前缀立方的索引*)

丁胡临　冯剑琳　聂　晶
(华中科技大学计算机科学与技术系　武汉 430074)

摘　要　前缀立方是最近提出的一种新的数据立方结构。它利用前缀共享和基本单元组有效地缩小了数据立方的尺寸,相应减少了数据立方的计算时间。为提高前缀立方的查询性能,本文提出了它的一种索引机制 Prefix-CuboidTree。文中用真实数据集和模拟数据集进行大量实验,证明了该索引机制的查询性能。

关键词　Prefix-CuboidTree,前缀立方,浓缩数据立方,基本单元组

Indexing PrefixCube

DING Hu-Lin　FENG Jian-Lin　NIE Jing
(Department of Computer Science, Huazhong University of Science and Technology, Wuhan 430074)

Abstract　Recently a new data cube structure called PrefixCube is proposed, which reduces the size of normal data cube and hence the data cube computation time efficiently by augmenting BST condensing with prefix-sharing. To improve query processing performance, in this paper, we propose a new mechanism called Prefix-CuboidTree for indexing a PrefixCube. Extensive experiments are conducted to demonstrate the effectiveness of Prefix-CuboidTree using both synthetic and real world data.

Keywords　Prefix-CuboidTree, PrefixCube, Condensed cube, Base single tuple

1　引言

为支持 OLAP 中复杂的多维查询,JimGray 等人在 1996 年提出了数据立方(DATA CUBE)和立方算子(CUBE BY)的概念[1],CUBE BY 是传统的 GROUP BY 算子的多维扩展,它计算了 CUBE BY 子句中各属性的所有可能组合所对应的 GROUP BY。一个 GROUP BY 被称为一个数据小方(cuboid)。

很明显,CUBE BY 是计算代价相当昂贵的算子,其计算结果远远超过了基表的大小。而且,随着维数和基表元组数的增多,数据立方会急剧增大。它的巨大尺寸不仅使存储开销变得很大,而且其生成过程中的大量 I/O 导致计算时间也非常可观。

为缩小数据立方的尺寸,减少存储和计算的时空开销,很多方法被提出以解决这方面的问题,如浓缩数据立方(Condensed Cube)[2],Dwarf[3],Quotient Cube[4] 和 QC-Trees[5]。这些方法的基本思想是尽量消除立方元组间的冗余值。文[3]明确指出立方元组间通常存在两种冗余,即前缀冗余和后缀冗余。

给定含三个维 A,B,C 的数据立方,显然,维 A 的每一个值都会出现在数据小方(A),(AB),(AC)和(ABC)中,这种冗余被称为数据小方间的前缀冗余。另外,在每一个数据小方(AB),(AC)和(ABC)中,维 A 的某些值也可能重复出现多次,由此产生的冗余被称为数据小方内的前缀冗余。

当属于不同数据小方的元组是由同一组基表元组计算得到时,就会产生后缀冗余。举一个极端的例子,假设基表 R 仅包含一条元组 $r(a_1, a_2, \cdots, a_n, m)$,则由 R 计算得到的数据立方中包含 2^n 条元组,分别为$(a_1, a_2, \cdots, a_n, V_1)$,$(a_1, *, \cdots, *, V_2)$,$(*, a_2, *, \cdots, *, V_3)$,$\cdots$,$(*, *, \cdots, *, V_m)$,其中 $m=2^n$,$*$ 代表 ALL 值。由于基表 R 中仅有一条元组,则我们可以得到 $V_1 = V_2 = \cdots\cdots = V_m = aggr(r)$。因此在这个数据立方中,我们实际上只需保存一条元组$(a_1, a_2, \cdots, a_n, V)$,$V = aggr(r)$,再保存一些附加信息用以说明这条元组代表一组立方元组。针对该数据立方的任何查询,我们都可以直接返回聚集值,而无需做进一步的聚集计算。

浓缩数据立方便是将由同一条基表元组计算聚集得到的立方元组浓缩成一条元组,以缩小数据立方的尺寸。文[6]将 BU-BST 基本单元组的浓缩[2]和小方内的前缀冗余消除结合,进一步提出了前缀立方 PrefixCube。

