使用MapReduce框架实现SimRank算法

algorithm (1) (/categories.html#algorithm-ref)

algorithm 1 (/tags.html#algorithm-ref) mapreduce 2 (/tags.html#mapreduce-ref) recommendation 1 (/tags.html#recommendation-ref) simrank 1 (/tags.html#simrank-ref)

02 January 2014

Background

SimRank是一种基于图结构的相似性度量算法，算法依赖于一个基本的事实**“two objects are similar if they are related to similar objects”**。简单来说，如果两个物体都与另一物体有关联，那么这两个物体就有一定的相似度，因此SimRank是从图的结构来描述物体之间的相似度。本文从SimRank算法入手，介绍算法的时空复杂度和mapreduce框架下的实现，分析其优缺点和框架带来的限制。

Basic SimRank算法

算法定义

Wikipedia (http://en.wikipedia.org/wiki/SimRank)对于SimRank算法的描述是：

> SimRank is a general similarity measure, based on a simple and intuitive graph -theoretic model. SimRank is applicable in any domain with object-to-object relationships, that measures similarity of the structural context in which objects occur, based on their relationships with other objects. Effectively, SimRank is a measure that says “two objects are similar if they are related to similar objects.”

对于图中的顶点v，我们假定$I(v)$和$O(v)$分别为该顶点的in-neighbors和out-neighbors，每一个in-neighbor表示为$I_i(v)$，其中$1 \leq i \leq |I(v)|$；每一个out-neighbor表示为$O_i(v)$，其中$1 \leq i \leq |O(v)|$.

我们以$s(a,b)$来表示物体a和b之间的相似度,约定$s(a,b) \in [0,1]$,则:

$$s(a,b) = \begin{cases} 1 & a = b \\ \frac{C}{|I(a)||I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{I(b)} s(I_i(a), I_j(b)) & a \neq b \end{cases}$$

其中C是介于0和1之间的常数。如果a或b没有任何in-neighbors，则$s(a,b)=0$。

以上公式给出了Simrank算法的递归形式，那么实际应该如何求解SimRank呢？一种最基本的方法是通过迭代以上公式使SimRank值收敛于一个固定的值，迭代公式如下所示：

$$R_0(a,b)=\begin{cases}1 & \text{if } a=b,\\ 0 & \text{if } a\neq b.\end{cases}$$

$$R_{k+1}(a,b)=\begin{cases}\frac{C}{|I(a)||I(b)|}\sum_{i=1}^{|I(a)|}\sum_{j=1}^{I(b)}R_k(I_i(a),I_j(b)) & a\neq b\\ 1 & a=b\end{cases}$$

第$k+1$次迭代中(a,b)的相似度是由(a,b)的所有邻接点对在第k次迭代中的相似度累加所得。$R_k(*,*)$的值的更新是单调非递减的，它的值最终会收敛于某一固定值$s(*,*)$。对于任意$a,b\in V$，$\lim\limits_{k\to\infty}R_k(a,b)=s(a,b)$。

注意这里所指的邻接并不是图G上的邻接点，而是pair graph G^2上的邻接点对。

我们假定图中任意点的平均入度是p，那么上述公式的时间复杂度就是$O(p^2)$，假定图中有n个顶点，那么计算所有顶点两两之间的相似度的时间复杂度是$O(p^2n^2)$。

以上给出了SimRank算法的定义和迭代公式，根据迭代公式我们就可以得出任意两顶点之间的相似度，接下来我们就要介绍MapReduce框架如何实现上述算法。

MapReduce实现和分析

MapReduce算法的伪代码如下所示：

```
1.  Algorithm1: Computing SimRank on MapReduce
2.  Input: Graph G, initialized S0
3.    for t = 0: T-1
4.      Map Function((a,b), St(a,b))
5.      find a,b's neighbor I(a) and I(b) respectively
6.      for each c belongs I(a), d belongs I(b)
7.        output ((c,d), St(a,b))
8.      Reduce Function (Key=(c,d), Values=vs[])
9.        if c = d
10.         St+1(c,d) = 1
11.       else
12.         St+1(c,d) = C/len(vs)(sum(vs))
13.       output(c,d),St+1(c,d)
14. Output: update St
```

在Map阶段对于每一个点对(a,b)，求出其所有邻接点对，并把SimRank值贡献给这些点对。在Reduce阶段把相同点对的值都加起来就得出了给点对的SimRank值。

这里所描述的MapReduce算法是对**Basic SimRank**算法的实现，存在着几个问题：

1. 对于每一个点对(a,b)，我们需要求出$I(a)$和$I(b)$，并对其做笛卡尔积来求出邻接点对，这里的平均时间复杂度是$O(p^2)$，最坏的情况下会达到$O(n^2)$。如果图中边的分布是80-20分布或是Zipfian分布的话，极有可能造成的Map task处理时间的严重倾斜。

