2007 年 11 月 第 11 期
Vol. 28 No. 11 2007

结构化网络中聚合 Top-K 查询优化技术

李 奈, 王 斌, 关 晶, 王国仁

(东北大学 信息科学与工程学院, 辽宁 沈阳 110004)
E-mail: emiliyar@163.com

摘 要: top-k 查询在分布式环境中引起越来越多的关注, 但是现存的一些 top-k 算法大都只适用于集中式网络. 提出了一个解决分布式网络中 top-k 查询的新方法—Histogram-Container 算法(简称为HC 算法), 它不仅网络延迟小, 网络带宽花费少, 而且能够运行在任何结构的分布式网络中. 本文将基于一个树型拓扑网络来说明如何使用本地的直方图和 bloom filter 信息来优化查询, 以及如何在中间节点进行部分结果的合并. 实验评估和性能分析表明HC 算法在网络带宽消耗和查询响应时间方面要优于其他同类方法

关 键 词: top-k 查询; 直方图; bloom filter; 分布式网络
中图分类号: TP311 **文献标识码**: A **文 章 编 号**: 1000-1220(2007)11-2033-05

Top-K Query for Aggregate Value in Structured Networks

LI Nai, WANG Bin, GUAN Jing, WANG Guo-ren

(*College of Information Science & Engineering, Northeastern University, Shenyang* 110004, *China*)

Abstract: Top-k query processing has received more and more attention, but existing top-k algorithms can be only applied in the centralized network. This paper presents a new algorithm to answer top-k query, called Histogram-Container (HC for short), which can achieve major performance gains in terms of query response time and network bandwidth, and furthermore, it can resolve queries in any kind of structured overlay networks. We will show how to use local histogram and bloom filter in a tree structure for optimizing the query and how to process intermediate results in inner peers. Our experimental results show that HC can achieve major performance gains in terms of network bandwidth, query response time.

Key words: top-k; histogram; bloom filter; distributed network

1 引 言

目前, 关于 top-k 查询的研究主要有两个方面: 一个是直接对属性值进行 top-k 查询; 另一个需要对属性值作聚合, 然后对聚合值进行 top-k 查询. 本文提出了一个关于聚合值的 top-k 查询处理技术: Histogram-Container(简称为HC).

例如, 某个公司想在最受欢迎的 50 个网站上为自己的产品作广告宣传, 这时他感兴趣的就是"整个网络中点击次数最高的前 50 个网站的网址." 如果一个网址在某个 peer 上点击次数非常高, 而在其他 peer 上点击次数为 0, 那么这个网址就不能被认为是最受欢迎的. 因此查询最受欢迎的网址应该查询在每个 peer 上都受欢迎的网址, 也就是查询在每个 peer 上点击次数之和最高的网址. 这里的聚合函数就是求和函数, 属性值为点击次数.

评价 top-k 算法的性能的标准是

1) 低网络延迟

2) 低带宽消耗

网络延迟是由top-k 算法的执行步骤决定的; 带宽消耗是由算法执行过程中, 网络中传输的数据量决定的. 本文提出的 HC 算法能够很好地满足上述的标准, 并且可以有效地应用在分布式网络中.

2 相关工作

到目前为止, top-k 查询已经受到了很多领域的重视, 例如多媒体数据上的相似性查找[1], 数字图书馆和 Web 上的半结构文档的排序查找[2], 以及电子商务中相关商品的查找[3].

Threshold Algorithm (TA)[4]算法在集中式网络环境中应用的最为广泛. TA 算法的执行步骤是不固定的, 它采用的是随机查找策略, 这种查找方法使 TA 算法查找代价大.

Three-Phase Uniform Threshold (TPUT)算法的执行步骤固定, 它是由[5]提出的, 应用于星型拓扑结构, 只需三步就能得到 top-k 结果. TPUT 算法对我们的算法影响很大, 我们借鉴了它执行步骤固定的思路.