在本文中,我们提出了前缀立方的一种索引机制。实验表明它能取得较好的查询效果。

接下来,本文第 2 节介绍了前缀立方的基本结构;第 3 节详细描述了称为 Prefix-CuboidTree 的索引机制的基本思想,以及在这种索引机制下对应的查询算法;第 4 节利用模拟数据集和真实数据集进行具体实验,并对实验结果进行了分析,最后对全文进行总结。

2　前缀立方

前缀立方[6]是在基本单元组浓缩的基础上,消除数据小方内的前缀冗余得到的。一条基本单元组的形式定义如下:

定义 2.1(基本单元组,BST)　给定一组维属性 $SD=\{D_1, D_2, \cdots, D_n\}$,将基表在 SD 上进行划分,若在某一个分组中仅包含一条元组 r,则我们称 r 是 SD 上的基本单元组,SD 称为 r 的单值维集(the single dimension set)。

一个 3 维的基表 R,包含三个维属性 A,B,C 和一个度量属性 M,假设 R 中仅含一条元组 $r(a_1, a_2, a_3, m)$,由 R 计算得到的普通数据立方中含有 8 条立方元组(a_1, a_2, a_3, m),

*本文研究得到国家自然科学基金(项目编号 60303030)的资助。丁胡临　硕士研究生,主要研究方向是数据仓库与 OLAP。冯剑琳　博士,主要研究方向是数据仓库和数据挖掘。聂　晶　硕士研究生,主要研究方向是数据仓库和数据挖掘。

$(a_1, *, *, m)$,$(*, a_3, m)$,…,$(*, *, *, m)$,所有元组的聚集值相同,并且都源自 r。在任何一个维集上对 R 作划分,均得到一个仅包含元组 r 的分组。譬如,在维集$\{A\}$上作划分后,得到一个仅含元组 r 的分组,我们称 r 是维集$\{A\}$上的基本单元组,$\{A\}$是 r 的一个单值维集。最小浓缩数据立方[2](miniCube)用基本单元组 r 表示所有这些立方元组,并附记必要的单值维集信息。

文[2]提出了 BU-BST 算法来计算浓缩数据立方,并称其为 BU-BST 浓缩数据立方。一个 BU-BST 浓缩数据立方具有如下重要特性:

1)一条基本单元组与且仅与一个单值维集相关。

2)一条基本单元组能且仅能浓缩它的祖先元组,各条祖先元组所属的对应祖先数据小方的聚集维集都必定以该基本单元组的唯一单值维集为前缀,且所有祖先元组均和这条基本单元组共享了在单值维集上的维值,因此,又可以说这些立方元组存在共享的前缀维集值,基本单元组的浓缩实质上也是一种特殊的前缀共享。

3)任何一条数据立方元组要么被唯一一条基本单元组浓缩,要么不被任何基本单元组浓缩,也就是说它是一条普通立方元组。

一个 BU-BST 浓缩数据立方被分成两个互不相交的子立方:一个由所有普通立方元组组成,称为普通子立方,另一个由所有基本单元组组成,称为单元组子立方。普通子立方中的元组按聚集维集聚簇,称为普通数据小方(normal cuboid),单元组子立方中的基本单元组按单值维集聚簇,称为虚数据小方(virtual cuboid)。将每个普通数据小方或虚数据小方内的元组再进行前缀共享就得到了前缀立方。

对于前缀立方,无论是普通数据小方还是虚数据小方,它们的物理存储方式是统一的。一个数据小方里的元组按维值从小到大的顺序存储到若干固定大小的磁盘页面内,一个页面被称为一个数据块(data block)。数据块内的元组因前缀共享而形成若干个分组(group)。同一个分组中的所有元组都具有相同的第一维值,我们称为分组的公共维值。这样,在同一个数据块内,不同分组按公共维值从小到大的顺序分布;同一个分组内的元组按维值升序排列,并依次共享前缀。逻辑上,属于同一个数据小方的所有元组长度相同,即各元组的维值数目和聚集值数目对应相同,普通数据小方元组的维数等于聚集维属性的数目,虚数据小方元组的维数等于对应的虚数据小方单值维集中维的个数加上单值维集的最后一个维之后的所有维的个数。但实际存储一条元组时,该元组的某个维值因共享前一条元组对应维的值而无需存储,从而导致实际存储的元组的长度各不相同。所以,我们把元组结构进行扩展,在原始元组前面加上一个数值,用来表示这条元组的实际长度。

