软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(12):3051−3066 [doi: 10.13328/j.cnki.jos.005012]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

Top-*k* 相似连接算法性能优化*

王洪亚, 杨利宏, 刘晓强

(东华大学 计算机科学与技术学院,上海 201620)

通讯作者: 王洪亚, E-mail: hywang@dhu.edu.cn

摘 要: 相似连接算法在数据清理、数据集成和重复网页检测等领域有着广泛的应用.现有相似连接算法有两种类型:基于相似度阈值的相似连接和Top-*k*相似连接.Top-*k*连接算法非常适合于相似度阈值未知的应用场景,目前最为有效的Top-*k*相似连接算法是Xiao等人提出的Top*k*-join.为了解决Top*k*-join中存在的性能问题,提出了一种Top-*k*相似连接算法 Opt-join,该算法将 Token 批处理技术集成在现有的事件驱动框架中,以降低前缀事件的处理代价;通过置换哈希查找与过滤操作的执行位置来降低哈希查找代价,并理论证明了该置换的正确性.实验结果表明:与Top*k*-join算法相比,Opt-join取得了1.28倍~3.09倍的性能提升.实验数据还显示:随着数据长度的增加或*k*值的增长,Opt-join的性能优势有不断增加的趋势.

关键词: Top-*k* 相似连接;事件驱动框架;Token 批处理;哈希查找优化

中图法分类号: TP311

中文引用格式: 王洪亚,杨利宏,刘晓强.Top-*k* 相似连接算法性能优化.软件学报,2016,27(12):3051−3066. http://www.jos.org.cn/1000-9825/5012.htm

英文引用格式: Wang HY, Yang LH, Liu XQ. Optimizing top-*k* similarity join algorithm. Ruan Jian Xue Bao/Journal of Software, 2016,27(12):3051−3066 (in Chinese). http://www.jos.org.cn/1000-9825/5012.htm

Optimizing Top-*k* Similarity Join Algorithm

WANG Hong-Ya, YANG Li-Hong, LIU Xiao-Qiang

(School of Computer Science and Technology, Donghua University, Shanghai 201620, China)

Abstract: Similarity join is widely used in data cleaning, data integration and the detection of near duplicate Web pages. Existing similarity join algorithms fall into two categories: Threshold-based similarity join and Top-*k* similarity join. Top-*k* similarity join is suitable for applications in which the threshold is unknown in advance. The most efficient Top-*k* similarity join algorithm is Top-*k*-join, which is proposed by Xiao et al. In order to resolve the performance problemsof Top*k*-join, a novel Top-*k* similarity join algorithm Opt-join is proposed in this paper. By integrating the token batch processing technique into the existing event-driven framework, Opt-join reduces the cost of processing the prefix events. In addition, Opt-joinreduces the cost in hash lookup by switching the positions of the hash lookup and filtering operations. The correctness of the new algorithm is proved. Experimental results show that 1.28x-3.09xspeed-up is achieved by Opt-join compared with Top*k*-join. More importantly, with the increase of the record length or the *k* value, Opt-join surpasses Topk-join by a larger margin.

Key words: top-*k* similarity join; event-driven framework; token batch processing; hash lookupoptimization

近似重复数据(near duplicate data)是具有很高相似度但又不完全相同的数据.产生重复数据的原因很多,例如印刷错误、镜像页面和完全或部分剽窃的文档等.相似连接(similarity join)是一种有效查找数据集中相似记

* 基金项目: 国家自然科学基金(61370205); 上海市自然科学基金(13ZR1400800); 中央高校基本科研业务费专项资金

Foundation item: National Natural Science Foundation of China (61370205); Shanghai Municipal Natural Science Foundation (13ZR1400800); The Fundamental Research Funds for the Central Universities

收稿时间: 2015-06-11; 修改时间: 2015-09-08; 采用时间: 2015-12-04; jos 在线出版时间: 2016-01-16

CNKI 网络优先出版: 2016-01-18 13:50:53, http://www.cnki.net/kcms/detail/11.2560.TP.20160118.1350.001.html

录对的方法,因而在数据清理、数据集成和近似网页去重等领域得到了广泛的应用[1−24].例如:在数据清理中,通过相似连接算法,可以为不同数据源提供准确、一致的数据;在数据集成中,通过相似连接对命名实体识别进行归类;在镜像页面中,使用相似连接抓取相似网页,提高网页搜索引擎的执行效率.

目前,相似连接主要有两种类型:一是基于相似阈值的相似连接[1,2,4−9,10−23],即,找出数据集中相似值不小于阈值(由用户自定义)的记录对;另一种是 Top-k 相似连接[3],即,找出数据集中相似度最高的前 k 个记录对,其中,k 值由用户自己定义.

基于阈值的相似连接算法可分为两类[14]:第 1 类使用过滤技术过滤掉多余的候选对,例如 AllPair[1], PPJoin[2],EDJoin[10],QChunk[8],VChunk[11]和 AdaptJoin[4]等,也有使用其他的签名(signature),例如 PartEnum[18], PassJoin[9]和 FastSS[15];第 2 类算法利用特殊的树形索引来计算连接的结果,典型的有 TrieJoin[13].

在基于相似阈值的相似连接算法中,阈值需要用户自己定义.在实际应用中,相似度阈值往往与数据集特性相关,因此会出现面对新的数据集无法确定合适阈值的问题.此外,在一些应用中,用户有时关心的不是相似阈值有多大,而是记录集中哪几个记录对最相似.为此,文献[3]提出了一种 Top-k 相似连接算法 Top$k$-join.与基于阈值的相似连接算法相比,Top$k$-join 避免了因输入阈值太大而得到空的结果,或者输入阈值太小而运行时间过长的问题.

Top$k$-join 使用事件驱动(event driven)框架来查找数据集中最为相似的 k 对记录,算法中最为核心的概念是前缀事件(prefix event).前缀事件是一个三元组$\langle r,p_r,sp_r\rangle$,其中,$r$ 表示记录,p_r 表示记录中待处理 Token 的位置,sp_r 是该记录与数据集中所有其他记录相似度的最大值.Top$k$-join 使用临时结果堆保存当前最相似的 k 个记录对;使用前缀事件堆存储所有前缀事件,并每次弹出具有最大 sp_r 值的前缀事件进行处理.文献[3]证明:当程序结束时,临时结果堆中一定存储着最相似的前 k 个记录对.本文第 2 节将会对 Top$k$-join 做更为详细的介绍.

与基于阈值的相似连接算法 PPJoin[2]相比,在相似阈值未知的情况下,Top$k$-join 算法具有更好的性能.但是我们在研究中发现,Top$k$-join 存在如下性能问题.

(1) 为了减少重复计算,算法执行过程中生成的每一个候选记录对都需要查找哈希表,从而导致哈希查找代价过大;

(2) 每次前缀事件只处理记录的一个 Token,使得临时结果集逼近真实结果的速度较慢,同时造成前缀事件和临时结果堆的维护代价过大.

