

文章编号: 1000-5013(2010)02-0174-06

# 在线数据流的连续限制查询算法

黄凯, 余金山

(华侨大学 计算机科学与技术学院, 福建 泉州 362021)

**摘要:** 提出并实现一个基于持续限制查询的在线系统,使其能在网络日志数据流中,当满足一定条件的事件时自动发出警告;而条件是根据用户需要而设定的,用来侦测可能严重破坏网站正常运作的行为.分析并评测时间索引算法、无时间索引算法、紧过期时间算法、紧过期时间无索引算法、多查询并行算法 5 种算法的性能.结果表明,单个查询时,无时间索引算法是最好的选择;而在多个查询时,多查询并行算法是最佳的算法.

**关键词:** 持续限制查询; 警告触发; 网络日志; 在线数据流; 算法

**中图分类号:** TP 393.07; TP 301.6

**文献标识码:** A

用户的每一次点击链接或下载的行为,都会在其终端发送一个请求给服务器.这些请求是为了从服务器上获取某个或者某些特定的文件类型,比如一个页面文件(HTML) 或一个图片文件(GIF, JPEG 等).网络日志文件全面地记录了这些请求,手动或利用软件处理记录,生成一个能反映网站用户活动的统计.近年来,通讯技术上的进步已使收集来自各种源设备(GPS 设备,各种感应器等)的具有时间和空间属性的海量信息成为可能.然而,海量而高速的信息致使对它的存储和线下分析变得不切实际,而人们对实时的数据流更加感兴趣.当限制条件在一个关于时间或者地点的流数据中全部被满足的时候,连续限制查询(Continuous Constraint Query, CCQ)<sup>[1]</sup> 就会发出一个警告.大量的连续查询(Continuous Query)<sup>[2]</sup>, 可以使用不同的限制(Constraint) 组合出来.这些限制可以是关于时间的,空间的,或者是其他的事件属性.CCQs 与传统的数据库触发器程序相似,它们都要在每一个新的时间来临的时候进行评估,而其不同之处在于 CCQ 系统并不需要在内存中存储大量的历史数据.因为限制都是和时间有关系的,系统只要存储这些时间以内的数据.本文提出并实现一个基于持续限制查询的在线系统.

## 1 基本定义

(1) 事件. 一个事件  $e$  就是一些属性的组合  $\{t, i, f_t, f_s, p\}$ .  $e.t$  是事件的时间属性(如到来时间),  $e.i$  是 IP 地址,  $e.f_t$  是文件的类型,  $e.f_s$  是文件的大小,  $e.p$  是事件其他属性的组合.

如一个样本事件: 222. 205. 47. 182- [13/Dec/2005: 00: 00: 27+ 0800]“GET /~ fit/reservation/love. mp3 HTTP/1. 1”200 5766 这一行信息里的内容是: 一个 IP 地址是“222. 205. 47. 182”的访客从服务器获取文件“/~ fit/reservation/love. mp3”, 网络协议是 HTTP/1. 1, 回应信息码是 200, 文件大小是 5 766 KB.

(2) 限制. 一个一元的限制满足以下 3 个条件: (1) 一个事件的集合  $E = \{E_1, E_2, \dots, E_m\}$ ; (2) 集合中对于每一个事件  $E_i$ , 都能够满足一定的属性限制; (3) 一个一元限制就是过滤这些事件的过滤器. 如果一个事件能够满足一个一元的限制, 那么它就变为系统的一个变量. 因此, 一个一元的限制是用来检测事件的属性是否在一个合法的定义域中, 就好像是一个过滤器, 它把那些有用的事件从海量的数据中过滤出来.

一个两元限制的关系满足如下 3 个条件: (1) 一个集合的变量  $V = \{V_1, \dots, V_m\}$ . 这些变量都是利

收稿日期: 2008-11-20

通信作者: 余金山(1952-), 男, 教授, 主要从事软件工程的研究. E-mail: yjs@hqu.edu.cn.