为了在一个数据块内尽快搜索到目标元组,数据块内部建立了分组索引。分组索引记录了指向每一个分组的指针,即分组在数据块内的相对位置。这样,一个数据块被分成了三部分:数据部分,分组索引部分,块控制信息部分。下面我们举例进行说明。表 1 是一个 3 维的普通数据小方 *Cuboid*(A,B,C),包含三个聚集维属性 A,B,C 和一个度量属性 M。它被存入一个数据块中,如图 1 所示。很明显,块内含有两个分组,索引 p1 指向第一个分组,索引 p2 指向第二个分组。每个分组的第一条元组都是长度为 5 的完整的元组,即没有共享其余元组的维值。如,数据块的第一条元组$(5,a1,b1,c1,m1)$,其中 5 是元组的长度,等于 1 个长度值加上 3 个维值加上 1 个度量值。

表 1 普通数据小方(A, B, C)

	A	B	C	M
t1	a1	b1	c1	m1
t2	a2	b2	c2	m2
t3	a3	b3	c3	m3

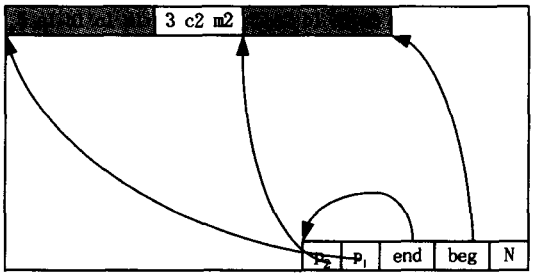

图 1 表 1 对应的数据块

3 前缀立方的索引 Prefix-CuboidTree

3.1 CuboidTree

范围查询是对数据立方的每一个维 D 给定一个取值区间$[D_l, D_h]$,其中 D_l 是维取值的下限,D_h 是上限。可以观察到维取值区间总可归结为以下两种模式:

1)D_l 和 D_h 都等于 0(0 代表 ALL 值)。这表明维 D 是非聚集维;2)D_l 和 D_h 都大于 0。这表明维 D 是聚集维。

对于区间$[0, h]$,$h>0$,可以分解为$[0, 0]$,$[1, h]$。通过具有上述模式的区间确定对应的维是否出现在某个 GROUP BY 的分组属性组合之中,从而决定对哪个数据小方进行范围查询。

基于上述观察,文[7]指出对数据立方的范围查询事实上都可归结为对若干个特定数据小方的范围查询,由此,文[7]提出了一种新的数据立方的索引机制 CuboidTree,即对每一个数据小方都建立 zkd B-Tree[8] 索引,然后对各个根结点进行索引。其中,zkd B-Tree 是指在物理上使同一个数据小方内的元组按 z-order 码[9] 的顺序存储,然后以 z-order 码为索引关键字对每一个数据小方建立 B-Tree 索引。文[7]采用 CuboidTree 对 BU-BST 浓缩数据立方进行索引,取得了较好的查询效果。

3.2 Prefix-CuboidTree

回顾第 2 节关于前缀立方的描述,我们可以归纳出它具有以下结构特性:1)元组按数据小方进行聚簇;2)数据小方内的所有元组按维值大小升序排列;3)物理上,数据小方存储成若干个数据块,数据块内的元组按分组聚簇,各分组按公共维值升序排列。分组内的元组依次共享前缀。

根据前缀立方的第一个结构特点和范围查询可以分解为对各数据小方进行查询的特性,我们认为 CuboidTree 的基本结构,即对每一个数据小方建立 B-Tree 索引然后对各 B-Tree 根结点建立索引,依然适用于前缀立方。

这样,问题的关键就在于:怎样对具有特点 2)和特点 3)的单个数据小方建立 B-Tree 索引。问题可进一步集中到两个方面:

1)*索引粒度的选择* 即怎样选择最小的索引单位,是一条元组还是一个分组。也就是说,B-Tree 索引部分最底层结点内的索引项里存放的是一条元组还是一个分组的物理位置,或者有另外的选择。

2)*索引关键字的选择* 搜索 B-Tree 过程中,所费时间主要消耗在将树结点读入内存这个方面。而每一次搜索实际上是把 B-Tree 从根到某个叶子的所有结点依次读入内存再查