针对上述问题,本文提出了一种基于 Token 批处理的 Top-k 相似连接算法 Opt-join.新算法通过调换哈希查找与过滤操作位置以及对 Token 进行批处理操作,显著提高了 Top-k 相似对的查找速度.本文的主要贡献有:

- 理论证明了调换哈希查找和过滤操作位置能够保证算法正确性.以此为基础,在 Opt-join 中对候选记录对先进行过滤操作(位置过滤和后缀过滤),随后再进行哈希查找.在大量减少哈希查找数量的同时,还提高了过滤操作的过滤能力;
- 根据数据集特性,设计了有效选择 Token 批处理长度的方法.通过在事件驱动框架中集成 Token 批处理技术,加快了相似度阈值的迭代速度,并减小了前缀事件和临时结果堆维护的代价;
- 分别实现了 Opt-join 和 Top$k$-join 算法,并基于真实数据集对两种算法进行了全面的实验性能对比.实验结果显示:与 Top$k$-join 算法相比,Opt-join 的 Size 过滤和位置过滤的能力均得到了显著提高,堆处理的代价极大地降低,哈希查找的代价减小了 1 倍~2 倍,并最终取得了 1.28~3.09 的加速比.在数据长度增加或 k 值增加时,Opt-join 的性能优势会变得更加明显.

本文第 1 节介绍背景知识.第 2 节给出 Topk-join 算法的执行过程.第 3 节详细讨论本文提出的两个优化策略.第 4 节是实验和结果分析.最后,第 5 节总结全文.

1 问题定义和背景知识

1.1 问题的形式化定义

有限集 U 为 Token 的集合.O 为定义在 U 上的一个全序,按照 Token 的文档频率(document frequency)升序

排列.记录 r 为若干 Token 的集合且这些 Token 在 r 中按照 O 进行排列.给定两个记录 r 和 s,相似函数 $sim(r,s)$ 返回一个[0,1]区间的相似值,该值越大,表示记录 r 和 s 越相似.我们用$|r|$表示记录 r 中 Token 的个数.本文讨论的相似函数及计算公式见表 1.

Table 1 Similarity functions

表 1 相似函数

Similarity function	Definition
Jaccard	$sim_J(r,s)=\dfrac{\lvert r\cap s\rvert}{\lvert r\cup s\rvert}$
Cosine	$sim_C(r,s)=\dfrac{\lvert r\cap s\rvert}{\sqrt{\lvert r\rvert\cdot\lvert s\rvert}}$
Dice	$sim_D(r,s)=\dfrac{2\cdot\lvert r\cap s\rvert}{\lvert r\rvert+\lvert s\rvert}$
Overlap	$sim_O(r,s)=\lvert r\cap s\rvert$

表 1 中给出的 4 种相似度函数有其内在的联系[2].具体地,给定某种相似函数和相似度阈值θ,若 $sim(r,s)\geqslant\theta$,那么一定有 $O(r,s)\geqslant\partial$.其中,$O(r,s)$表示记录 r 和 s 的 Overlap 相似度,∂是通过相似函数计算的 Overlap 相似度阈值.表 2 给出了 Jaccard 相似度、余弦相似度和 Dice 相似度中 Overlap 相似度阈值的计算公式.如非明确说明,本文后面部分都以 Jaccard 相似度函数为例进行讨论.

Table 2 ∂ values in different similarity functions

表 2 不同相似度函数对应的∂值

Similarity function	Jaccard	Cosine	Dice
∂值	$\dfrac{\theta}{1+\theta}(\lvert r\rvert+\lvert s\rvert)$	$\theta\cdot\sqrt{\lvert r\rvert\cdot\lvert s\rvert}$	$\dfrac{\theta\cdot(\lvert r\rvert+\lvert s\rvert)}{2}$

不同于传统基于相似度阈值的连接操作,Top-k 相似连接定义如下.

定义 1(Top-k 相似连接). 给定两个记录集 $R=\{r_1,r_2,\ldots,r_n\}$和 $S=\{s_1,s_2,\ldots,s_m\}$、相似函数 sim 和用户定义的参数 k,Top-k 连接找出 $R\times S$ 中最相似的前 k 个记录对,即$\{(r,s)_i|(r,s)_i>(r,s)_j,(r,s)\in R\times S,i\in[1,k],j\notin[1,k]\}$.

与文献[3]类似,本文重点考虑 $R=S$ 的情况,即,同一数据集中记录项之间的自连接.

1.2 过滤技术

现有的相似连接算法使用过滤技术来减少需要精确计算相似度的候选记录对个数,目前常用的过滤技术如下所示.

(1) 前缀过滤(prefix filtering)

传统的相似连接将记录中所有 Token 插入到倒排索引(inverted index)中,通过扫描倒排索引生成候选记录对.这种方法的缺点是索引表过于庞大,同时,当记录集中某一个 Token 出现频率较高时,会生成大量重复的候选记录对.为解决上述问题,文献[6]在对记录集进行预处理(按 Token 的文档频率升序排列)的基础上,提出了前缀过滤算法.前缀过滤算法有效地解决了索引表过大和候选对大量重复的问题,并将插入倒排索引表的 Token 数由$|r|$减小到了$|r|-\partial+1$.

定理 1(前缀过滤)[6,16]. 给定记录 r 和 s,如果 $sim_O(r,s)\geqslant\partial$,那么记录 r 的$(|r|-\partial+1)$-前缀与记录 s 的$(|s|-\partial+1)$-前缀至少有一个公共的 Token 值.表 3 给出了不同相似函数下记录前缀的长度.

Table 3 Prefix lengths for different similarity functions

表 3 不同相似函数下的前缀长度

Similarity function	Jaccard	Cosine	Dice
Prefix length	$\lvert r\rvert-\lceil\theta\lvert r\rvert\rceil+1$	$\lvert r\rvert-\left\lceil\theta\cdot\sqrt{\lvert r\rvert}\right\rceil+1$	$\lvert r\rvert-\left\lceil\dfrac{\theta}{2-\theta}\cdot\lvert r\rvert\right\rceil+1$

(2) 长度过滤(size filtering)[2]

记录间的相似度与记录的长度存在一定的关系.如果两个记录比较相似,它们的长度也比较接近.例如,在Jaccard 相似度下,如果 $sim_J(r,s)\geqslant\theta$,那么记录 r 和 s 的长度关系见公式(1).

$$\theta\cdot|r|\leqslant|s|\leqslant\frac{|r|}{\theta} \tag{1}$$

表 4 给出了不同相似度函数下记录的长度范围.根据 r 和 s 的长度关系,可以过滤掉那些明显不满足条件的记录对.

Table 4 Record length ranges for different similarity functions

表 4 不同相似函数下的记录长度范围

Similarity function	Min len	Max len
Jaccard	$\theta\cdot\|r\|$	$\theta/\|r\|$
Cosine	$\theta^2\cdot\|r\|$	$\|r\|/\theta^2$
Dice	$(\theta\cdot\|r\|)/(2-\theta)$	$(2-\theta)\cdot\|r\|/\theta$
Overlap	–	–

(3) 位置过滤(positional filtering)

前缀过滤的前提是记录集中每条记录 Token 都是有序的.在前缀过滤的基础上,文献[2]利用 Token 在记录中的位置信息进一步减小候选对的数量.