基金项目: 福建省自然科学基金计划资助项目(A0810013)

用上面的过滤器过滤得到的。(2) 对于任何变量对,都要进行测试二元变量。(3) 一个二元限制的集合. 一个限制  $C_{i,j}$  是一个对于变量对  $\{V_i, V_j\}$  的限制,就是这两个变量对的关系. 二元关系代表了两个事件的关系,主要包括事件时间戳的差异,IP 地址的偏移,状态码的差异,文件大小的差异.

(3) 查询. 查询就是一个组合,包括一元和二元的限制. 这些限制都是关于网络日志中事件的属性的,包括文件的大小、文件的类型、协议等等. 当这些限制被满足时,查询就会向用户发出警报.

## 2 算法设计

### 2.1 数据结构

(1) 查询索引. 查询在内存中是通过过滤器和二元限制的图来表示的. 对于每一个新事件, 查询索引会重新用全部的查询来测试这个新来的事件是否符合限制的要求. 查询索引用来快速定位事件. (2) 事件索引. 用来存储过去那些有用的事件. 每一个新事件到来的时候, 当这个事件通过某个查询的过滤器以后, 事件的索引就会找到该查询过去的事件, 用这些事件的组合来检测是否能够符合二元限制. 如果可以, 就要把这个新事件也存储到事件索引当中. (3) 过期时间索引. 根据事件的过期时间建立一个事件的索引, 从而将过期的事件在事件索引中及时地清除.

### 2.2 时间索引算法

过期时间索引是一个二叉树的数据结构, 可用来存储事件的指针和事件的过期时间. 利用过期时间的索引加速系统可定位过期的事件. 二叉树的根是过期时间(系统指定), 左子树是一个比过期时间较早的事件, 右子树是一个比过期时间较晚的事件. 然后, 根据规则对存储在内存中的事件, 建立基于二叉树的过期时间索引.

当一个新事件到来的时候, 如果这个新事件的时间戳和系统(并非真实时间)当前的时间不相同, 那么就将系统的时间更新为这个事件的最新时间. 然后, 用这个新的时间来遍历过期时间的索引. 如果事件已经过期, 就可以通过事件的指针定位并删除内存中过期事件, 以节省空间.

下一步就是检测这个新事件是否满足某个存储在查询索引中的查询. 首先, 程序会使用每个查询的第 1 个过滤器来测试这个新事件. 如果事件能够通过这个过滤器, 就要存储这个新事件到一个新的临时结果变量当中. 然后, 在过期事件索引中加入这个新事件, 以便后面快速地将它删除. 当然, 这个事件还可能通过查询的其他过滤器或者其他查询的过滤器, 程序也会将其存储在内存中对应的临时结果当中. 也就是说, 一个事件可能会被多个临时结果所存储, 即程序不会漏掉任何一个查询结果.

最后就是检测新事件到来之后, 查询是否被满足. 每一个查询都有一个属性表示它所允许的最大的事件个数, 如果临时结果的事件个数等于这个查询的最大事件个数, 那么查询被满足, 系统发出警报, 同时删除这个临时变量.

算法的伪代码:

```

if(S.t < E.t)
{
    S.t = E.t;
    for( each expired event in the expire_table)
        delete(expired event);
}
Get all (Vi, G) pairs such that G in Q, Vi in G;
for each (Vi, G) do
    Vi.sat = true;
    Vi.timeToLive = G.timeToLive;
    for each Vj in Vi.past do
        use Vi = e, Ci,j, and Cj to test
        if unsatisfied{
            Vi.sat = false;

```

```
        break;
    }
    if V i. sat {
        expire_table.append(V i);
        if(isSatisfied( V i. past, V i, C i, j in C))
            if( V i. past and V i can satisfy the whole query)
                alert(query);
            else
                store(V i);
    }
    read next event.
```

其中: S 代表系统, S. t 代表系统的时间, E 是新到来的事件, V. past 表示在临时结果中的过去的变量, Q 是查询的索引, C 是查询中存储限制的数据结构.

