北京大学学报(自然科学版),第 43 卷,第 1 期,2007 年 1 月
Acta Scientiarum Naturalium Universitatis Pekinensis, Vol. 43, No. 1 (Jan. 2007)

# 基于覆盖码预排序的数据立方体部分和 top-k 查询算法*

张卫华[1)]　谭少华　殷普杰　张煜宇　成　富
(北京大学信息科学技术学院,视觉与听觉信息处理国家重点实验室,北京,100871; 1)通讯作者,E-mail:cogent_zwh@126.com)

**摘　要**　在决策支持系统中,排序查询是研究的热点问题。提出了一种在 OLAP(数据仓库)数据立方体中对部分和查询结果进行排序的高效算法,该算法综合利用覆盖码和预排序,有效地解决了对部分和结果的 top-k 查询问题。实验结果表明无论数据在随机分布还是存在主导集情况下,该算法都能很好地改进查询的时间代价。
**关键词**　联机分析处理; 部分和; top-k; 覆盖码
**中图分类号**　TP 311.13

# An Efficient Algorithm Based on Covering Codes about Ranking the Results of Partial-Sum Queries

*ZHANG Weihua*[1)]　*TAN Shaohua*　*YIN Pujie*　*ZHANG Yuyu*　*CHENG Fu*
*(National Laboratory on Machine Perception, School of Electronis Engineering and Computer Science, Peking University, Beijing, 100871;*
*1) Corresponding Author, E-mail: cogent_zwh@126.com)*

**Abstract**　In decision-support systems, ranking-aware queries have been gaining much attention recently. An efficient algorithm is introduced, which, by incorporating covering codes and pre-sorting, can effectively resolve the top-k queries over partial-sum results. This algorithm is empirically evaluated and the experimental results show that the query cost is improved significantly.
**Key words**　OLAP; partial-sum; top-k; covering codes

## 0　引　言

在信息发现的过程中,对排序后的多维数据集进行 top-k 查询是一个基本的步骤。在 OLAP 中,部分和查询和 top-k 查询都是应用非常广泛的操作。现在,越来越多的决策分析应用需要将两者结合起来使用。考虑下面这个例子,一个关于社区居民就业状况的数据立方体,假设这个数据立方体包括 3 个功能属性(维度):时间、职业状况、所在社区。再假设时间维度的值域是 1995 年第一季度到 2004 年第四季度,职业状况维度的值域是{学生、公务员、公司职员、待业、下岗、退休、内退、无业、失业},所在社区维度的值域是深圳市南山区 96 个社区,那么这个数据立方体就有 40 ×9 ×96 个单元格,每一个单元格内包含相对应的人数(度量属性),例如(28, 1995Q1,下岗,大冲)。在分析的时候经常需要查询各个社区在某个时间段内,下岗、内退、失业及无业人员的人数之和,这样一类查询被称之为部分和查询。为了更好地分析各社区失业状况严重程度,需要根据上述部分和查询的结果,对各社区按照部分和的值进行排序,找出失业人数最多的 5 个社区,重点给予关注。

基于数据立方体模型[1]的区域和查询研究非常多[2-5],他们对于本文的研究工作提供了很多可以借鉴的地方,但是区域和查询的算法不能直接用于解决不连续的部分和问题。Li 等人[6]对于连续和查询做出了卓越贡献,虽然连续和可以看作部分和的一种特殊情况,但是连续和毕竟不同于部分和,所以

* 国家自然科学基金(60473051 和 60473072)资助项目
收稿日期: 2005-12-30; 修回日期: 2006-04-28

这样的算法不能直接应用于部分和查询中。Ho 等人[7]提出了一种新算法—覆盖码,用于解决部分和查询的问题[8—10],并给出了空间和时间复杂度的平衡算法。不过他们的算法并没有涉及直接针对部分和查询结果作排序的预计算问题。就目前了解的情况来看,尚未看到有算法可以对于部分和查询的结果排序问题作出直接回答,本文第一次提出了解决该问题的算法。