定理 2(位置过滤)[2]. 给定两个 Token 按照全序 O 排列的记录 r 和 s,对于任意的 Token $e=r[i]=s[j]$,它将记录 r 分为前缀部分 $Prefix_r^i=r[1...(i-1)]$ 和后缀部分 $Suffix_r^i=r[i...|r|]$ (s 同理).如果 $sim_O(r,s)\geqslant\partial$,那么有:

$$sim_O(Prefix_r^i,Prefix_s^j)+\min(|Suffix_r^i|,|Suffix_s^j|)\geqslant\partial \tag{2}$$

(4) 后缀过滤(suffix filtering)

根据前缀定理,随着相似度阈值的增加,前缀长度逐渐减小,而后缀长度逐渐增大.位置过滤只针对记录的前缀,而对于记录后缀却缺少有效的利用.为了进一步减小候选对数量,文献[2]中提出了后缀过滤技术.

后缀过滤基于记录对的海明(Hamming)距离.任意记录对的海明距离可通过它们的 Overlap 相似度计算出来[2].具体来说,给定记录 r 和 s,如果 $sim_O(r,s)\geqslant\partial$,则 $dis_H(r,s)\leqslant|r|+|s|-2\partial$,其中 $dis_H(\cdot,\cdot)$计算两个记录的海明距离.记录 r 和 s 的最大海明距离($H_{\max}$)见公式(3).

$$H_{\max}=|r|+|s|-2\cdot\partial-(i+j-2\cdot sim_O(Prefix_r^i,Prefix_s^j)) \tag{3}$$

其中,∂是 Overlap 相似度阈值,i 和 j 分别表示记录 r 和记录 s 前缀中最后一个 Token 的位置,$sim_O(Prefix_r^i,Prefix_s^j)$表示前缀中的 Overlap 相似度值.

有了 r 和 s 的最大海明距离,后缀过滤通过一个递归算法估算 r 和 s 的实际海明距(H_c),具体算法见文献[2].当一个候选记录对的 $H_c>H_{\max}$ 时,我们就可以将该记录对丢弃.

2 Top*k*-join 算法

不同于传统的相似连接算法(例如 AllPair[1],PPJoin[2]),Top*k*-join 使用事件驱动框架来寻找记录集中最相似的前 k 个记录对[3],其中,核心的概念是前缀事件.前缀事件是一个三元组$\langle r,p_r,sp_r\rangle$,其中:$r$ 表示记录;p_r 表示下一个要插入到前缀中的 Token 的位置;sp_r 是记录 r 与数据集中其他记录相似度值的上边界,计算方法见公式(4).

$$sp_r=1-\frac{p_r-1}{|r|} \tag{4}$$

算法执行过程中的临时结果也是一个三元组$\langle r,s,\theta_{rs}\rangle$,其中,$r$ 和 s 均表示记录,θ_{rs} 表示记录 r 和 s 的相似度值.Top*k*-join 分别使用最大堆(E)和最小堆(T)保存算法执行过程中生成的前缀事件和临时结果.

Top*k*-join 的执行框架如下:

(1) 初始化前缀事件堆 E(将 sp_r=1 的所有前缀事件插入到最大堆中)和临时结果堆 T(任意 k 个记录对插入到最小堆中);

(2) 取出 E 的堆顶元素$\langle r,p_r,sp_r\rangle$.如果 sp_r 的值不小于 T 中堆顶元素$\langle r,s,\theta_{rs}\rangle$的相似值$\theta_{rs}$(即 $sp_r\geqslant\theta_{rs}$ 时),使用位置 p_r 上 Token 查找倒排索引(inverted index)以生成候选记录对;否则,整个程序结束,此时,临时结果堆中的记录对即为 Top-k 记录对;

(3) 计算新生成候选对的相似度 θ_{rs}^{+},如果该值大于 T 堆顶的相似值θ_{rs},即 $\theta_{rs}^{+}>\theta_{rs}$,则将该候选对和对应相似值插入到 T 中,同时弹出 T 中的最小值,以保证堆的大小为 k;

(4) 删除已处理完的前缀事件(T 的堆顶元素),生成相应记录的下一个前缀事件$\langle r,p_r+1,sp_r\rangle$并插入 T 中,继续执行步骤(2)和步骤(3).

算法 1 给出了 Top$k$-join 的伪代码.初始阶段,将所有 sp_r=1 的前缀事件插入到事件堆 E 中;选取记录集合的第 1 条记录和第 2–k+1 条形成 k 个记录对,计算相应的相似值,并将结果插入到临时结果堆 T 中(第 1 行).执行阶段,首先取出前缀事件堆的堆顶元素(第 3 行),比较上界值(sp_r)与临时结果堆的堆顶记录对相似度值,判断程序是否结束(第 5 行).对新生成的候选对,先查找其是否已存在于全局哈希表(H)中,对查找不到的候选对进行过滤操作(第 9 行、第 10 行).最后,对过滤不掉的候选对计算相似度值(第 11 行),并更新全局哈希表、前缀事件堆、临时结果堆和倒排索引(第 12 行、第 13 行和第 15 行).

算法 1. *Topk-join*(R,k).

输入:R 为所有记录的集合;k 表示要输出的相似对个数;

输出:最相似的前 k 个记录对.

1. Initialize prefix event heap E and result heap T.
2. WHILE $E\neq\varnothing$
3. $\langle r,p_r,sp_r\rangle\leftarrow E.pop(\cdot)$;
4. $sim_k=T[k].\theta_{rs}$;
5. IF $sp_r\leqslant sim_k$
6. break;
7. ENDIF
8. Generate candidate pairs (r,s) via index search;
9. IF $(r,s)\notin H$
10. Perform filtering (including size, positional and suffix filtering);
11. Compute the similarity of (r,s);
12. $H\leftarrow H\cup(r,s)$;
13. Update the inverted index and T;
14. Calculate new similarity bound sp_r;
15. $E.push(r,p_r+1,sp_r)$;
16. ENDIF
17. ENDWHILE

3 Top-k 相似连接算法 Opt-join

本节将讨论 Top$k$-join 中存在的两个重要性能缺陷,并提出哈希查找优化策略和前缀批处理技术作为解决方案.通过在事件处理框架中结合这两种方法,设计了一种新的相似连接算法 Opt-join.

3.1 优化哈希查找

3.1.1 哈希查找的性能问题及解决方法

在事件驱动框架中,两个记录的前缀中有几个重复 Token 值,这两个记录组成的候选对就会被重复生成相应的次数,从而导致同一个候选对可能会被验证多次.算法 1 中使用了一个全局哈希表(H)来避免候选对的重复

验证,主要设计思路是:对于生成的候选对,首先查找全局哈希表(算法 1 第 9 行),如果该候选对不在哈希表中,则进行过滤操作(算法 1 第 10 行).对没过滤掉的候选对,精确计算其相似度值(算法 1 第 11 行),并将其插入到哈希表中(算法 1 第 12 行).若该候选对已经存在于哈希表中,表示其已经被处理过,不再需要重复计算.