KLEE[6]算法得到的 top-k 结果是近似结果. KLEE 算法的执行步骤也是固定的, 为 3 步或 4 步, 4 步中多出的一步为优化候选集合. KLEE 的效率和结果质量成反比, 带宽的节省量和通讯的步骤成反比.

收稿日期: 2006-07-14 基金项目: 国家自然科学基金项目(60573089; 60503036)资助; 霍英东基金优选课题项目(104027)资助. 作者简介: 李 奈, 女, 1982 年生, 硕士研究生, 研究方向为peer to peer; 王 斌, 男, 1972 年生, 讲师, 研究方向为peer to peer; 关 晶, 女, 1980 年生, 硕士研究生, 研究方向为peer to peer; 王国仁, 男, 1966 年生, 教授, 博士生导师, 研究方向为数据库理论和技术, 分布与并行数据库, web 数据管理技术, 生物信息管理, 面向对象数据库等.

现存的这些查找方法大多数都是应用在星型网络中, 当中心节点发出 top-k 查询时, 它可以和其他所有 peer 直接通讯. 但是在现实生活中, 更多的是按照树型拓扑结构连接的.

3 问题定义

假设网络中有 n 个 peer, 每个 peer $p_i(i=1,2,\cdots,n)$ 上保存着一个数据列表 I_i, I_i 中的每一个数据项都是一个数据对 $<x, v_i(x)>$, 这里的 x 是数据对象的名称, 相当于引言中的例子中的网站地址. $v_i(x)>0$ 是对象 x 在 p_i 上的值 value, 相当于例子中的点击次数. 数据列表 I_i 中的数据项都是按照 $v_i(x)$ 从高到低的顺序排列的, 并且每个有序列表 I_i 中的对象可以是不相同的.

发出 top-k 查询 Q(k) 的 peer, 我们称为 P_{init}. 查询中用到的聚合公式为:

$$V(x)=\sum_{i=1,j=1}^{i=n,j=m}\mu_i * f(\omega_j, v_i(x)) \tag{1}$$

这里的 $f(\omega, x)$ 可以是任何一个单调函数. 因为每个对象 x 在用户心目中的地位有可能不一样, 所以对于不同的对象可以赋予不同的权值 ω, 又因为每个 peer 在网络中的地位可能不一样, 所以查询时也可以对不同的 peer 赋予不同的权值 μ_i. Q(k) 的结果就是 V(x) 最大的 k 个对象 x. 我们把 Q(k) 结果中第 k 大的对象值, 称为 *topK value*.

在我们的例子中, 我们假设每个网站在广告商心目中都是平等的, 网络中的每个 peer 的地位也是平等的 (即 $\omega_j=1$, $\mu_i=1$), 聚合函数为求和, 所以

$$V(x)=\sum_{i=1}^{n} v_i(x) \tag{2}$$

广告商所要查找的就是 V(x) 最大的 50 个网址, 这里的 *topK value* 就是第 50 大的 V(x). 当然对于不同的查询, 可以根据需要采用不同的权值, 也可以采用不同的聚合方法, 在本文中, 为了方便, 我们采用的都是求和算法. 在 top-k 算法执行过程中, 需要 p_i 向 P_{init} 上传一部分数据, 称为 $list_i$. 若一个对象 x 在某个 $list_i$ 中没有出现, 我们就称这个 x 为丢失对象. 这时在计算 V(x) 时, 就会出现不同的计算方法.

