1989: 549-554.

[11] Raman R, Raman V, Satti S R. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets[J]. ACM Trans Algorithms, 2007, 3(4): 43-68.

[12] Brisaboa N R, Ladra S, Navarro G. DACs: bringing direct access to variable-length codes[J]. Inf Process Manage, 2013, 49(1): 392-404.

[13] Arroyuelo D, Cnovas R, Navarro G. Succinct trees in practice[J]. ALENEX, 2010: 84-97.

[14] Vitter J S. Algorithms and data structures for external memory[J]. Found Trends Theor Comput Sci, 2008, 2(4): 305-474.

[15] Text Mining Software. C++ string B-tree library[CP]. http://wikipedia-clustering.speedblue.org/strBTree.php, 2007.

Research on Algorithms for Dictionary Index in External Memory Based on Space Compression

CAO Yanting

(Department of Information and Mechatronics Engineering, Suzhou Polytechnic Institute of Agriculture, Suzhou Jiangsu 215008, China)

Abstract: With the coming of big data age, large scale of data needs to be processed, and string dictionaries are becoming a significant part of it. The existing string dictionary indexes are either too space-consuming, or lack of locality of access, making them inapplicable on the external disk environment settings. Targeted with these problems, we proposed a new string dictionary index data structure SB-trie, which is not only succinct on space, but also has good locality of access, making it I/O efficient on external memory settings. Experiments show that SB-trie consumes less space and has great searching performance on disk environment.

Key words: string dictionary; succinct data structure; space compression; Trie; big data processing

(责任编辑　游中胜)