哈希表的使用解决了候选对重复验证的问题,但也不可避免地带来了哈希表查找的代价.通过记录哈希表的查找和插入过程,我们统计了 Top*k*-join 花费在哈希操作上的时间代价.图 1 给出了 TREC 数据集(详见实验部分)下程序总运行时间与哈希查找和插入时间的比较,可以看到,在哈希查找和插入上花费的时间占总时间的 20%以上.造成这一情况的原因是:虽然每次哈希查找代价不大,但每个新生成的候选对都需要查找哈希表,因此当候选对总数很大时,就会花费过多的时间.

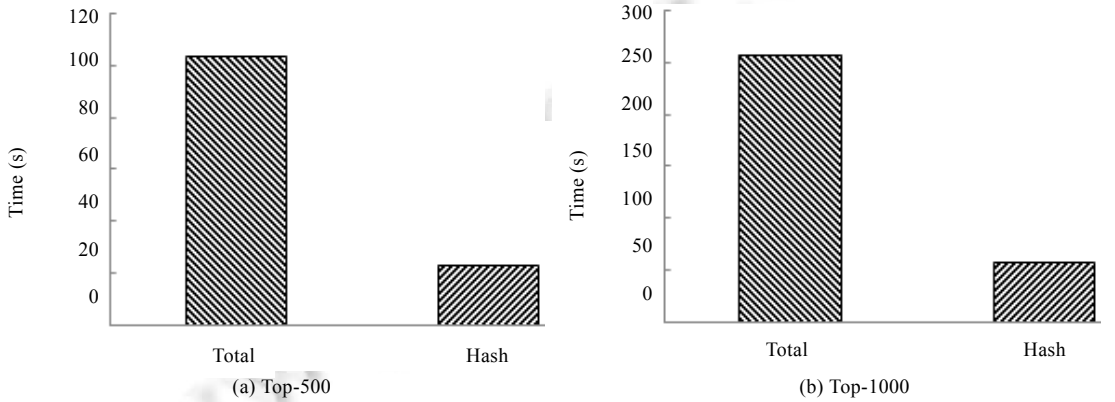


Fig.1 Comparison of hash operations and total run time

图 1 哈希操作花费时间与总运行时间的比较

以 top-500 查询为例,程序在运行过程中总共生成 2.15 亿个候选对,最终哈希表的大小为 247 万.在 2.15 亿个候选对中,被哈希表过滤掉的候选对(即重复验证的候选对)为 169 万个,只占到总数的 0.8%,而 99%的候选对被其他过滤操作(包括长度、位置和后缀过滤)过滤掉.如果这些被过滤掉的候选对不经过哈希查找,那么哈希表的代价就会极大地降低.

基于上述分析,我们将哈希查找推迟到各种过滤技术之后进行,见算法 2.在程序执行过程中,新生成的候选对先进行过滤,过滤不掉的候选对再进行全局哈希表的查找.这样不仅避免候选对重复计算,又减小了哈希代价.算法 2 用于替换算法 1 中的第 9 行~第 13 行.

算法 2. *Opt-hash*(*r*,*s*).

1. Perform filtering operations (including size, positional and suffix filtering);
2. IF $(r,s)\notin H$
3. Compute the similarity of (r,s);
4. $H\leftarrow H\cup(r,s)$;
5. Update the inverted index and T;
6. ENDIF

上述优化带来了一个新问题:在优化前,过滤操作只在候选对第 1 次出现公共 Token 时才会被调用,随后生成的重复候选对将会被全局哈希过滤掉,所以,过滤操作只执行 1 次.优化后,每个公共 Token 生成的候选对都要进行过滤操作(位置过滤使用公式(2),后缀过滤使用公式(3)).观察公式(2)和公式(3)可以发现,位置过滤和后缀过滤均依赖于候选对前缀的 Overlap 值.要精确计算 Overlap 值,需要每次扫描候选对的前缀,这样将会花费更大的代价.为此,我们考虑将候选对前缀的 Overlap 值($sim_O(Prefix_r^i, Prefix_s^j)$)置为 1,即,使用公式(5)来代替公式(2)、使用公式(6)代替公式(3)来进行过滤,避免对候选对前缀的重复扫描.

$$1+\min(|Suffix_r^i|,|Suffix_s^j|)\geqslant\partial \tag{5}$$

$$H_{\max}=|r|+|s|-2\cdot\partial-(i+j-2) \tag{6}$$

容易看出:当前缀中 Overlap 值为 1 时,使用公式(5)和公式(6)过滤算法均能正确执行.而当前缀 Overlap 值大于 1 时,过滤算法的正确性则有待证明,下面我们将讨论这一问题.

3.1.2 正确性证明

Top$k$-join 算法使用 Size 过滤、位置过滤和后缀过滤来减少候选对的数量.Size 过滤利用记录对的长度关系完成过滤,因此,第 3.1.1 节的优化对 Size 过滤没有影响.下面我们只需要证明位置过滤和后缀过滤在优化后的正确性.

观察 1. 在对候选记录对进行位置过滤的过程中,公式(5)与公式(2)的正确性相同,且公式(5)具有更强的过滤能力.

证明:容易看出:当第 1 次生成候选对时,记录对前缀的 Overlap 相似度为 1.

即,公式(2)中的 $sim_O(Prefix_r^i,Prefix_s^j)=1$.此时,公式(5)与公式(2)等价.

当一个候选对第$\xi(\xi>1)$次出现时,候选对前缀的 Overlap 相似度为ξ,即 $sim_O(Prefix_r^i,Prefix_s^j)=\xi$.使用公式(2),当 $\xi+\min(|Suffix_r^i|,|Suffix_s^j|)<\partial$ 时,候选对才会被过滤掉;而使用公式(5),当 $1+\min(|Suffix_r^i|,|Suffix_s^j|)<\partial$ 时,候选对就会被过滤掉.因为恒大于 $1+\min(|Suffix_r^i|,|Suffix_s^j|)$,即:

$$1+\min(|Suffix_r^i|,|Suffix_s^j|)<\xi+\min(|Suffix_r^i|,|Suffix_s^j|)<\partial.$$

所以当$\xi>1$,公式(5)与公式(2)的正确性相同,且有更强的过滤能力.证毕. □

观察 2. 在对候选对进行后缀过滤过程中,公式(6)与公式(3)的正确性相同,且公式(6)具有更强的过滤能力.

证明:与位置过滤类似,当第 1 次生成候选对时,前缀中的 Overlap 相似度为 1.

即,公式(3)中的 $sim_O(Prefix_r^i,Prefix_s^j)=1$.此时,公式(6)与公式(3)等价.