定义 1 只考虑 x 的确切值, 丢失对象的值记为 0, 这样得到的聚合值称为部分和 (partial sum, 简称 PS). 公式如下:

$$PS(x)=\sum_{i=1}^{n} v_i'(x) \qquad \begin{cases} v_i'(x)=v_i(x) & if\ x\in list_i \\ v_i'(x)=0 & if\ x\notin list_i \end{cases}$$

定义 2 用丢失对象 x 在 p_i 上的最大可能值 α 代替 x 在 p_i 上的确切值, 得到 x 的"上界值"(upper bound value, 简称 UV). 公式如下:

$$UV(x)=\sum_{i=1}^{n} v_i'(x) \qquad \begin{cases} v_i'(x)=v_i(x) & if\ x\in list_i \\ v_i'(x)=\alpha & if\ x\notin list_i \end{cases}$$

定义 3 用丢失对象 x 在 p_i 上的最小可能值 β 代替 x 在 p_i 上的确切值, 得到 x 的"下界值"(lower bound value, 简称 LV). 公式如下:

$$LV(x)=\sum_{i=1}^{n} v_i'(x) \qquad \begin{cases} v_i'(x)=v_i(x) & if\ x\in list_i \\ v_i'(x)=\beta & if\ x\notin list_i \end{cases}$$

4 HB-Container 的数据管理结构

我们的算法适用于任何集中式和分布式的结构. 为了讨论问题的简单和方便, 我们的查询框架采用了超立方体结构[7]. 图 1 是一个完整的超立方体, 当 p_2 发出 Q(k) 时, 广播的路径是一棵以 p_2 为根的生成树.

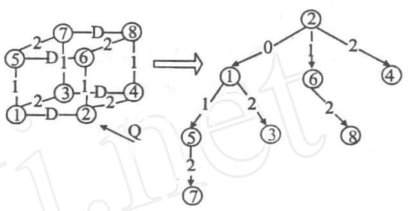

图 1 超立方体的广播路径

Fig. 1 Broadcast trace of HyperCub

在 HC 算法中, 每个 peer 的本地磁盘上除了保存数据列表 I 外, 还要为前 z% 的数据构建 hash 表, 并把剩下的数据压缩到直方图中. 这样处理本地数据是因为在进行 top-k 查询时, top-k 结果对 I_i 中靠前的数据比较依赖, 用 hash 表压缩前 z% 的数据, 能够使算法在执行时快速地获得位置靠前的对象的确切值. 而 I_i 中靠后的数据对 top-k 结果影响不是很大, 所以就用直方图对其进行压缩, 在查找时, 我们只要知道这些对象的取值范围就可以基本确定 top-k 的结果了.

我们采用的直方图是等密度的直方图, 由 n 个单元组成, 横坐标为数据对象的 value, 纵坐标为平均值 average value. 直方图的每个单元 i(i=1, 2⋯, n) 包含以下的信息:

1. 上界值和下界值: UV[i] 和 LV[i], 这两个值分别为单元 i 的左边界值和右边界值.

2. 平均值 average value: avg[i], 它是这个单元中所有对象的平均 value.

3. 对象名列表: name-filter[i], 它里面保存了对象 x (LV[i]< v(x)< UV[i]), 为了压缩数据, 这个 name-filter[i] 是由 Bloom filter 表示的.

Bloom filter 已经被应用在很多研究领域, 受到了很大的关注. 它的工作原理简单描述如下: 首先 Bloom filter 初始化是一个 b 位的向量 V, V 中每一位都为 0. 为了压缩集合 $S=\{a_1, a_2, \cdots, a_s\}$ (这里的 $a_1, a_2, \cdots, a_s$ 相当于本论文中的对象名, 即网址名称), 可以使用 h 个 hash 函数 $h_1, h_2, \cdots, h_h$, 这样 S 中的每个元素都会产生 h 个 value ($1<=value<=b$). 然后我们把向量 V 中的第 value 位置 1, 这个向量 V 就是 Bloom filter. 如果要测试一个元素 x 是否属于集合 S, 只要用相同的 h 个 hash 函数作用这个元素 x, 然后检查向量 V 中的第 $h_1(x), h_2(x), \cdots, h_h(x)$ 位, 如果有一位是 0, 就可以断定 x 不属于 S, 如果都是 1, 则可以假设它属于集合 S (之所以为"假设", 是由 hash 冲突造成的. 本来不属于 S 的元素 x, 经过测试后, 若认为它属于 S, 则称为"false positive"). 若 S 中元素个数固定, 通过调整 h 和 b 的大小, 我们可以控制 false positive. 计算 false positive (PFP) 的公式为 [6] 中提到的:

$$PFP\approx(1-1^{-hs/b})^h$$

利用Bloom filter 和直方图可以快速有效地判断一个对象名 x 的值 v(x)的范围, 这对估计 top-k 结果有很大的帮助. 由于直方图是事先构造好的, 所以构造直方图和相应的Bloom-filter 所花费的时间并不计算到我们的算法中, 算法只考虑传输直方图信息所花费的带宽和时间.

5　HC 算法描述

因为在网络中查询top-k 的主要花费在于带宽和时间, 所以我们的算法的最根本就在于减小传输的数据量, 尽量把更多的计算工作放在peer 本地完成. 当 P_{init} 发出Q(k)时, 它首先把Q(k)传给自己的孩子节点. 如果孩子节点为叶子节点, 那么就在叶子节点上开始执行算法. 如果孩子节点不是叶子节点, 则继续往下传, 直到传到叶子节点.

算法分 3 步:

1) 估计 *topKvalue*.

2) 优化候选集. 用 *topKvalue* 的估计值来尽可能多地排除不合格的对象.

3) 确定 top-k 查询的正确结果.

5.1　估计 topKvalue

估计 *topKvalue* 这一步是由一次通讯完成的, 能够确定与 *topKvalue* 有密切关系的对象的集合.

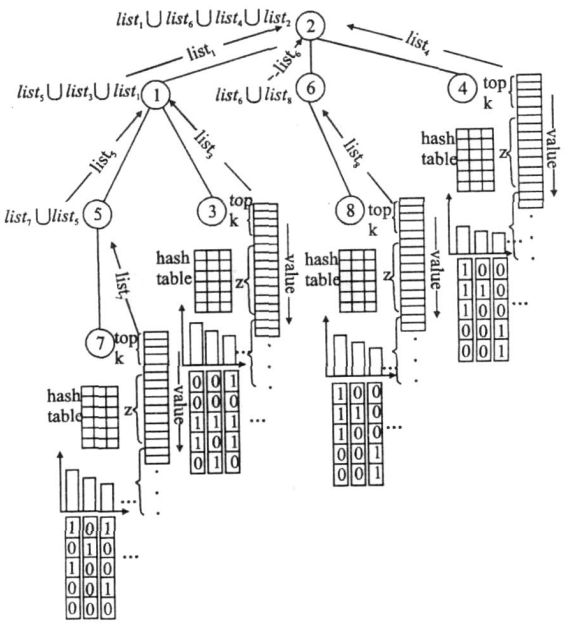

图 2　确定 *topKvalue* 的执行过程

Fig. 2　The process of determining *topKvalue*

首先从叶子节点开始, 每个叶子节点 p_i 把本地前 k 个数据项($list_i$)上传给父亲节点 p_j. 并把 $list_i$ 中的每个对象 x 的visitvector 的相应位置 1, $V(x) = V(x) + v_i(x)$. p_j 收到 $list_i$ 后, 在本地hash 表中查找 $list_i$ 中的每个对象 x, 若 hash 表中有 x, 则计算 x 的部分聚合值 $V(x) = V(x) + v_j(x)$, 并把 p_j 相应的visitvector 位置 1, 这样以后就不用再在此peer 中查找 x 了; 若 hash 表中没有x, 则在直方图的name-filter 中查找, 若找到, 则计算 x 的下界值 $LV(x) = V(x) + LV[i]$ 和上界值 $UV(x) = V(x) + UV[i]$(x 在 name-filter[i]中), 并且把 p_j 的 estimatevector 的相应位置 1, 表明 x 在该节点上查找到了上界值和下界值; 若在 name-filter 中没有找到 x, 则不改变 V(x), 但是也把 visitvector 的相应位置 1, 这是用来标明该节点中并没有对象 x, 以后就不用再来找了. 查找完 $list_i$ 中的每个对象后, p_j 把 $list_i$ 和本地的前 k 个数据项合并, 然后再发送给上一级的节点, 直至根节点. 在根节点形成的数据集合 S 就是与计算 *topKvalue* 有密切关系的数据集合. 这一步中, 每个节点上传的数据, 以及每个非叶子节点合并数据的过程如图 2 所示.