本文接下来按照如下章节组织:第1部分通过一个简单的例子说明这个新的算法的应用范围以及朴素算法存在的问题;第2部分引入了覆盖码技术相关的概念和在部分和查询中的应用,并详细介绍了解决此问题的算法;在第3部分中给出实验结果用于证明算法的有效性;第4部分总结了全文并对未来的研究做出展望。

## 1 朴素算法存在的问题

为了便于描述问题,可以将引言中的例子简化为一维的部分和排序问题。如表1所示,现在有 $n$ 个社区($D1$, $D2$ ……$Dn$)的失业人口资料,人们想知道哪些是失业人数最多的前 $k$ 个社区,而失业人数包括下岗($L1$)、无业($L2$)、待业($L3$)和内退($L4$)等。由于统计口径不同,需要取不同的集合来计算失业人数,那么得到的社区排名自然不同。人们关心在不同情况下如何尽快找出前 $k$ 个社区。例如当统计口径定义为失业 = 下岗 + 待业 + 内退,人们关注的是计算符合 $L1$, $L3$, $L4$ 3种情况的失业人数之和最高的前 $k$ 个社区。

**表1** 社区居民就业状况数据立方体的切片

Table 1 Slice of data in the data cube of job status of community residents

| 分类 | 社区 | | | | |
|---|---|---|---|---|---|
| | $D1$ | $D2$ | $D3$ | $D4$ | $D5$ |
| $L1$ | 8 | 11 | 9 | 15 | 15 |
| $L2$ | 18 | 22 | 11 | 26 | 23 |
| $L3$ | 25 | 14 | 29 | 29 | 31 |
| $L4$ | 7 | 5 | 6 | 8 | 9 |
| $L5$ | 9 | 15 | 25 | 11 | 13 |

对于这样的一类查询问题,最朴素的算法就是逐个计算每个选定的数据项,即先计算各个社区 $L1$, $L3$, $L4$ 的数据的和($sum_1$, $sum_2$ ……$sum_n$),然后对这些和进行全排序,取出 top-k 个社区。朴素算法有2个显著缺点:(1) 当 $n$ 较大时,由于需要计算涉及的数据项的数量非常大,而数据立方体往往是存在外存中,所以这种方法的 I/O 次数和时间复杂度都相当大。(2) 尤其当 $>> k$ 时,尽管需要的只是前 $k$ 个结果,却需要计算 $n$ 个部分和,并且要对 $n$ 个部分和的结果进行排序,排序的过程也会有相当大的时间代价。这些都会导致查询反应效率的极大下降。为了有效地解决朴素算法的问题,需要从两个方面减少时间代价:一方面通过预计算减少部分和查询计算的响应时间,另一方面,采用预排序减少对部分和排序找出 top-k 的时间。

## 2 基于覆盖码和预排序的算法

### 2.1 覆盖码的基本概念

首先,为了提高部分和排序的效率,本文采用预计算部分和的方式来提高查询效率。理论上,可以预计算全部的部分和,例如对于一个具有5个元素的数组,可以计算得到 $2^5$ 个所有可能的部分和,从而使回答任何部分和查询的时间是一个常数。但是这样的预计算量太大,人们总是希望在取得良好的查询效率的同时预计算量不要太大,在预计算工作量和查询效率之间取得较好的平衡,本文利用覆盖码[1]来达到这种平衡。为了便于理解,首先引入覆盖码理论的一些基本概念,他们来源于纠错码理论[9,10]。

**定义1** 权重:$n$ 位二进制向量的权重为这个二进制向量中1的个数,如 $V$ = (00110) 的权重为2。

**定义2** 距离:2个 $n$ 位二进制向量的距离为这两个向量异或的结果的权重,如计算 $V_1$ = (00010) 和 $V_2$ = (00011) 的距离,$V_1 \oplus V_2$ = (00001),(00001) 的权重是1,所以 $V_1$ 和 $V_2$ 的距离为1。