当一个候选对第$\xi(\xi>1)$次出现时,候选对前缀的 Overlap 相似度为ξ,即 $sim_O(Prefix_r^i,Prefix_s^j)=\xi$.使用公式(3),当估算海明距 $H_c\geqslant|r|+|s|-2\cdot\partial-(i+j-2\cdot\xi)$时,候选对才会被过滤掉;而使用公式(6),当 $H_c\geqslant|r|+|s|-2\cdot\partial-(i+j-2)$时,候选对就会被过滤掉.因为$|r|+|s|-2\cdot\partial-(i+j-2\cdot\xi)$恒大于$|r|+|s|-2\cdot\partial-(i+j-2)$,即 $H_c\geqslant|r|+|s|-2\cdot\partial-(i+j-2\cdot\xi)\geqslant|r|+|s|-2\cdot\partial-(i+j-2)$,所以当$\xi>1$ 时,公式(6)与公式(3)的正确性相同,且有更强的过滤能力.证毕. □

3.2 Token批处理

3.2.1 事件驱动框架存在的问题

事件驱动框架主要执行过程是:对于前缀事件$\langle r,p_r,sp_r\rangle$,将记录 r 插入到 p_r 对应 Token 的倒排列表中,扫描该倒排列表生成候选对,随后再进行过滤和精确的相似度计算.事件驱动框架保证最后结果的正确性,但是它存在的问题有:

(1) 前缀事件堆的维护代价比较高;

(2) 每次只处理记录的一个 Token,限制了临时结果堆顶候选对相似度 sim_k 的增长速度;

(3) 存放候选对的哈希表太大,需花费大量的建立和查找代价.

如果第 k 个最相似记录对的相似度值θ_k已知,那么传统相似度连接算法(PPJoin 或 PPJoin+)的效率明显优于 Top$k$-join 算法.通过比较我们发现,PPJoin 性能良好的原因在于:

(1) PPJoin 将记录前缀中的 Token 全部批量的插入到索引表中;

(2) PPJoin 在处理每一条记录时,都利用一个较小的哈希表过滤重复的候选对;

(3) PPJoin 生成的候选对数量明显少于 Top$k$-join 算法.

上述分析表明,每次只处理一个 Token 是造成事件驱动框架效率低下的主要原因.为此,我们考虑借鉴传统相似连接算法(PPJoin)中 Token 批处理的思想,对事件驱动框架进行改进.具体来说,当一个前缀事件到达之后,我们不只处理一个 Token,而是同时处理多个 Token,即,下一个前缀事件由$\langle r,p_r+1,sp_r\rangle$转变为$\langle r,p_r+MultiLen,sp_r\rangle$,其中,$MutiLen$ 为不小于 1 的正整数.在 Token 批处理中,$MultiLen$ 的选取成为一个关键性的问题,下面我们将对

其进行详细的讨论与分析.

3.2.2 *MultiLen* 的选取

如果第 k 个最相似记录对的相似度值 θ_k 已知,根据前缀定理,*MultiLen* 的值为

$$MultiLen=\lfloor |r|\cdot(1-\theta_k)\rfloor+1 \tag{7}$$

但是,θ_k 在没有完成查询处理之前是无法获得的.在 Top$k$-join 算法执行过程中我们发现:当处理完上界值为 1 的前缀事件(即 p_r=1,sp_r=1)后,临时结果堆顶的候选对相似度值 sim_k 与 θ_k 相当接近.表 5 给出了 TREC 数据集上执行 Top200-1000 查询时这两个相似度的值.基于上述观察,我们考虑利用 sim_k 代替公式(7)中的 θ_k 求解 *MultiLen* 的值.

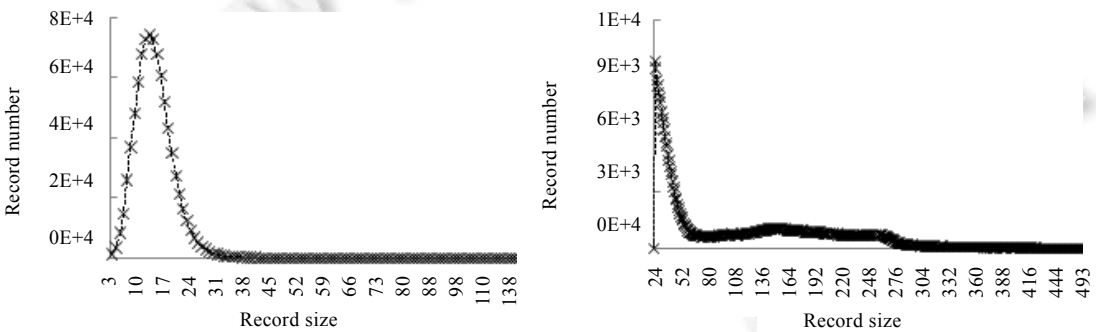
Table 5 Comparison between sim_k and θ_k

表 5 sim_k 与 θ_k 的比较

Top-k	200	400	600	800	1 000
sim_k	0.935	0.8	0.75	0.719	0.7
θ_k	0.938	0.821	0.781	0.761	0.75

直接用 sim_k 代替 θ_k,带来的问题是每条记录的前缀都将会被扩大.以 TREC 数据集上 top-1000 查询为例,如果某一条记录的长度为 100,那么它的前缀将由 70 扩大到 75.前缀的扩张增加了过滤和哈希的代价,同时也增加了候选对的数量,所以需要对公式(7)中的长度$|r|$进行修正.

在分析数据特性时我们发现,很多数据集在预处理之后记录的长度会集中在某一个区间.图 2 给出了 DBLP 和 TREC 数据集的长度分布图.



(a) DBLP (b) TREC

Fig.2 Length distribution of datasets

图 2 数据集长度分布

可以看到,DBLP 中记录的长度主要集中在[10,23],TREC 中记录的长度主要集中在[24,46].如果我们选择频率最高的记录长度(即区间中的 record number 最大的记录长度,记为 L_m)来代替公式(7)中的$|r|$,见公式(8),那么会带来如下好处:

(1) 会使长度小于 L_m 的前缀能够一次性执行完成,临时结果堆顶相似度 sim_k 迭代加快;

(2) 对于长记录的前缀也不会扩得太大.

$$MultiLen=\lfloor (1-sim_k)\cdot L_m\rfloor+1 \tag{8}$$

同样以 TREC 数据集上 Top-1000 查询为例(L_m=25),如果某一条记录的长度为 100,根据公式(8)计算的前缀长度为 8,小于该记录的实际前缀长度 26.

虽然选取 L_m 具有上述优点,但是仍然有以下两个问题需要解决.

(1) 对于短记录的处理

对于长度远小于 L_m 的记录来说,由于 sim_k 与 θ_k 非常接近,如果仍然按照公式(8)计算 *MultiLen*,生成的前缀长度将会明显大于记录的实际前缀长度,有的甚至超过记录本身.以 DBLP 数据集中长度最小的记录为例,其中最

小长度 $L_{\min}=3$,$L_m=12$,当查询 Top-20000 时,$\theta_k=0.680$,根据公式(8)计算出来的 $MultiLen\geqslant 4>L_{\min}$.因此,有必要限制 $MultiLen$ 值的大小.我们知道,前缀事件$\langle r,p_r,sp_r\rangle$中的 p_r 为前缀中下一个要处理 Token 的位置.如果临时结果堆顶的候选对相似度值为 sim_k,根据前缀定理,所有未处理前缀长度的最大值为

$$|r|-p_r-sim_k\cdot|r|+1 \tag{9}$$

将公式(9)带入到公式(4)中,对于每一条记录,我们定义了 $MultiLen_{\max}$ 来限定 $MultiLen$ 的取值,如公式(10)所示.

$$MultiLen_{\max}=\lfloor(sp_r-sim_k)\cdot|r|\rfloor+1 \tag{10}$$

具体来说,最终的 Token 批处理长度为 $MultiLen_{ult}=\mathrm{MIN}(MultiLen,MultiLen_{\max})$.