2. 同时由上可知平均的shuffle数据量是$O(p^2n^2)$，最坏的情况下会达到$O(n^4)$。在数据规模很大的情况下，shuffle数据量将变得非常大，整个算法将会变成IO intensive的算法。
3. 相比于处理能力的线性增长，算法复杂度的增长将会是平方级，不管是时间还是空间复杂度上。

Delta SimRank算法

在上面的分析中我们已经看到了**Basic SimRank**算法的有比较高的时空复杂度，同时shuffle量也达到了$O(p^2n^2)$，为了能够降低算法的复杂度，这里引入了**Delta SimRank** (http://dprg.cs.uiuc.edu/docs/deltasimrank/simrank.pdf)算法。算法将传统的直接计算SimRank值变为计算delta SimRank值，并且将小于ϵ的值忽略掉，这样大大减少了计算和shuffle的数据量，降低了时空复杂度。同时该算法也证明了在舍弃小值的情况下虽然会丢失一定的精度，但是误差的范围是可以计算并获得的，同时相似性的单调非递减特性也依然保持。下面给出算法的迭代公式：

$$\Delta^{t+1}(a,b) = \frac{C}{|I(a)||I(B)|}\sum_{c\in I(a)}\sum_{d\in I(b)}\Delta^t(c,d)$$

$$s^{t+1}(a,b) = s^t(a,b) + \Delta^{t+1}(a,b)$$

MapReduce伪代码如下所示：

```
Algorithm2: Computing Delta-SimRank on MapReduce
Input: Graph G, initialized delta_t
Map function((a,b), delta_t(a,b))
  if a = b or delta_t(a,b) <= eplison
    return
  find a,b's neighbor I(a) and I(b) respectively
  for each c belongs to I(a), d belongs I(b)
    output (c,d), c/(|I(c)||I(d)|)delta_t(a,b)
Reduce function (Key=(c,d), Values=vs[])
  if c = d
    output delta_t+1(c,d) = 0
  else
    output delta_t+1 = sum(vs)
Output update delta_t+1

Algorithm3: An efficient approach to compute SimRank
Input: Graph G, init SimRank s0
Update SimRank using Algorithm 1 and obtain s1
init Delta-SimRankk by delta_1 = s1 - s0
for t = 1: T-1
  update delta_t+1 SimRank as in Algorithm 2.
  St+1 = st + delta_t+1
Output: updated SimRank score St
```

与**Basic SimRank**算法的MapReduce实现相比较，**Delta-SimRank**算法的主干非常相似，唯一不同的是**Delta-SimRank**迭代计算delta值，并把delta值加回到SimRank值上去，而非直接迭代计算SimRank值。因此相比于**Basic SimRank**的MapReduce实现，**Delta-SimRank**多了一个回加的步骤。

Delta-SimRank算法的时间复杂度是$O(p^2M)$，shuffle的空间复杂度也是$O(p^2M)$，这里M由ϵ决定，当ϵ较大的时候，就会过滤掉更多的delta值，计算复杂度的下降就会更快；反之若ϵ较小，则delta值大多被保留，计算复杂度就会趋近于**Basic SimRank**算法，但是精确度会更高。

同时迭代中delta值的大小由衰减系数(C)决定，如果C较大，delta值的衰减就会比较慢，计算复杂度无法快速下降；当然如果C较小的话，复杂度在2-3轮迭代后就会显著减小。

因此**Delta-SimRank**算法的复杂度由两个值决定：衰减系数C和阈值ϵ。调整这两个值可以在精度和复杂度上取一个平衡点。

SimRank Matrix Multiplication Method

Basic SimRank的迭代公式可以改写为矩阵相乘的形式，这样可以利用矩阵乘法的MapReduce实现就求解SimRank值。

首先我们将**Basic SimRank**迭代公式改写为：

$$s_{a,b}^0 = \begin{cases} 1 & \text{if } a = b, \\ 0 & \text{if } a \neq b. \end{cases}$$

$$\begin{aligned} s_{a,b}^{k+1} &= \frac{C}{|I(a)||I(b)|} \sum_{i=1}^{n} \sum_{j=1}^{n} p_{i,a} s_{i,j}^{(k)} p_{j,b} \\ &= c \sum_{i=1}^{n} \sum_{i=1}^{n} (\frac{p_{i,a}}{\sum_{i=1}^{n} p_{i,a}}) s_{i,j}^{(k)} (\frac{p_{j,b}}{\sum_{j=1} np_{j,b}}) \end{aligned}$$