最后在根节点处得到候选集, 其中第 k 大的LV(x)就是这一步得到的 *topKvalue*. 由 *topKvalue* 可以得到阀值 T = *topKvalue*/n.

5.2　优化候选集

这是算法执行的第 2 步, 优化候选集, 它是由两步完成.

第一步是把每个 peer 中大于 T 的元素取出, 经过层层节点的合并最终在根节点形成候选集合 C. 这一步也是从叶子节点开始, 取元素时, 已经计算过的元素不重复取, 然后在上级节点中把对象名字相同的元组进行合并, 并更新此元素的visitvector. 在非叶子节点合并完成后还要检查每个元组的visitvector, 若对应的表项的值为 0, 则先在此节点中的 hash 表中查找, 若存在, 则更新此对象 x 的 V(x), 并把 visitvector 的对应项至 1; 若不存在, 则继续在Bloom filter 中查找, 若存在, 则更新它的 $LV(x) = LV(x) + LV[i]$, $UV(x) = V(x) + UV[i]$(x 在 name-filter[i]中), 即计算 x 的下界和上界, 并把 estimatevisitvetor 的相应位置 1; 若不存在, 则置 visitvector 项为 1, 但不更新数据 value 值. 在这一步, 我们的算法没有上传 bloom filter, 这是因为 bloom filter 是不可以合并的, 随着节点的增多, 越来越多的 bloom filter 会使上传的数据量变得非常大.

算法 1. fetch-CandidateSet

```
1. for every child of the peer
2.       fetch-CandidateSet;
3. fetch a value from the peer;
4. while(value> = T&& x is not visited )
5.       refresh the visitvector and value
6.       C= C∪{< x, value> };
7.       fetch next value;
8. unify the values from both the local peer and the chil
                                    dren peer into one set C;
9. for every elements in C
10.      if visitvector[I]is not true
11.         query the hashtable;
12.            if exit
13.               refresh the visitvector and value
14.            else
15.               query the bloom filter
16. return C to its parent peer;
End fetch-CandidateSet
```

第二步是对候选集合C 进行优化处理, 精简集合.

首先把 C 中第 k 大的 LV (x) 值作为 *topKvalue'*, *topKvalue'* > = *topKvalue*. 然后检查C 中每个对象 x 的 visitvector 向量和estimate vector 向量, 若一个x 的visitvector 中存在某一项为0, 而对应的estimatevector 向量为1, 则表示该x 在此项对应的节点上存在, 而且上界值已经计算过了. 若该 x 的visitvector 中存在某一项为0, 而estimatevector 中对应项也为 0, 则表明在该节点中没有查找过该对象, 那么我们就用 T 作为x 在该节点中的上界值, 并计算UV (x). 把所有 visitvector 中含有 0 的对象处理完后, 我们比较每个 x 的UV (x) 和 *topKvalue'* 的关系, 如果一个对象 x 的UV (x) < *topKvalue'*, 那么我们就直接把该x 从C 中删除. 这是因为如果x 的上界值都小于*topKvalue'*, 那么x 的真实值一定也小于*topKvalue'*, 那么x 一定不是top-k 的结果; 若一个对象x 的visitvetor 中都是 1, 那么说明该对象 x 的值V (x) 是 x 的确切值, 这时就比较V (x) 与 *topKvalue'* 的关系, 删除方法和上面相同.