仍以数据集 DBLP 为例,利用公式(10),在 Top-20000 中最短记录的 $MultiLen_{ult}$ 为 1 而不再是 4,从而解决了短记录 $MultiLen$ 过长的问题.

(2) 对于长记录的处理

对于长度大于 L_m 的记录,若每一次只处理 $MutiLen$ 个 Token,则有时需要多次才能执行完成.以 TREC 数据集为例,最长的记录为 $L_{\max}=609$.当 $k=400$,$\theta_k=0.8$ 时,利用公式(8)计算的 $MultiLen$ 值为 10.根据前缀定理该记录的实际前缀长度为 122,如果每次只处理 10 个 Token,那么要 13 次才能完成处理,从而增加了算法运行时间.为了减小长记录的计算次数,我们对 $MultiLen$ 进行扩展,见公式(11).

$$MultiLen=[(1-sim_k)\cdot L_m+1]\cdot count^3 \tag{11}$$

其中,$count$ 为该记录目前的处理批次.当第 1 次处理该记录的前缀事件时,公式(11)得到的 $Multilen$ 值与公式(8)相同.对于需要多批次处理的长记录,$Multilen$ 相对 $count$ 的指数增长使得少量的批次就能完成该记录的处理,从而提升算法性能.例如,通过 $MultiLen$ 扩展,TREC 数据集中的最长记录第 1 次处理 10 个 Token,第 2 次处理 80 个 Token,最后一次处理剩余的 32 个 Token,因此只需通过 3 次就可执行完成.

3.2.3 Opt-join 算法

通过在 Top$k$-join 中集成 Token 批处理技术和哈希优化策略,新的 Opt-join 见算法 3.Opt-join 首先调用 $Bach\text{-}one$(算法 4)处理 $sp_r=1$ 的所有前缀事件,并为每条记录设定一个计数器,计算记录执行的次数(算法 4 第 6 行);随后,使用公式(10)和公式(11)分别计算 $MultiLen$ 和 $MultiLen_{\max}$,将两者中最小值作为该记录批次的 Token 长度 $MultiLen_{ult}$(第 9 行、第 10 行);前缀事件的处理不再是只插入一个 Token 到倒排列表中,而是插入 $MultiLen_{ult}$ 个 Token(第 10 行~第 17 行),这样就加快了临时结果堆顶 sim_k 的迭代速度;最后,更新记录执行次数,计算新前缀事件的上界值,并将新前缀事件插入到堆中(第 18 行、第 19 行).

算法 3. $Opt\text{-}join(R,k)$.

输入:R 为所有记录的集合;k 表示要输出的相似对的个数;

输出:最相似的前 k 个记录对.

1. Initialize prefix event heap E and result heap T.
2. $Bach\text{-}one(\cdot)$;
3. WHILE $E\neq\varnothing$
4. $\langle r,p_r,sp_r\rangle\leftarrow E.pop(\cdot)$;
5. $sim_k=T[k].sim$;
6. IF $sp_r\leqslant sim_k$
7. break;
8. ENDIF
9. Calculate $Multilen$ and $MultiLen_{\max}$;
10. $MultiLen_{ult}\leftarrow\mathrm{MIN}(MultiLen,MultiLen_{\max})$;
11. FOR $pos=p_r$ to $p_r+MultiLen_{ult}$
12. Generate candidate pairs (r,s);

13. IF $(r,s)\notin H_0$

14. *Opt-hash*(r,s);

15. $H_0\leftarrow H_0\cup(r,s)$;

16. ENDIF

17. ENDFOR

18. $count_r\leftarrow count_r+1$;

19. *E.push*$(r,p_r+MultiLen_{ult},sp_r)$ and free H_0;

20. ENDWHILE

在 Opt-join 算法中,我们调用算法 2(*Opt-hash*(r,s))来对候选对进行过滤(第 14 行).同时,为了防止候选对重复计算,在候选对生成之后,增加了一个小哈希表 H_0(第 13 行).在 p_r 到 $p_r+MultiLen_{ult}$ 之间生成的所有候选对,都要通过 H_0 过滤.$MultiLen_{ult}$ 个 Token 处理完成后,释放小哈希表 H_0(第 19 行).

算法 4 负责处理所有 sp_r=1 的前缀事件.前缀事件堆 E 的初始化阶段是将所有 sp_r=1 的前缀事件插入到堆中(详见第 2 节),此时,堆的大小为$|R|$.在对所有$|R|$个前缀事件处理地同时,为每条记录初始化一个计数器(第 6 行),记录当前记录处理的次数.

算法 4. *Bach-one*(·).

1. FOR i=0 TO $|R|$

2. $\langle r,p_r,sp_r\rangle\leftarrow E.pop(\cdot)$;

3. Generate candidate pairs (r,s) via index search;

4. *Opt-hash*(r,s);

5. Update the inverted index and T;

6. $count_r\leftarrow 0$; //Intialize the record execution times

7. *E.push*(r,p_r+1,sp_r);

8. ENDFOR

在 Token 批处理过程中,由于 sim_k 与 θ_{rs} 之间的差值,不可避免地增加了某些记录前缀的长度,从而相应增加了生成的候选对数量.但是这一策略加快了 sim_k 的迭代速度,提高了过滤技术的过滤能力(主要是 Size 过滤和位置过滤),实际上是减小了需要验证的候选对的数量,这在实验的结果中会有所体现.

4 实验评估

本节对 Opt-join 和 Top*k*-join 进行了实验比较.实验环境为:Dell T320,1.9GHz Xeon(R) E5-2420 六核处理器,16G 内存和 1T 磁盘;操作系统为 Ubuntu 12.04;程序设计语言为 C++,编译器为 gcc-4.5.

为了能更好地看出哈希优化策略和 Token 批处理技术带来的性能提升,实验中比较了如下 3 种算法.

(1) Top*k*-join:文献[3]中提出的 Top-*k* 相似连接算法;

(2) 不使用 Token 批处理的 Opt-join(OPT1):该算法在 Top*k*-join 算法的基础上,只使用了哈希优化策略;

(3) Opt-join:该算法在事件驱动框架基础上,结合了 Token 批处理技术,且包含了 OPT1 的优化技术.

4.1 实验数据

与文献[2,3]类似,我们使用 DBLP 和 TREC 数据集来进行实验.DBLP 包含了 0.9M 的记录(作者+标题);TREC 选自于 MEDLINE 数据库,我们从中抽取 0.34M 作为实验数据.预处理之后,数据集的统计信息见表 6.与文献[3]类似,我们使用 Jaccard 和 Cosine 相似度函数衡量记录对的相似度.