**定义3** 覆盖半径:一个向量集合 $C$ 的覆盖半径 $r$,是任何同样长度向量到这个集合中任意一个向量的距离的最小值的集合中的最大值。设 $C$ 是给定向量集合(向量长度为 $n$),$\{0,1\}^n$ 是所有长度为 $n$ 的向量集合,$d(x,u)$ 表示任何长度为 $n$ 的向量 $x$ 到集合 $C$ 中的向量 $u$ 的距离,那么 $C$ 的覆盖半径可以表示为:

$$r(C) = \max\{\min\{d(x,u) \mid u \in C\} \mid x \in \{0,1\}^n\}。$$

**定义4** 覆盖码:一个二进制向量集合 $C$ 称为 $(m, K, R)$ 覆盖码,是当存在以下情况时:(1) 所有 $C$ 里的向量长度都是 $m$;(2) $C$ 里一共有 $K$ 个向量;(3) 集合 $C$ 的覆盖半径是 $R$。

具体的例子,$C$ = {(00000),(00111),(10000),

(01000) ,(11011) ,(11101) ,(11110) }是一个(5 ,7 ,1) 覆盖码。

本文用一个二进制向量来表示部分和,例如,用(00110)表示第 3 位和第 4 位的和,这样利用(5 ,7 ,1)覆盖码,预先进行部分计算覆盖码对应的部分和,可以做到对于其他任何的部分和,都只需要最多一步加法或者一步减法就能得到需要的数据。

对于上面的例子,$L1 + L3 + L4$ 的对应的二进制向量是(10110) ,也就是(11110) - (01000) ,由于已经预先算好了(11110)对应的部分和 $sum = L1 + L2 + L3 + L4$,只需要进行一次减法运算 $sum - L2$ 就可以得到 $L1 + L3 + L4$ 的和。

为了减少存储空间,对于规模较大的数组,可以先将数组分块,然后对每一块利用覆盖码预计算,最后把每一组的和再相加,就得到了结果。比如一个长度为 9 的数组,如果直接应用覆盖码(9 ,62 ,1) 需要预计算并存储 62 个值,而如果分为长度分别为 5 和 4 的两块,在每一块上应用(5 ,7 ,1) 覆盖码进行计算,只需要预计算存储 14 个值。

## 2.2　基于覆盖码和预排序的算法

对于分组后的其中一组来说,当采用距离为 1 的覆盖码时,部分和可以是一个覆盖码对应的值,也可以是一个覆盖码对应的值与一个原始值的和(或者差)。可以预先对所有的覆盖码计算结果和原始数据都进行排序,对原始数据保留一个双向链表,以便取得前 $k$ 个或者最后 $k$ 个数据,对覆盖码保留一个从高到低的链表,以便取得前 $k$ 个数据。对应于表 1 的排序覆盖码数据立方体如表 2 所示。对于需要计算的部分和,因为覆盖码和所有的部分和距离不超过 1,所以只有 3 种情况:

(a) 需要计算的部分和就是覆盖码中的一个或者是单独的某一行数据。这个时候不需要进行计算,因为原来已经预先排好序了,只需要直接取出前 $k$ 个就可以。

(b) 需要计算的部分和是覆盖码中的某一个 $Cn$ 加上一个原始数据 $Lm$。$Cn$ 和 $Lm$ 都已经排好序,$Cn$ 从前往后取数据到 $Set1$ 中,$Lm$ 也从前往后取数据到 $Set2$ 中,直到 $Set1 \cap Set2$ 不小于 $k$,取 $Set = Set1 \cup Set2$,计算 $Set$ 里的社区的部分和,取前 $k$ 个就是所需要的答案。

(c) 需要计算的部分和是覆盖码中的某一个 $Cn$ 减去一个原始数据 $Lm$。$Cn$ 和 $Lm$ 都已经排好序,$Cn$ 从前往后取数据到 $Set1$ 中,$Lm$ 从后往前取数据到 $Set2$ 中,直到 $Set1 \cap Set2$ 不小于 $k$,取 $Set = Set1 \cup Set2$,计算 $Set$ 里的社区的部分和,取前 $k$ 个就是所需要的答案。