5.3 确定结果

这是HC 算法的最后一步. 根据优化后的候选集合C 中的元素, 和它们的 visitvector 向量、estimatevector 向量, 在每个节点中查找它们的准确值, 最终形成准确的解集合.

对于候选集合中的每个元素项, 若其 visitvetor 项中某位为 0, 而 estimatevector 向量中对应项为 1, 那么只需要在该项对应的节点的直方图所覆盖的数据中查找确切值. 如果 estimatevector 中对应项也为0, 那么说明该对象x 在这个peer 中没有查找过, 这时就需要从该peer 的第k+ 1 位置开始查找这个x.

6 实验结果及分析

所有的测试都是在一台迅驰1. 5GHz CPU, 512M 内存的笔记本电脑上进行的. 测试系统使用的操作系统为Windows XP, 使用Microsoft Visual C++ 6. 0 编译器编译. 我们的试验是用C 编写的, 采用的是随机生成的数据, 所有的数据都存储在外存上.

我们要测试的算法有:

TA: 我们采用的是分布式TA 算法, 它是普通TA 算法的扩展. 每个peer 都把I 中的数据项打包, 每k 个数据项为一包. P_{init} 每进行一次查询, 所有其它的peer 都把本地的一包数据传送给 P_{init}.

TPUT: 这是第二节中介绍的三步执行算法.

KLEE-4: 这是4 步KLEE 算法, KLEE 中采用的直方图为等宽直方图, 宽度为30. 这些算法中, 只有 KLEE 是估计算法, 它的效率很高, 但是它的高效率是建立在损失一部分正确率的基础上的. 在测试的时候, 我们只考虑KLEE 的效率而不考虑它的准确率.

HC: 本论文介绍的算法. 我们测试的时候, 对 KLEE-4 和 HC 都采用了相同的3 个hash 函数来压缩它们的对象名.

我们要比较的性能方面有:

1) 消耗的带宽. 它是由网络中传输的总的 byte 量决定的.

2) 查询时间. 从 P_{init} 发起查询开始, 到最终得到结果所花费的时间.

我们的实验中, 直方图的每个单元中含有 20 个对象, name-filter 采用了3 个hash 函数, 并且要求bloom filter 的PFP < 0. 004. 计算执行时间时, 我们规定了磁盘的 IO 时间延迟和网络时间延迟. 每当算法查找磁盘数据时, 磁盘 IO 延迟我们规定为 9ms, 数据从磁盘传送到内存, 我们规定速度为 8MB/s. 网络延迟时间按照RTTs规定为每传送1KB 需要

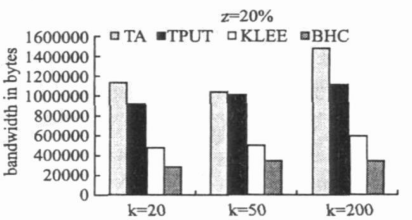

图3 z= 20% 时, 消耗的带宽

Fig. 3 z= 20%, bandwidth consumption

150ms. 整体的查询时间为本地的 IO 时间和网络传输时间之和. 因为peer 之间是并行地运行 top-k 算法, 因此每一步都只需考虑时间花费最多的那个peer. 计算TA, TPUT, KLEE 的带宽, 我们采用的公式是:

$$\sum_{i=0}^{n} \delta(p_i) * m$$

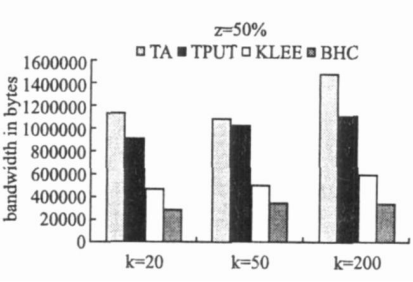

图4 z= 50% 时, 消耗的带宽

Fig. 4 z= 50%, bandwidth consumption

这里的 $\delta(p_i)$ 是 p_i 到 P_{init} 的跳数, m 是 p_i 传送的信息量. 图3, 图4 显示的是每个算法带宽的消耗量. 带宽的消耗