Table 6 Dataset statistics

表 6 数据集统计信息

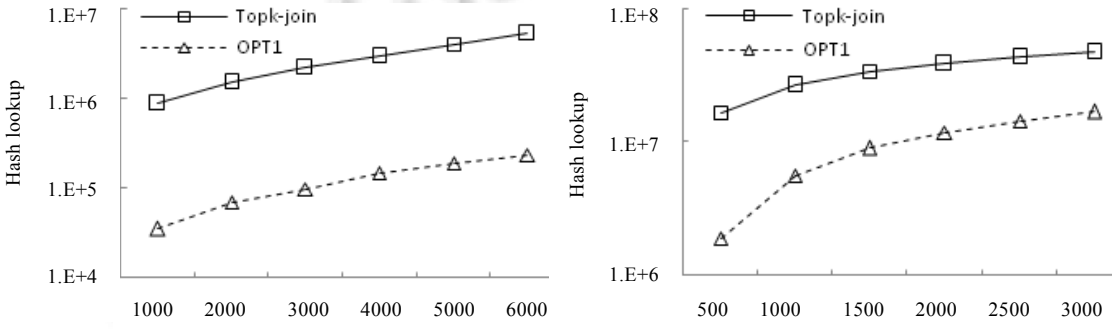
Dataset	*N*	Min.Len	Max.Len	Avg.Len
DBLP	861 567	3	284	14.3
TREC	345 969	24	609	114.4

4.2 实验结果分析

下面我们将从哈希代价、候选对过滤能力、前缀事件堆大小、候选对数量以及运行时间 5 个方面对上述 3 种算法进行比较.受篇幅限制,我们只列出了部分代表性实验结果.

(1) 全局哈希表代价对比

哈希表的代价主要来自于哈希查找.图 3 展示了在 Top*k*-join 算法和 OPT1 算法中,哈希查找次数的对比.*x* 轴表示 *k* 值,*y* 轴表示哈希表查找的次数.可以发现:相对于 Top*k*-join,OPT1 的哈希查找次数明显减少.这是因为在 Top*k*-join 算法中,所有的候选对都要进行哈希查找,只有查找失败的候选对才进行过滤(位置过滤和后缀过滤);而在 OPT1 算法中,先做位置过滤和后缀过滤,然后才进行哈希查找,大量不符合条件的候选对已经被过滤掉了.所以,OPT1 算法的哈希代价会远小于 Top*k*-join.



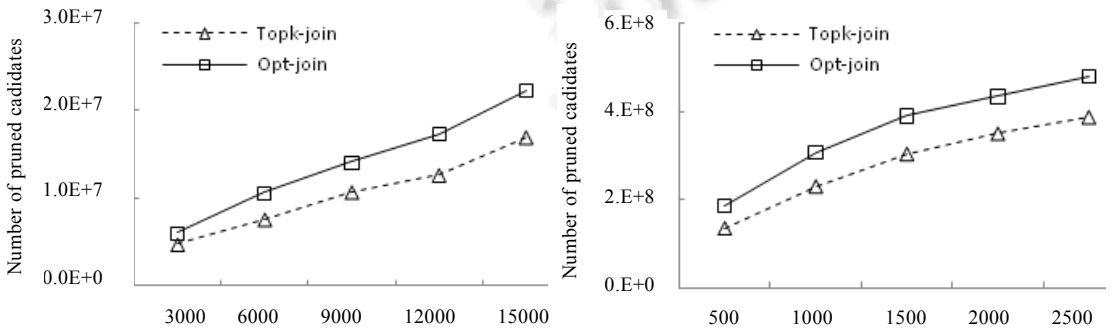
(a) DBLP, Cosine (b) TREC, Jaccard

Fig.3 Cost of hash lookup

图 3 哈希表查找代价

(2) 候选对过滤能力

图 4 和图 5 分别比较了 Top*k*-join 和 Opt-join 算法 Size 过滤和位置过滤的过滤能力,其中,*x* 轴表示 *k* 值,*y* 轴表示被过滤掉的候选对个数.



(a) DBLP, Cosine (b) TREC, Jaccard

Fig.4 Comparison of size filtering capability

图 4 Size 过滤能力的对比

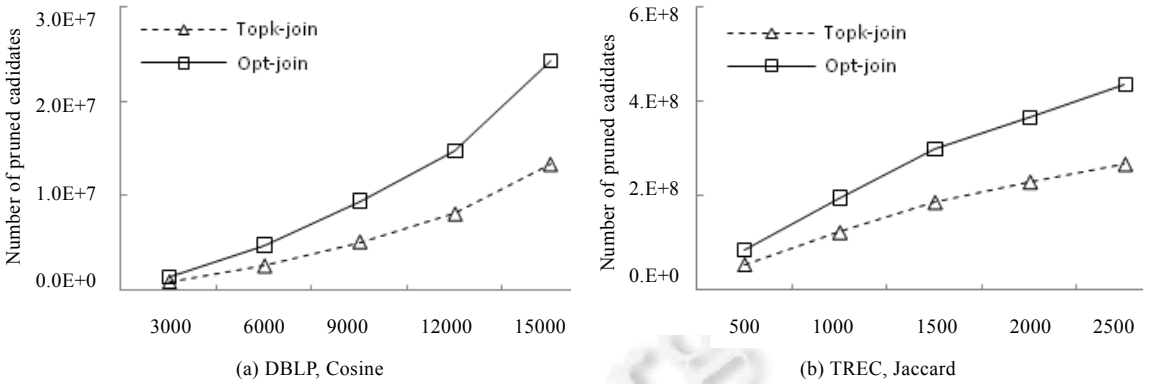


Fig.5 Comparison of positionalfiltering capability

图 5 位置过滤能力的对比

可以发现:在 Opt-join 算法中,Size 过滤和位置过滤的能力均得到了加强.这是由于在 Opt-join 算法中,Token 批处理使得临时结果堆顶的候选对相似度 sim_k 迭代加快.由公式(1)可知,Size 过滤能力主要依赖于 sim_k.sim_k 增长越快,Size 过滤的过滤区间就越小,其过滤能力就越强.所以,sim_k 的快速增长,使得 Size 过滤能力得到加强.

位置过滤能力增强的原因有:

1) 公式(2)和公式(5)中,∂值越大,位置过滤的过滤能力越强.见表 2 中,∂值的大小依赖于 sim_k.因此, sim_k 的迭代加快导致阈值∂迅速增大,从而使得位置过滤能力得到加强;

2) 定理 3 中,使用公式(5)代替公式(2)执行位置过滤后,位置过滤能力得到加强.

由定理 4 可知:使用公式(6)代替公式(3)后,后缀过滤能力将得到加强.但是,由于大量不符合条件的候选对已被 Size 过滤和位置过滤剔除掉,致使后缀过滤能力增加的不明显.以 TREC 数据集 Top-500 查询为例,在 Top*k*-join 算法中,后缀过滤剔除的不符合条件的候选对为 25 662 651;而在 Opt-join 中,剔除的不符合条件的候选对为 26 868 795,后缀过滤的能力只提高了 4.7%左右.

(3) 前缀事件个数

图 6 比较了 Top*k*-join 和 Opt-join 中前缀事件的个数.在 Top*k*-join 中,一个前缀事件只对应一个 Token;而使用了 Token 批处理技术后,一个前缀事件对应于多个 Token.在记录前缀不变的情况下,使用 Opt-join 算法时前缀事件明显减少,这就意味着前缀事件堆护所需的代价也相应降低.