找的过程。因此，搜索时间与树高度成正比。为了减少搜索时间，最直接的想法就是使 B-Tree 的高度尽可能小，从而减少一次搜索过程中读入内存的结点个数。为避免索引项过多使树的高度随着小方内元组数目的增多而快速增加，我们并没有将索引粒度固定在一条元组或一个分组。我们以数据块为最小索引单位，即 B-Tree 索引部分最底层结点内的索引项里存放的是一个数据块的物理位置。

虽然 z-order 码能使同一个数据小方内元组的物理邻近性接近于逻辑上的多维空间邻近性[9]，从而达到较好的查询效果。但是鉴于前一段的分析和开始提到的前缀立方的结构特点 2)和特点 3)，我们没有采用 z-order 码作为关键字，而是以数据块内第一个分组里的第一条元组代表整个数据块，将该元组各聚集属性值顺序组合成一个数值串作为关键字。这条元组实质上也是该数据块的第一条元组，我们称其为数据块的代表元组。如图 1 所示，普通数据小方(A,B,C)对应的数据块的代表元组是($a1,b1,c1,m1$)，元组中我们略去了索引长度值，仅取维值和聚集值；索引关键字是“$a1b1c1$”。虚数据小方的数据块代表元组与普通数据小方一样中，是数据块的第一条元组，但我们仅取代表元组中对应到单值维集中各维的维值，将它们顺序组合在一起生成索引关键字。譬如在数据立方(A,B,C,D)中，假定虚数据小方(A,B)对应的一个数据块的代表元组是($a3,b3,c3,d3,m3$)，那么该数据块的索引关键字是“$a3b3$”。

这样，我们便得到了 PrefixCube 的一种索引机制—Prefix-CuboidTree：

1)其基本结构与 CuboidTree 类似，是对每一个数据小方(包括虚小方)建立 B-Tree 索引，再对 B-Tree 的根结点建立索引。

2)B-Tree 索引部分的结点由索引项构成。每一个索引项是这样的序对：〈关键字，被索引页面在磁盘上的物理位置〉。B-Tree 最底层结点直接对数据块进行索引。

3)采用数据块内第一个分组里第一条元组的聚集维值或单值维值的顺序组合作为索引关键字。

4)数据块内建立了分组索引。

3.3 Prefix-CuboidTree 的范围查询算法

从上一节的分析可知，建立了 Prefix-CuboidTree 的前缀立方可看作是由若干个带有 B-Tree 索引的数据小方组成。结合范围查询的特性，对前缀立方的查询实质上可归结为对若干个带有子 B-Tree 索引的数据小方的查询。而且，对前缀立方里每一个数据小方作范围查询处理时，与 BU-BST 浓缩数据立方一样，要分为两部分，一部分是对普通数据小方的查询，另一部分是对聚集维集的所有前缀维集对应的虚数据小方的查询。因为虚数据小方与普通数据小方具有相同的存储模式和子索引结构，所以我们在查询时，对它们采用了统一的算法，不再分开处理。查询算法详见算法 1 和算法 2。

算法 1 描述了通过 B-Tree 索引在数据小方中搜索所有满足条件的元组的过程。它的输入是两条元组 $q1$ 和 qh，分别代表查询上界和查询下界，$q1$ 中每一个维值为该维上查询范围的下界值，qh 中每一个维值为该维上查询范围的上界值。该算法的思路是这样的，它把一个数据小方看作一个多维的查询空间，把每一条满足条件的元组(包括查询上界和查询下界)看作查询空间上的一个点，并以 $q1$ 为初始搜索点(行 1)，在索引中搜索可能包含该搜索点的数据块(行 3)。如果得到了某个数据块，算法紧接着找出其中所有满足条件的元组(行 4－5)和元组关键字的上界(行 6)。如果找不到这样的数据块，算法把 *upper* 直接设置为整个索引的最小关键字(行 7－8)，即根结点第一个索引项包含的关键字。这样，根据关键字上界 *upper* 和查询上下界就可以确立下一个搜索点了(行 10)，当然，搜索点对应的关键字要大于等于 *upper* 并且处于查询范围以内。不断重复以上过程，当找不到处于查询范围内的搜索点时，算法就结束搜索并输出所有满足条件的元组(行 11－13)。