2.3 无时间索引算法

无时间索引算法不使用过期时间, 即每一个事件使用一个属性来说明自己的过期时间. 当事件被存储在临时结果的第 1 个位置时, 其属性就会决定这个临时结果何时被删除. 算法根据查询的二元限制来决定这个过期时间的大小, 当然这个时间是越小越精确越好, 能及时删除过期的事件, 节约内存.

当一个新事件到来的时候, 如果这个新事件的时间戳和系统(并非真实时间)当前的时间不相同, 就要将系统的时间更新为事件的最新时间; 然后, 检查并删除全部的过期的临时结果; 如果新到来的事件的时间戳和系统当前存储的时间戳是相同的, 或者小于系统当前的时间戳(在有些网络日志中可能会出现这种情况), 就不用检查过期事件.

剩下的部分和时间索引算法大体相似, 各参数代表的含义也相同. 算法的伪代码:

```
if(S. t < E. t)
{
    S. t= E. t;
    for( each event stored)
        delete(expired event);
}
Get all (V i, G) pairs such that G in Q, V i in G;
for each (V i, G) do
    V i. sat:= true;
    V i. timeToLive= G. timeToLive;
    for each V j in V i. past do
        use V i= e, C i, j, and C j to test
        if unsatisfied{
            V i. sat:= false;
            break;
        }
    if V i. sat {
        if(isSatisfied( V i. past, V i, C i, j in C))
            if( V i. past and V i can satisfy the whole query)
                alert(query);
            else
                store(V i);
    }
}
```

read next event.

2.4 紧过期时间算法

利用 Dijkstra 算法<sup>[34]</sup> 计算紧密事件的过期时间. 与前面算法的不同的是动态更新过期时间的方法. 为定义一个紧密的过期时间, 当一个新事件到来时, 如果它能够符合查询中对应的过滤器和二元限制时, 就根据查询中的过期时间数组来更新这个事件的临时结果. 这就是一个紧密时间的版本.

这个算法也使用二叉树作为过期时间的索引. 但是, 每一个事件的过期时间是随着后面事件的过期时间而改变的, 需要根据新来的数据不停地修改前面存储的过期时间. 不停地维护和更新过期时间索引的代价是很高的, 所以在一些情况下, 使用过期时间索引可能会降低整个系统的效率.

2.5 紧过期时间无索引算法

在紧过期时间算法的基础上, 不使用过期时间索引, 而是在每一个新事件到来时, 采用无时间索引算法的方法扫描过去的事件. 当一个新事件到来时, 如果其时间戳和系统(并非真实时间)的当前时间不相同, 就将系统的时间更新为新事件的最新时间, 检查并删除全部的过期临时结果. 如果新到来的事件的时间戳和系统当前存储的时间戳是相同的, 或者小于系统当前的时间戳(在有些网络日志中可能会出现这种情况), 就不用检查过期事件.

然后, 算法检测该事件是否能够符合某个查询或多个查询, 每一个查询都可能包含多个过滤器和二元的限制. 如果该事件能够通过过滤器和二元限制, 将该事件存储到临时结果中, 并更新这个临时结果的过期时间; 反之, 删除该事件.

最后, 算法检测这个最新改变的临时结果, 看它是否能够满足所对应的查询的要求. 如果可以, 发出警报; 反之, 保存这个临时结果直到它过期. 这个算法不需要一直维护一个过期时间索引.

2.6 多查询并行算法<sup>[5]</sup>

前面所提到的算法, 其临时结果都是存储在一个数据结构之中. 如果只是执行一个查询, 对程序的性能是不会有影响的, 但是, 如果同时进行多个查询, 就要线性地扫描那些不相关的临时结果. 所以, 多查询并行算法是将这些临时结果分布到不同的数据结构之中. 属于不同的查询, 能够通过不同过滤器和满足二元限制的临时结果, 将被存储在不同的数据结构之中.

当一个新事件到来时, 检测其能否满足任意一个查询的过滤器. 如果可以, 则将其加入到一个对应的查询过滤器的数据结构中. 运行代价将会是一个常量值, 而不是前面的算法中的  $O(n)$ . 然后, 根据临时结果能否满足查询中的二元限制, 把它移动到对应的下一个过滤器中或者发出一个警报, 并删除临时结果. 如果新事件对于每一个临时结果都没有用途的话, 从事件索引中将其删除, 再读入下一个事件.