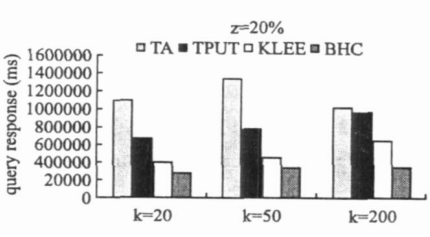

图5 z= 20% 时, 查询响应时间

Fig. 5 z= 20%, the response time

量是和传送的信息个数有关的. TA 的平均带宽消耗量是比较高的, 这是由于TA 的循环次数不确定, 而且每次上传数据

时, 上传的都是一包数据(k 个数据项), 而且 TA 没有优化候选集合的步骤, 所以它的带宽消耗量与其它算法相比是比较大的. 随着 k 的增加, TA, TPUT 和 KLEE 算法的带宽消耗也相应增加, 但是我们的 HC 算法的带宽消耗基本不变. 这是由于我们的算法能够在中间 peer 处进行合并, 并且有强大的优化机制, 这保证了, 随着 k 的增加, 带宽消耗和时间花费基本不变, 并且比其他算法都要有效.

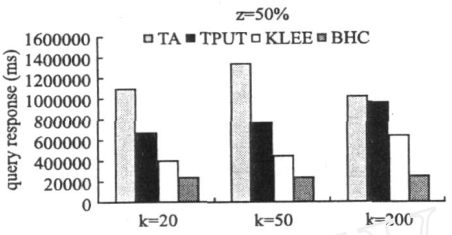

图 6　z= 50% 时, 查询响应时间

Fig. 6　z= 50%, the response time of query

图 5 和图 6 显示的是查询响应时间, 由于 TA 和 TPUT 需要大量的随机查询, 这导致 I/O 时间延迟巨大, 因此比 KLEE 和 HC 算法消耗的时间都要多. 由于 HC 的具有合并机制, 并采用了两个向量来标记查询位, 所以要比 KLEE 优越的多, 并且随着 k 的增大, 这种优势更加明显.

7　总　结

本文提出的 HC 算法仅仅需要三步就能够解决 top-k 查询, 不仅网络延迟和网络带宽花费小, 而且能够应用在任何结构的网络中. 本文基于一个树型拓扑网络详细地描述了 HC 是如何使用本地的 hash 表和直方图信息来优化查询的, 以及如何在中间节点进行部分结果的合并. 最后的实验结果表明, HC 算法比现有的 top-k 算法都要有效, 由于 HC 使用了直方图来估计结果值, 并且还使用了合并的方法来逐步合并结果, 所以它查询响应时间都比其它算法优越很多, 这使得 HC 算法更有现实意义.

References:

[1] Chaudhuri S, Gravano L, Marian A. Optimizing top-K selection queries over multimedia repositories[J]. TKDE, 2004, 16(8): 992-1009.

[2] Theobald M, Weikum G, Schenkel R. Top-k query evaluation with probabilistic guarantees[C]. VLDB, 2004, 648-659.

[3] Marian A, Gravano L, Bruno N. Evaluating top-k queries over web-accessible databases[J]. TODS, 2004, 29(2): 319-362.

[4] Fagin R, Lotem A, Naor M. Optimal aggregation algorithms for middleware[C]. In Symposium on Principles of Database Systems, 2001.

[5] Cao P, Wang Z. Efficient top-K query calculation in distributed networks[C]. PODC, 2004.

[6] Michel S, Triantafillou P, Weikum G. KLEE: a framework for distributed top-k query algorithms[C]. VLDB, 2005, 637-648.

[7] Schlosser M, Sintek M, Decker S, et al. HyperCup-hypercubes ontologies and efficient search on P2P networks[C]. Intern. Workshop on Agents and P2P Computing, Bologna, Italy, 2002, 112-124.