算法 2 描述了获取下一个搜索点的过程。它的输入是查询上下界对应的索引关键字和当前被搜索页中元组关键字的上界 *upper*。算法从组成 *upper* 的第一个维开始搜索(行 2)，如果当前维维值小于下界对应维值(行 3)，那么将该维及其后所有维值设成下界对应维值(行 4)，结束搜索并返回这个新的搜索关键字(行 5)；如果当前维值大于上界的对应维值，算法将从前一个维开始往前搜索，直到找到一个维，它的维值小于上界对应维值(行 8)。如果满足这样的条件的维不存在，那么结束搜索，返回空值，表明已经不存在新的搜索点了(行 9)；如果存在，那么我们使得该维维值增 1(行 10)，然后将其后所有维维值设为下界对应维的值(行 11)，结束搜索并输出新的搜索关键字值(行 12)。如果 *upper* 的所有维值都在上下界范围内，我们简单的返回 *upper* 就可以了(行 15)。

在一个数据块中，算法利用数据块内的索引实现块内分组跳跃。如在图 1 所示的数据块里搜索关键字为“$a2b1c1$”的元组。我们首先用关键字的第一个维值 $a2$ 与第一个分组的公共维值 $a1$ 相比较，$a2$ 大于 $a1$，我们直接跳过第一个分组，并找到第二个分组的索引 $p2$，得到第二个分组的位置，比较关键字的第一个维值和公共维值。如果维值相等，则接着比较其余维的值。第二个分组中第一条元组与被搜索元组各维值相等，表明元组已经找到，停止搜索。

算法 1 Search_Prefix-SubTree(Tuple $q1$, Tuple qh)

```
1://output tuples in query box
  start = KEY(q1); end = KEY(qh); cur = start;
2:while 1 do
3:     dataPage = getPage(cur); //find data page
4:     if dataPage ! = NULL then
5:      FilterTuples(dataPage, ql, qh); //get tuples in query box
6:      upper = getUpperKey(dataPage);//get upper bound key
7:     else
8:      upper = getLeastKey(); //get the least key of this index
9:     endif
10:     cur = getNextkeyValue(upper, start, end);
11:     if cur = = NULL then
12:          return all tuples in query box;
13:     end if
14:end while
```

算法 2 getNextKeyValue(keyValue *upper*, keyValue *start*, keyValue *end*)

```
1://ourput new search key value or null
  dimsNum = getDimsNum(); //get group dimensions number
2:for int number =0; number<dimsNum; number++ do
3:    if upper[number]< start[number]then
4:      upper[number : dimNum-1]= start[number: dimNum-
1];
5:      return upper;
6:    end if
7:    if upper[number]> end[number] then
8:       search number dim from number-1 to 0 in upper and such
         dim value is less than the corresponding dimension value in
         end;
9:       if dim dose not exist then output NULL; end if
10:        upper[dim]++;
11:        upper[dim+1 : dimNum-1]= start[dim+1: dimNum
-1];
12:        return upper;
13:    end if
14:end for
15:return upper;
```

4 实验分析

我们采用 CuboidTree 对 BU-BST 浓缩数据立方和普通数据立方进行索引，采用 Prefix-CuboidTree 对前缀立方进行

索引，并对它们进行了查询比较实验。实验采用的数据集为气象数据集[10]，模拟生成的均匀分布数据(Uniform)和自相似(Self-Similar)数据。气象数据集基表总共包含 1,015,367 条元组，每条元组都有 9 个维，各个维的基数依次为：station-id(7, 037)，longitude(352)，solar-altitude(179)，latitude(152)，present-weather(101)，day(30)，weather-change-code(10)，hour(8)，brightness(2)。自相似数据基表和均匀分布数据基表都有 200,000 条元组，每条元组 9 个维，各个维的基数分别为 2000,1000,500,200,100,80,60,50,30。实验平台为：PetinumIII 800MHZ，256MB 内存，30GB IDE 硬盘，Windows XP 操作系统。