该算法有一个明显的好处, 就是它不会因为同时进行多个查询而降低系统的效率. 由于其不具有过期时间索引, 从而不需要时时地维护. 但是, 它可能要使用更多的内存来存储事件.

3 算法的性能评测

测试数据是由某网络部门提供的网络日志真实数据集. 在测试当中, 计算机的中央处理器(CPU)的占用率始终接近 100%, 因此, 只研究在处理相同查询的情况下, 各种算法的运行时间消耗、内存占用的峰值及平均值. 时间索引算法、无时间索引算法、紧过期时间算法、紧过期时间无索引算法、多查询并行算法分别以编号 A1, A2, A3, A4, A5 表示.

3.1 不同输入行数的运行时间消耗

在其他条件不变的情况下, 改变读入文件中数据的行数来评测系统中各个算法的性能. 运行时间( $t$ )消耗的测试结果, 如表 1 所示. 表 1 中:  $N_1$  为输入行数,  $m$  为警告数目. 从表 1 可知, 运行时间随输入行数的增长呈线性增长. 算法的时间复杂度是  $O(n)$ , 在单查询的情况下, 各个算法的时间差距很微小. 总体来说, A2 是效率最高的算法, 而 A1 的效率最低.

在单查询测试中, 时间索引的创建和操作维护需要太多的时间, 远远大于其他结构操作需要的时间, 所以包含有时间索引的算法(A1, A3) 时间效率比较低. A3 的过期时间比较短, 当输入数据的行数增加到一定程度, A3 的效率将变得越来越高. 另外的 3 种算法不包含时间索引, 所以效率比较高. A4 比 A2 效率稍微低一些, 是因为过期时间的计算需要时间. 至于 A5, 在单查询的情况下性能表现一般.

3.2 不同输入行数的内存占用

不同输入行数的内存使用峰值( $k$ )和内存使用平均值( $k_{av}$ )的测试结果,如表1所示.在内存峰值测试中,各种算法得出结果几乎相等.因为在单查询的情况下,只有很少的警告分布在海量的事件中,而一旦临时结果无用,算法立刻释放占用的内存,故内存峰值相差无几.从内存峰值中看不出各算法在内存使用效率方面的差别,所以需要通过内存使用平均值来判别效率的高低.

包含紧过期时间索引的算法(A3, A4, A5)有着较低的平均内存占用率,其中A3和A4的占用率是相同的, A5则略高一点.而A1比A2的平均内存占用率要高一些.

在包含紧过期时间索引的算法中,过期时间的值设定得相对比较小.事件槽中存储的临时结果一旦超出了过期时间的范围,将被认为是无用数据,它们的内存空间槽将立刻被释放.当这种情况足够多时,用于存储临时结果的空间索引将一直保持较小的状态,从而使得这3种算法的平均内存占用率较低,也就是空间效率较高.另外, A1, A2算法在内存管理机制上是相似的, A1比A2多耗费一个时间索引上的内存空间,所以, A1的内存平均占用率要高过A2.

3.3 不同查询数的运行时间消耗

在这个测试中,将网络日志输入数据的行数锁定在1 000 000行,只改变查询的数目( $N_2$ ),并且让查询的数目呈线性增长,以评测各算法在多查询情况下的性能表现.运行时间测试结果如表2所示.

表 1 不同输入行数的算法性能

Tab.1 Algorithm performance  
in different input lines

| $N_1$ | $t/s$   |           |           |           |           |
|-------|---------|-----------|-----------|-----------|-----------|
|       | 500 000 | 1 000 000 | 1 500 000 | 2 000 000 | 2 500 000 |
| $m$   | 32      | 58        | 86        | 223       | 254       |
| A1    | 12.000  | 25.766    | 39.313    | 52.313    | 66.156    |
| A2    | 10.687  | 24.078    | 35.406    | 48.375    | 62.39     |
| A3    | 12.125  | 24.765    | 36.921    | 49.359    | 62.188    |
| A4    | 11.235  | 24.703    | 37.156    | 50.235    | 65.421    |
| A5    | 11.578  | 26.094    | 37.485    | 51.485    | 66.953    |