在上面的例子中,要计算 $L1 + L3 + L4$ 的时候,它对应的覆盖码是(10110) ,它可以转化为覆盖码(11110)对应的值减去 $L2$ 对应的值。因此,首先要取链表 $C4$ 的前面 $i$ 个社区集合 $Set1$ 和 $L2$ 数据最小的社区集合 $Set2$,当 $Set1 \cap Set2$ 的元素个数不小于 $k$ 时,取 $Set = Set1 \cup Set2$,计算 $Set$ 中各个社区对应的部分和,然后将 $Set$ 中各个社区对应的部分和进行排序,取最大的前 $k$ 个。利用这种算法,一方面减少了求和的计算量,只需要计算 $|Set|$ 个部分和,而不是全部 $n$ 个部分和,另一方面,减少了排序计算的工作量,这样只需要对 $|Set|$ 个部分和排序取前 $k$ 个,而不用对 $n$ 个部分和进行排序。利用反证法,可以很容易地证明 top-k 个社区必然在 $Set$ 中。证明从略。

**表 2　对应于表 1 的预排序覆盖码数据立方体**

Table 2　The sorted cube of the pre-sorting algorithm based on covering codes for the data cube shown in Table 1

| 分　类 | 社　区 | | | | |
|---|---|---|---|---|---|
| | $D1$ | $D2$ | $D3$ | $D4$ | $D5$ |
| $L1$ | $8^{D3}_{n}$ | $11^{D4}_{D3}$ | $9^{D2}_{D1}$ | $15^{D5}_{D2}$ | $15^{n}_{D4}$ |
| $L2$ | $18^{D2}_{D3}$ | $22^{D5}_{D1}$ | $11^{D1}_{n}$ | $26^{n}_{23}$ | $23^{D4}_{D2}$ |
| $L3$ | $25^{D3}_{D2}$ | $14^{D1}_{n}$ | $29^{D4}_{D1}$ | $29^{D5}_{D3}$ | $31^{n}_{D4}$ |
| $L4$ | $7^{D4}_{D3}$ | $5^{D3}_{n}$ | $6^{D1}_{D2}$ | $8^{D5}_{D1}$ | $9^{n}_{D4}$ |
| $L5$ | $9^{D4}_{n}$ | $15^{D3}_{D5}$ | $25^{n}_{D2}$ | $11^{D5}_{D1}$ | $13^{D2}_{D4}$ |
| $C1$(00111) | $41^{D4}_{D2}$ | $34^{D1}_{n}$ | $60^{n}_{D5}$ | $48^{D5}_{D1}$ | $53^{D3}_{D4}$ |
| $C2$(11011) | … | … | … | … | … |
| $C3$(11101) | … | … | … | … | … |
| $C4$(11110) | $58^{D4}_{D3}$ | $52^{D3}_{n}$ | $55^{D1}_{D2}$ | $78^{D5}_{D1}$ | $78^{n}_{D4}$ |

对于多个组的情况,每个不同的组 $i$ 会分为不参与计算、覆盖码或原始值、需要计算 3 种情况。对于不参与计算的组,就可以忽略它。对于覆盖码或者原始值的组,根据已经排好序的表格,可以逐次取得前 $k$ 大的集合 $Set_i$。对于需要计算的情况,可以逐次得到这一组的进行计算的两行的集合 $S1_i$、$S2_i$ 的交集 $S_{交i}$ 和并集 $S_{并i}$。这样可以逐次得到最终的交集 $S_{交}$ 和并集 $S_{并}$,交集 $S_{交}$ 是每一组的交集 $S_{交i}$(或者排好序的集合 $Set_i$)的交集,并集 $S_{并}$ 是每一组的并集 $S_{并i}$(或者排好序的集合 $Set_i$)的并集。类似地,当 $S_{交}$ 中的元素个数大于等于 $k$ 的时候,前 $k$