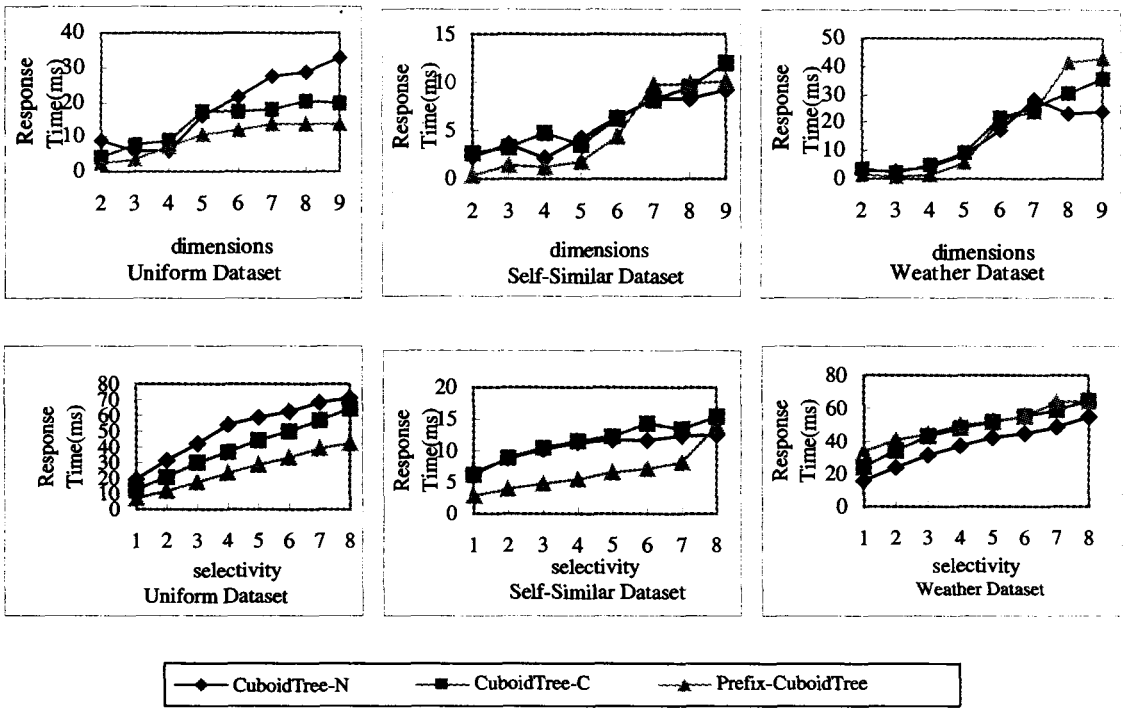

图 2　查询性能

4.1　实验设计

我们考虑到两个可能影响范围查询性能的因素：一个是数据立方里维的数目，我们称之为维度(dimensions)；一个是范围查询的长度，即对应维值的上界与下界之间的差值。我们把某个维上范围查询的长度与该维的基数的比值称作选择度(selectivity)。为此，我们设计了两类实验：

1)固定所有维的选择度为 1/4，对 2—9 维的数据立方分别进行范围查询。对每一个数据小方，我们随机生成 10 个范围查询。

2)固定维度为 9，选择度从 1/8－1 变化(每次递增 1/8)对每一数据立方分别进行 8 次范围查询。一次查询中将随机生成对每一个数据小方的 10 个范围查询。