| $N_1$ | $k/\text{事件槽}$ |           |           |           |           |
|-------|----------------|-----------|-----------|-----------|-----------|
|       | 500 000        | 1 000 000 | 1 500 000 | 2 000 000 | 2 500 000 |
| $m$   | 32             | 58        | 86        | 223       | 254       |
| A1    | 13             | 14        | 14        | 14        | 14        |
| A2    | 13             | 13        | 14        | 14        | 14        |
| A3    | 13             | 13        | 14        | 14        | 14        |
| A4    | 13             | 13        | 14        | 14        | 14        |
| A5    | 13             | 13        | 14        | 14        | 14        |

| $N_1$ | $k_{av} \times 10^{-3}/\text{事件槽}$ |           |           |           |           |
|-------|------------------------------------|-----------|-----------|-----------|-----------|
|       | 500 000                            | 1 000 000 | 1 500 000 | 2 000 000 | 2 500 000 |
| $m$   | 32                                 | 58        | 86        | 223       | 254       |
| A1    | 7.516 0                            | 18.393 0  | 18.376 0  | 21.224 5  | 19.775 2  |
| A2    | 6.770 0                            | 16.562 0  | 16.639 3  | 19.289 0  | 17.966 0  |
| A3    | 5.598 0                            | 13.381 0  | 13.270 0  | 15.347 0  | 14.263 2  |
| A4    | 5.598 0                            | 13.381 0  | 13.270 0  | 15.347 0  | 14.263 2  |
| A5    | 5.630 0                            | 13.435 0  | 13.326 0  | 15.425 0  | 14.342 4  |

表 2 不同查询数的算法性能

Tab.2 Algorithm performance  
in different query number

| $N_2$ | $t/s$  |        |        |        |
|-------|--------|--------|--------|--------|
|       | 2      | 4      | 6      | 8      |
| $m$   | 1 485  | 1 586  | 1 862  | 2 705  |
| A1    | 30.468 | 31.796 | 40.078 | 47.672 |
| A2    | 27.109 | 28.062 | 35.453 | 40.625 |
| A3    | 26.578 | 27.531 | 28.921 | 29.86  |
| A4    | 26.781 | 27.437 | 28.766 | 29.875 |
| A5    | 26.843 | 26.971 | 27.537 | 27.815 |

| $N_2$ | $k/\text{事件槽}$ |       |       |       |
|-------|----------------|-------|-------|-------|
|       | 2              | 4     | 6     | 8     |
| $m$   | 1 485          | 1 586 | 1 862 | 2 705 |
| A1    | 201            | 402   | 668   | 866   |
| A2    | 200            | 400   | 668   | 862   |
| A3    | 199            | 398   | 586   | 785   |
| A4    | 199            | 398   | 586   | 785   |
| A5    | 199            | 398   | 619   | 810   |

| $N_2$ | $k_{av}/\text{事件槽}$ |           |           |            |
|-------|---------------------|-----------|-----------|------------|
|       | 2                   | 4         | 6         | 8          |
| $m$   | 1 485               | 1 586     | 1 862     | 2 705      |
| A1    | 1.014 140           | 2.029 180 | 7.662 820 | 13.148 100 |
| A2    | 0.791 899           | 1.578 070 | 7.410 210 | 11.146 500 |
| A3    | 0.224 026           | 0.408 890 | 0.599 092 | 0.909 103  |
| A4    | 0.224 026           | 0.408 890 | 0.599 092 | 0.909 103  |
| A5    | 0.230 466           | 0.427 070 | 0.634 361 | 0.961 198  |

在运行时间的测试中, A3, A4, A5的结果接近线性增长,而且彼此之间十分接近,所以,此3种算法的时间复杂度是 $O(n)$ .在多查询情况下, A5是效率最高,而A1运行得最慢;当查询数目越来越多时, A5的时间效率优势将越来越明显.