大的社区必然在并集$S_{并}$中。证明从略。

在具体计算中,为了提高效率,实验中采用了计算在并集中元素出现的次数的方法来判断某个元素是否在交集中出现。对于每一个组来说,如果被选择的行本身就是一个覆盖码或者原始值,那么进入交集需要出现的次数为1;如果被选择的行需要进行一次计算,那么进入交集需要出现的次数就是2。在分组情况中,进入最终交集需要某个元素出现的次数是应该在每组中出现的次数的和。

分组情况下的算法伪码如下:

```
Input:
AR(k,array,ccarray,query)        //top K, 原始数组, 覆盖码数组, 查询串
Procedure:
Set[] = Intersect(query)         //分解查询串,每组适合覆盖码长度的大小
setPointerCC = Φ                 //覆盖码的指针集合
setPointerV = Φ                  //原始值的指针集合
rowValid = 0                     //进入交集中的元素(社区)个数
For each set[i] of the set[] do
    property = Judge(set[i])     //判断这一部分的查询码是哪一种情况,得到相应的覆盖码和原始值
    If property.sign = "Value"   //原始值情况
        setPointerV.add(property.Vline.start)
        rowValid++
    else if property.sign = "CoveringCode"   //覆盖码情况
        setPointerCC.add(property.CCline.start)
        rowValid++
    else if property.sign = "add"   //覆盖码和原始值相加情况
        setPointerCC.add(property.CCline.start)
        setPointerV.add(property.Vline.start)
        rowValid = rowValid + 2
    else if property.sign = "minus"   //覆盖码和原始值相减情况
        setPointerCC.add(property.CCline.start)
        setPointerV.add(property.Vline.end,Reverse)
        rowValid = rowValid + 2
end for
countIntersect = 0;              //交集中元素数
countUnion = 0;                  //并集中元素数
setUnion = Φ                     //社区编号与和值对的并集
count[number] = {0}              //对每个社区的计数都设为零
while(countIntersect < k) do
    for every pointer[i] in setPointerCC
        count[setPointerCC.index]++      //社区计数增加1
        if(count[setPointerCC.index] = 1)    //刚刚进入并集
            setUnion.add({setPointerCC.index,sum(setPointerCC.index)})
        if( count[setPointerCC.index] = rowValid)    //刚刚进入交集
            countIntersect++
        pointer[i] = pointer[i].next
    end for
    for every pointer in setPointerV
        count[setPointerV.index]++       //社区计数增加1
        if(count[setPointerV.index] = 1)     //刚刚进入并集
            setUnion.add({setPointerV.index ,sum(setPointerV.index)})
        if( count[setPointerV.index] = rowValid)     //刚刚进入交集
```

```
            countIntersect + +
        pointer[i] = pointer[i]. next
    end for
end while
sort(setUnion)        //按照和值对并集进行排序
end procedure
return:
    top k of setUnion        //返回和值最大的 k 个社区
```

易知,对于有 $m$ 行 $n$ 列数据的情况,朴素算法对其中每一组进行相加求和,其算法复杂度是 $O(m)$。这样其求和部分的复杂度就是 $O(mn)$。在最后取出 top-k 个社区前需要对所有的 $n$ 个社区进行排序,排序部分的复杂度是 $O(n\log n)$。这样,朴素算法的复杂度就是 $O(mn + n\log n)$。

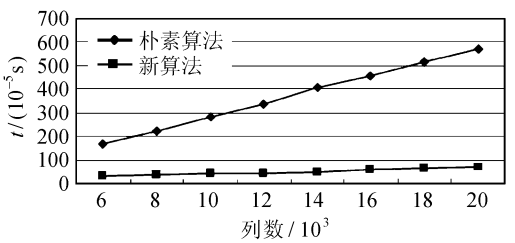
与朴素算法类似的是,本算法也分为求和部分和排序部分,但本算法在这两方面都对效率有大幅度的提升。同样地,对于有 $m$ 行 $n$ 列数据的情况,本算法使用长度为 $s$ 的覆盖码进行计算,也就是说每个组的大小固定为 $s$(最后一组可能小于 $s$),这样就把原有的数据分为了 $\lceil m/s\rceil$ 组,而最终需要计算和排序的社区数量就是算法结束时并集中的元素数量 $p$。本算法再求和部分的复杂度是 $O(\lceil m/s\rceil \cdot p)$,排序部分的复杂度是 $O(p\log p)$,算法的复杂度就是 $O(\lceil m/s\rceil \cdot p + p\log p)$。