我们目前设计的对每一个数据小方的范围查询都是前缀范围查询。即我们按顺序限定了各聚集维的取值范围，并且只要求取得聚集值。

4.2　实验分析

实验首先比较了在维数发生变化的情况下被 CuoidTree 索引的 BU-BST 浓缩数据立方(CuboidTree-C)和普通数据立方(CuboidTree-N)以及被 Prefix-CuboidTree 索引的前缀立方在查询性能上所受到的影响，见图 2 上半部分。实验结果表明，在维度较小的情况下的前缀立方查询性能比较好，但是当维度增大时，性能并不是很好。这是因为被 Prefix-CuboidTree 索引的分组里各元组具有相同的第一维的值，而且在此基础上元组依次共享其余各维。这样，我们搜索一个数据块时，如果被搜索的元组在第一个维上就与某分组的公共维值不同，我们便跳过整个分组所包含的元组，重新搜索下一个分组。对一个数据小方而言，在数据立方维度较小的情况下，分组内元组间共享的维数目所占的比重就越大，这样在前缀立方的一个数据页块内查询从一个分组到另一个分组的跳跃就越频繁，所以能取得较好的查询效果。维度增大时，按维序排列的元组的物理存储邻近性与多维空间邻近性的差异增大，分组内被共享的维数在聚集维中所占的比例减小。而此时，z-order 码能较好地保持多维空间中点的邻近性，所以在 CuboidTree 机制下的浓缩数据立方和普通数据立方都能取得比前缀立方更好的查询效果。我们同时发现，相同的索引机制下，维度增大时普通数据立方范围查询的性能反而要比 BU-BST 浓缩数据立方好。这是因为对浓缩数据立方里一个数据小方的查询分为两部分，对普通数据小方的查询和对其聚集维集中所有前缀维集对应的虚拟数据小方的查询。很明显，随着维度的增大，所需要查找的虚小方越来越多，这样使得浓缩数据立方的查询性能下降。从这个因素来说，前缀立方和 BU-BST 浓缩数据立方的变化趋势是一致的。均匀分布数据集上随维度增高查询性能下降的情况并不明显，这是因为基表数据集比较稀疏，在前几个维上产生的 BST 比较多，维度增大时，虚小方的数目并没有大量增加，同时，由于 BST 能够代表多个立方元组，因此缓冲一个虚小方块，可以供多个查询使用，这样一来在均匀分布数据集上前缀立方和浓缩数据立方的查询性能并没有随着维度增大而急剧恶化。

然后我们比较了在选择度改变的情况下三种数据立方所受到的影响，见图 2 下半部分。显然，在均匀分布和自相似数据集上，前缀立方的性能好于 BU-BST 浓缩数据立方和普通数据立方，因为这两个数据集相对比较稀疏，所以不论是前缀

共享还是 BST 浓缩都能达到较好的效果。因此查询性能更好。但是在气象数据集上面，前缀立方的性能却是最差的，这是因为在真实数据集上数据比较稠密，选择度越低，按 z-order 码进行多维聚簇的效果越有利于范围查询。选择度较高时，如选择度为 1 时，要查询整个数据小方，这时，如果同一个数据小方的元组按维序被聚簇到一起，自然有利于查询，所以这时候前缀立方查询性能变得好一些。

结论 本文基于 CuboidTree 索引，提出了前缀立方的一种索引机制 Prefix-CuboidTree。总体来说，采用这种索引机制对前缀立方进行查询能取得较好的查询效果。实验表明，在气象数据集和自相似数据集上，其查询性能随着数据立方的维度增大而下降，随着选择度增加而上升。这主要是因为元组按维序进行聚簇会导致在维度增大时其物理存储邻近性和多维空间的点邻近性的差异增大。选择度增加，能平衡这方面的差异，从而使范围查询性能得到一定的改善。

参 考 文 献

1 Gray J，Bosworth A，Layman A，et al. Data Cube：A Relational Operator Generalizing Group-By，Cross-Tab，and Sub-Totals. In：Proc. of the Int. Conf. on Data Engineering，1996

2 Wang W，Feng J，Lu H，Yu J X. Condensed Cube：An Effective Approach to Reducing Data Cube Size. In：Proc. of the Int. Conf. on Data Engineering，2002

3 Sismanis Y，Deligiannakis A，Roussopoulos N，Kotidis Y. Dwarf：Shrinking the PetaCube. In：Proc. of the ACM SIGMOD Int. Conf. on Management of Data，2002

4 Lakshmanan L V S，Pei J，Han J. Quotient Cube：How to Summarize the Semantics of a Data Cube. In：Proc. of Int. Conf. on Very Large Data Bases，202

5 Lakshmanan L V S，Pei J，Zhao Y. QC-Trees：An Efficient Summary Structure for Semantic OLAP. In：Proc. of the ACM SIGMOD Int. Conf. on Management of Data，2003

6 Feng J，Fang Q，Ding H. PrefixCube：Prefix-sharing Condensed Data Cube. To appear：ACM International Workshop on Data Warehousing and OLAP，2004

7 Feng J，Si H，Feng Y. Indexing and Incremental Updating Condensed Data Cube. In：Proc of the Int. Conf. on Scientific and Statistical Database Management，2003

8 Orenstein J A，Merret T H. A Class of Data Structures for Associate Searching. In：Proc. of ACM SIGMOD-PODS Conf.，Portland，Oregon，1984. 294～305