在这里我们假定$a \neq b$, ($s_{a,a}^k \equiv 1 (k = 0, 1, \ldots)$)。

这样上述公式的矩阵形式就如下所示：

$$\begin{cases} S^0 = I_n \\ S^{(k+1)} = (cQS^{(k)}Q^T) \vee I_n \quad （\forall k = 0, 1, \ldots) \end{cases}$$

这里邻接矩阵Q是按列归一化的。

Simrank Matrix Multiplication算法的时间复杂度是$O(n^3)$，在这里由于邻接矩阵Q是稀疏矩阵，平均入度是p，算法复杂度降为$O(p^2n^2)$，和上述**Basic SimRank**的时间复杂度相同。

对于shuffle的数据量，由于矩阵乘法在MapReduce框架中有几种不同的实现方式，而不同的实现方式shuffle的数据量是不同的。

首先我们考虑一下通用的pair-wise的矩阵乘法，假定$S = AB$，那么$S_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$，也就是说$A$中的点$a_{ik}$和$B$中的点$b_{kj}$只贡献了结果$s_{ij}$的1/n。那么以这种方式进行**Simrank Matrix Multiplication**其shuffle数据量将达到$O(n^3)$。

其次可以考虑column-wise或是row-wise的矩阵乘法，这样在每个task中计算得到的结果就是乘法最后的结果，无需再相加求和。其shuffle数据规模是$O(n^2M)$，其中M是task的个数。

最后如果每个node的内存足以存放邻接矩阵，我们可以将邻接矩阵分布到每个node上去，那么**Simrank Matrix Multiplication**算法的shuffle数据量可以降到$O(n^2)$。当然所有的前提建立在每个node的内存足以存放整个邻接矩阵。

随着迭代的进行SimRank矩阵S会逐渐变为稠密矩阵，这时候shuffle的数据量就会变得非常巨大，因此对于S降维或是丢弃一些小值也是一种可行的方法。

End

本文介绍了三种SimRank算法在MapReduce框架下的实现方式，包括**Basic SimRank**，**Delta-SimRank**，**SimRank Matrix Multiplication**，并分析了各自的时间复杂度和shuffle数据量。在这里分析shuffle数据量的原因是在MapReduce框架中shuffle的数据规模直接影响了整个算法的performance。其中**Basic SimRank**的MapReduce实现是一种通用的实现，整个程序的bottleneck集中于IO，IO的性能直接影响了整个算法的速度。而**Delta-SimRank**是一种丢失精度的实现方式，算法的复杂度与衰减系数和阈值有直接的联系，不同的取值对于整个算法的速度有不同的结果。最后介绍了**SimRank Matrix Multiplication**的实现方式，该算法将求解SimRank算法转变为矩阵乘法，根据数据规模的不同矩阵乘法有许多不同的MapReduce实现，直接影响了最后整个算法的速度。

在不损失精度的情况下，SimRank算法的复杂度相对比较高，尤其是当数据量达到一定规模时，整个算法的执行时间将变得非常之长，同时对于磁盘容量，吞吐率和网络带宽也是一个不小的考验。因此现实中的算法往往会舍弃一些精度来换取较低的时空复杂度，可以参考：

A Space and Time Efficient Algorithm for SimRank Computation
(http://www.cse.unsw.edu.au/~zhangw/files/wwwj.pdf)

Fast Computation of SimRank for Static and Dynamic Information Networks
(http://www.cs.uiuc.edu/~hanj/pdf/edbt10_cli.pdf)

当然如果跳出MapReduce框架，使用异步框架也可能带来更好的性能，如Asyn-SimRank:一种异步执行的大规模SimRank算法 (http://faculty.neu.edu.cn/cc/zhangyf/papers/async-simrank-ccfbigdata.pdf)。

最后还要说明的是计算相似度有许多不同的算法，虽然SimRank算法比较直观和易理解，但是其高时空复杂度是一个不可回避的问题，综合比较不同算法，考虑自身的处理能力以选择一个最佳的算法才是上策。

← Previous (/architecture/2013/10/08/spark-storage-module-analysis) | Archive (/archive.html)
Next → (/architecture/2014/01/04/spark-shuffle-detail-investigation)

CATEGORIES

test (1) (/categories.html#test-ref)

architecture (8) (/categories.html#architecture-ref)

functional programming (1) (/categories.html#functional programming-ref)

algorithm (1) (/categories.html#algorithm-ref)

LINKS

阮一峰的网络日志 (http://www.ruanyifeng.com/blog/)

刘未鹏 (http://mindhacks.cn/)

酷壳 (http://coolshell.cn/)

BeiYuu.com (http://beiyuu.com/)

MY FAVORITES