本算法在两个环节可以显著减少计算时间。第一,对于每一组,应用覆盖码对部分和求和运算的算法复杂度从 $O(m)$ 降低为常数,即使分组后也可以线性地将复杂度降低为朴素算法的 $1/s$。第二,通过预排序,并通过集合运算减少参与排序的社区数量,从而减少了排序的时间。由于第二步集合运算的结果和数据分布有很强的相关性,而实际情况中普遍存在主导集,例如有些社区行政区划较大,人口众多,从而在失业人口、无业人口、下岗人口等方面数量都较大。经过对数据集中存在主导集的情况所做的大量研究实验证明,在绝大多数时刻,新的算法在主导集存在的情况下表现出更好的效果。同时,主导集所占比例对算法效率的影响也通过实验进行了研究。

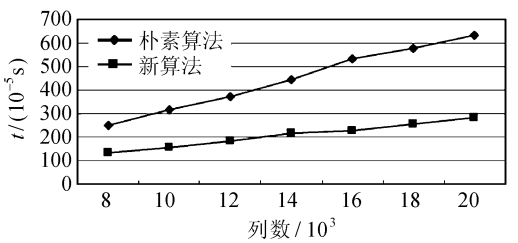
## 3　实　验

本文实验的数据集首先采用随机分布的数据集,然后又分别在随机分布数据集中插入生成了不同比例的主导集,分别得到了相应的实验结果。本实验的硬件环境: 戴尔台式机, CPU: Pentium4 3.00 GHz,系统内存: 512 M,硬盘为80 G、7 200 rpm; 软件环境: 操作系统为 Microsoft Windows XP Professional,数据立方体模型采用内存数据,编译环境: Microsoft VC 6.0。

(1) 数据集为自然随机分布时,通过以下实验结果图 1 (a)和(b)可以发现,随着社区数量和被选择行数的增长,朴素算法的用时基本上是线性增长的且增长速度很快,而覆盖码预排序算法显著优于朴素算法。




(a) 数据集行数为 5, $k$=15



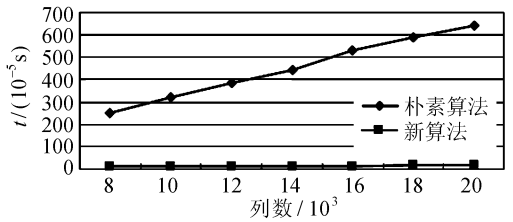

(b) 数据集行数为10, $k$=15

**图 1　随机分布数据集情况下社区数量变化对算法的影响效果**

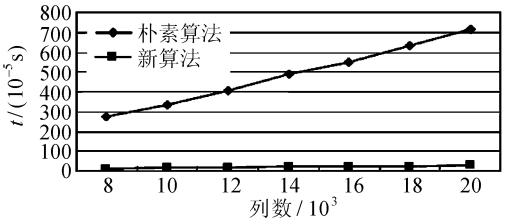
Fig. 1　Effect of the community dimension size over near-random datasets using uniform query set

(2) 数据集中存在主导集时,通过以下实验结果图 2 (a)和(b)可以发现,随着社区数量和被选择行数的增长,朴素算法的用时基本上是线性增长的且增长速度很快,而覆盖码预排序算法显著优于朴素算法。

(3) 最后,通过实验研究发现,数据集中主导集的比例和算法的效率相关。主导集占整个数据集的比例为 5 %～10 %时,该算法效率最高。而在其他情况下,新算法依然显著优于朴素算法。结果如图 3 所示。