9 Oracle INC. Oracle 9i Spatial，An Oracle Technical White Paper. 2001

10 Hahn C，Warren S，London J. Edited. Edited synoptic cloud report from ships and land stations over the globe，1982～1991

（上接第 43 页）

图 5 比较了两者的链路层时延。在引入了本文提出的拥塞控制后，缓解了链路层的压力，链路层时延变得很小，即使扩大成 10 倍，也远小于没有拥塞控制的情况。

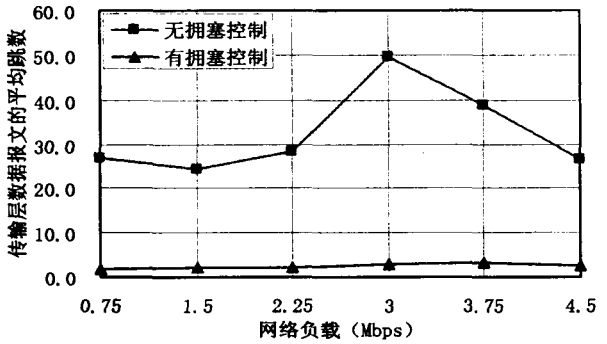

图 6 网络负载变化与传输层数据报文平均跳数的关系

图 6 对比了两者传输层报文的平均跳数，即传输层总的发送与转发次数÷成功发送的报文数目。两者相差相当悬殊，这说明加入了本文提出的拥塞控制机制后，传输失败的次数大大地减小了。

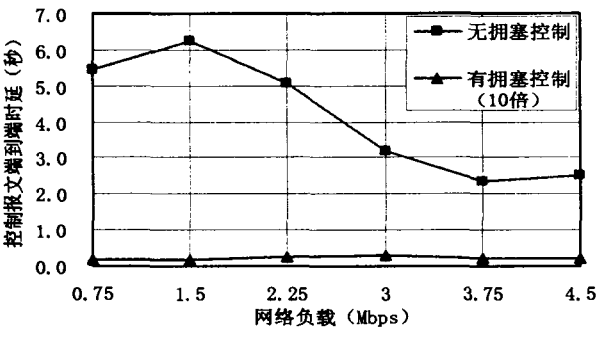

图 7 网络负载与控制报文端到端时延的关系

图 7 比较了两者路由层控制报文的平均时延。通过本文提出的拥塞控制算法，路由协议的运行不会受到过大的网络负载的影响。这里显示出了一个有趣的现象，在没有使用拥塞控制算法的情况下，控制报文的平均链路层时延居然大于总的平均链路层时延（如图 5），也就是说它大于数据报文的平均链路层时延。作者分析，这是由于盲目发送而引起的——有许多数据报文因为其发送/转发节点附近的链路层比较空闲而被发送了，这些数据报文的时延会比较短，加之数据报文的数量远大于控制报文，所以造成了这种现象。

结论 仿真结果验证了该算法的具有很好的效能——不但大大地降低了链路层时延，也显著地提高了传输层的吞吐率；同时也印证了前文对 ad hoc 网络的分析。制约 ad hoc 网络吞吐性能的因素主要有这两点：盲目发送问题以及同一报文的多个副本在网络中同时出现，这两个因素都无谓地加重了链路层的负担，特别是盲目发送问题。从结果图中可以看出，链路层的巨大网络负载带来了巨大的吞吐量（发射机工作时间长，电池能量消耗多），这加重了报文竞争信道时碰撞的可能性，造成发送失败率高，加大了发送时延，也就增加了链路层的时延，同时导致传输层的低吞吐率。而第二个问题，则是由链路层时延过大，又没有必要的控制措施而引起的。总之，要想解决这些问题，需要把握 ad hoc 网络的特点，实现链路层和传输层的通力配合，而传统的有线网络上的拥塞控制不能适应 ad hoc 网络的特点，这也正是本文提出的拥塞控制算法的优点所在。

参 考 文 献

1 IEEE 802.11 Working Group. http://grouper.ieee.org/groups/802/11/index.html

2 AODV Routing Protocol，rfc3561.txt. http://www.ietf.org/rfc/rfc3561.txt

3 Internetworking with TCP/IP Volume I：Principles，Protocols，and Architecture，3rd Edition. Douglas E. Comer，Department of Computer Sciences，Prudue University，West Lafayette

4 Datagram Congestion Control Protocol（DCCP）. http://www.ietf.org/internet-drafts/draft-ietf-dccp-spec-06.txt