在多查询情况下,包含紧过期时间的算法(A3, A4, A5)效率高.在警告密度很高的情况下,用在时间索引上的时间消耗变得微不足道,甚至可以被忽略.因此,只需要考虑用在对事件索引操作上的时间,紧过期时间算法能把事件索引的体积保持在一个很小的状态,而且比非紧过期时间算法要小得多.事件

索引体积小, 消耗在其上面的时间也会少得多, 所以 A3, A4, A5 要比 A1, A2 的时间效率高得多.

3.4 不同查询数的内存占用

不同查询数的内存使用峰值( $k$ ) 和内存使用平均值( $k_{av}$ ) 的测试结果, 如表 2 所示. 从表 2 可知, 紧过期时间算法( A3, A4, A5) 在空间利用上表现得非常出色, 不管是内存峰值还是平均值. 紧过期时间算法( A3, A4, A5) 有着较少的内存使用峰值, 而 A1 的峰值比 A2 稍高一些.

在包含紧过期时间索引的算法中, 临时结果一旦超出过期时间, 就会被认定为无用数据就被移出事件索引, 以节省内存空间, 而主要的内存占用就是在事件索引上. 所以, 算法的内存峰值和平均值也就少了. 其中, A3, A4 的占用率是相同的, 它们在内存管理机制上是相同的. A1, A2 也是如此, 但 A1 在时间索引上还要消耗一定的空间, 所以内存占用比 A2 稍多一点. 在内存占用平均值的测试中, 紧过期时间算法显示出了更加明显的优势, 原因和峰值测试相似, 不再赘述.

4 结束语

所研究的应用领域只是处理网络日志数据流, 且是只有时间属性的数据. 在解决 CCQ 问题上, 时间效率要比空间效率重要得多, 特别是当空间花费之间的差别十分微小的时候. 通过修改事件读入模块, 可以将此系统应用于更多其他领域, 比如可以应用于全球定位系统(GPS), 将事件扩展到含有空间属性的领域内, 这也是今后工作的一个方向.

参考文献:

[ 1 ] MARIOS H, NIKOS M, YUFEI T. Lecture notes in computer science: Continuous constraint query evaluation for spatiotemporal streams[ M]. Berlin: Springer, 2007: 348-365.

[ 2 ] CHEN Jian-jun, DEWITT D J, TIAN F, et al. Niagara CQ: A scalable continuous query system for Internet databases[ C] // Proc of the 2000 ACM SIGMOD International Conference Management of Data. New York: ACM , 2000: 379-390.

[ 3 ] HELGASON R V, KENNINGTON J L, STEWART B D. The one-to-one shortest-path problem: An empirical analysis with the two-three Dijkstra algorithm[ J]. Computational Optimization and Applications, 1993, 2( 1): 47-75.

[ 4 ] KUZNETSOV N A, FETISOV V N. Enhanced robustness Dijkstra algorithm for control of routing in the IP-networks[ J]. Automation and Remote Control, 2008, 69( 2): 247-251.

[ 5 ] 刘佳, 刘国华, 宋驰. 基于流数据技术的连续查询处理[ J]. 计算机工程, 2005, 31( 8): 74-77.

Algorithm of the Continuous Constraint Queries  
Based on the Online Data Stream

HUANG Kai, YU Jin-shan

( College of Computer Science and Technology, Huaqiao University, Quanzhou 362021, China)

**Abstract:** An on-line system on the basis of continuous constraint queries (CCQ) is proposed and constructed, which can trigger alerts whenever some events from data that arrive from an online Web log stream satisfy some specific patterns. These patterns are set by the user in order to detect the abnormal situation that could ruin the Web site significantly. The five algorithms, time index algorithm, no time index algorithm, close overdue time algorithm, close overdue time no index algorithm and multi-query parallel algorithm, are analyzed and evaluated. The result show that no time index algorithm is the best choice during single query and multi-query parallel algorithm is the best algorithm during multi query.

**Keywords:** continuous constraint query; alert triggering; Web log; online data stream; algorithm

( 责任编辑: 黄仲一 英文审校: 吴逢铁)