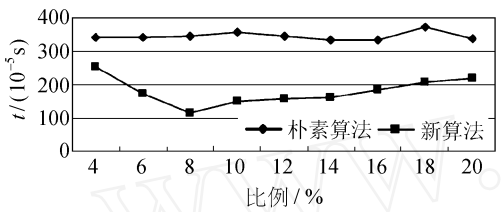

*k*=15,查询码:1000100111
(a) 数据集行数为10(2组)




*k*=15,查询码:10001001110010
(b) 数据集行数为15(3组)

**图2 存在主导集的数据集情况下社区数量变化对算法的影响效果**

Fig. 2 Effect of the community dimension size over dominant-set datasets using uniform query set




**数据集行数为**15(3**组**),**社区数量(数据集列数)为**10000,
**查询码**:001001000101011

**图3 数据集中主导集比例变化对算法的影响效果**

Fig. 3 Effect of the dominant-set proportion on the algorithm results

## 4 结 论

集成结果的排序问题是OLAP应用中非常普遍的,也是十分关键的操作。本文提出了一种基于覆盖码的对部分和结果进行排序的高效算法,实验结果证明,这种新的算法不但效率高,而且非常可靠,在几种不同数据分布情况下都表现出了很好的性能。在今后的研究中,作者将进一步探索物化存储与覆盖码相结合的办法,以进一步提高查询的效率。

## 参 考 文 献

[1] Loh Z X, Ling T W, Ang C H, et al. Analysis of pre-computed partition top method for range top-k queries in OLAP data cubes//CIKM'02, Mclean, Virginia, USA, 2002: 60-67.

[2] Li H G, Ling T W, Lee S Y, et al. Range-sum queries in Dynamic OLAP data cube//Proc 3rd International Symposium on Cooperative Database Systems for Advanced Applications, 2001: 74-81.

[3] Fagin R. Combining fuzzy information from multiple systems. In Proc Symp on Principles of Database Systems (PODS), 1996: 216-226.

[4] Ilyas I F, Shah R, Aref W G, et al. Rank-aware query optimization//Proc of SIGMOD, 2004: 203-214.

[5] Bruno N, Chaudhuri S, Gravano L. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. TODS, 2002, 27 (2): 153-187.

[6] Li H G, Yu H, Agrawal D, et al. Ranking aggregates. Technical Reports for 2004, [EB/OL]. [2004-03-11] http://www.cs.ucsb.edu/research/trcs/docs/2004-07.pdf.

[7] Ho C T, Bruck J, Agrawal R. Partial-sum queries in OLAP data cubes using covering codes. PODS 1997: 228-237.

[8] Gray J, Bosworth A, Layman A, et al. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals//Proc Int Conf on Data Eng (ICDE), 1996: 152-159.

[9] Graham R L, Sloane N J A. On the covering radius of codes//IEEE Trans Information Theory, May 1985, IT-31 (3): 385-401.

[10] Cohen G D, Honkala I, Litsyn S, et al. Covering codes. North-Hollans Math Lib, 1977, 54.

[11] Cohen G D, Litsyn S, Lobstein A C, et al. Covering radius 1985-1994. Journal of Applicable Algebra in Engineering, Communication and Computing, special issue, 1997, 8(3): 1-67.

[12] Agarwal S, Agrawal R, Deshpande P M, et al. On the computation of multidimensional aggregates//Proc of the 22nd Int'l Conference on Very Large Databases, Mumbai (Bombay), India, September 1996: 506-521.

[13] Harinarayan V, Rajaraman A, Ullman J D. Implementing data cubes efficiently//Proc of the ACM SIGMOD Conference on Management of Data, June 1996: 205-216.

[14] Gupta H, Harinarayan V, Rajaraman A, et al. Index selection for OLAP//In Proc of the 13th Int'l Conference on Data Engineering, Birmingham, U K, 1997: 208-219.

[15] Colby L S, Cole R L, Haslam E, et al. Red brick vista: Aggregate computation and management//Proc of the 14th Int'l Conference on Data Engineering, 1998: 174-177.