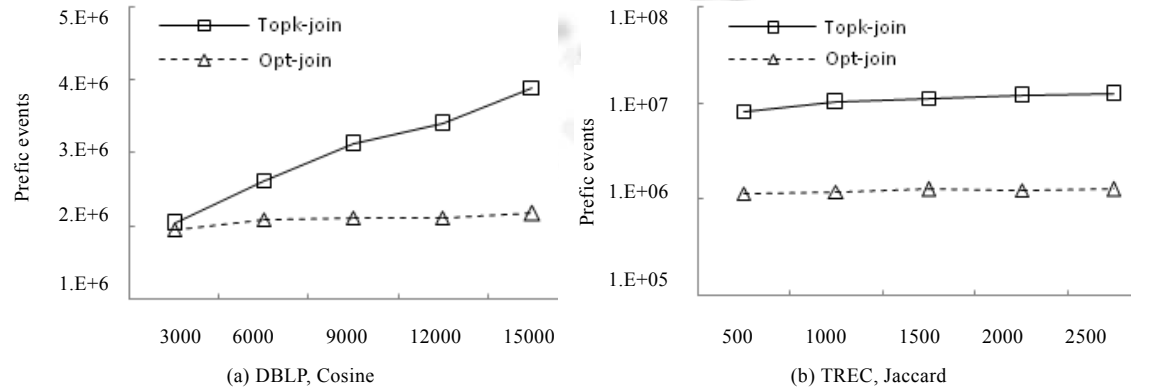


Fig.6 Comparison of the number of prefix events

图 6 前缀事件个数的比较

从图 7 中可以看出:TREC 数据集中,前缀事件减少了 10 倍左右;而在 DBLP 数据集中,前缀事件数目的减少

不如 TREC.主要原因是 DBLP 中记录长度太短,记录的相似度太高,导致计算出的 *MultiLen* 值太小.

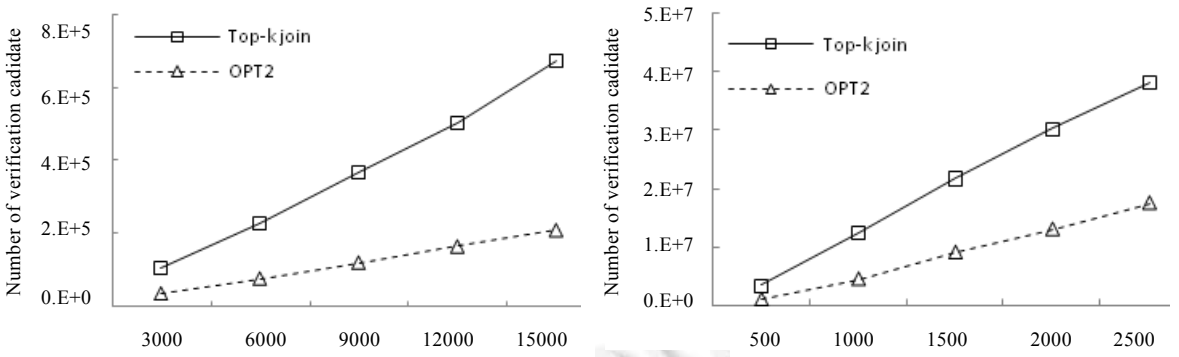


Fig.7 Comparison of the number of verificationpairs

图 7 验证对的比较

(4) 验证候选对数量对比

图 7 给出了 Top*k*-join 和 Opt-join 算法中,验证候选对的数量.验证候选对指通过过滤技术和哈希查找后,需要精确计算相似度的候选对.精确计算是算法代价的主要来源之一,验证候选对越少,精确计算的代价就越低.通过图 7 可以看出:Opt-join 算法使用 Token 批处理技术后,需要验证的候选对数量明显小于 Top*k*-join 算法(DBLP 减少了约 3 倍,TREC 约 2 倍).这是由于 Token 批处理技术使得 sim_k 的迭代加快(更快地达到实际的阈值),从而增强了候选对过滤能力(如图 4 和图 5 所示),使得需要精确计算相似度的候选对数量极大地减少.

(5) 程序运行时间

图 8 给出了 Top-k join,OPT1 和 Opt-join 算法程序运行时间的对比.为了方便比较,我们定义加速比为 Opt-join(OPT1)和 Top*k*-join 时间的比值.可以发现:OPT1 算法明显优于 Top*k*-join,主要原因是 Opt-join 算法通过变更哈希查找和过滤操作的位置,使得哈希表查找代价显著减小.Opt-join 算法优于 OPT1,主要是 Opt-join 在 OPT1 的基础上,又使用了 Token 批处理技术,使其在候选对过滤能力、前缀事件个数、候选对数量上都优于 OPT1(如实验 1~实验 4 所示).如图 8(a)、图 8(c)和图 8(d)所示,OPT1 比 Top*k*-join 算法的运行时间减少了 20%左右.在图 8(b)中,OPT1 的性能只提高了 8%,其原因是在相同 k 值下,Cosine 相似函数对应的相似阈值高于 Jaccard(例如在 k=500 时,Cosine 相似函数下的阈值为 0.96,Jaccard 相似函数下的阈值为 0.92),所以使用 Cosine 相似函数时生成的候选对个数远小于 Jaccard 下的个数(k=500 时,Cosine 相似函数下的候选对个数为 140 万,Jaccard 相似函数下的候选对个数为 253 万).候选对越少,Top*k*-join 中哈希代价越低,OPT1 性能提高的就越少.可以发现:随着 k 值增加,生成的候选对变得越多,OPT1 的优化也就越明显.

Opt-join 在 OPT1 基础上增加了 Token 批处理,与 Top*k*-join 相比,Opt-join 显著地提高了算法的性能.如图 8(a)所示,Opt-join 取得了 1.28~1.46 加速比.随着 k 的不断增加,加速比有不断增大的趋势.在图 8(b)中,Opt-join 的加速比是 1.65~1.69,这是由于相同 k 值下,Cosine 相似函数对应阈值高于 Jaccard 下的阈值,所以记录前缀长度小于 Jaccard 下相应的长度,这就导致 Token 批处理的次数减小,从而减少了程序运行时间.在 TREC 数据集下,Opt-join 在 Jaccard 和 Cosine 相似度函数下的加速比分别为 2.61~3.09 和 1.59~2.27,如图 8(c)、图 8(d)所示.这是由于 TREC 数据集中的记录长度比较长,当 k 相同时,Cosine 相似函数下的 sim_k 值大于 Jaccard 相似函数下的值.由公式(7)可知:Cosine 相似函数下,*MultiLen* 的长度会小于 Jaccard 相似函数下相应的值.根据公式(11),后续 *MultiLen* 的增长在 Jaccard 相似函数下的值会更大,因此,其批处理次数会减少,相应的加速比较 Cosine 要更好.需要注意的是:在 DBLP 数据集中,记录的长度较短,sim_k 的值比较大,导致计算出的 *MultiLen* 变化较小,所以起决定作用的是前缀的长度.

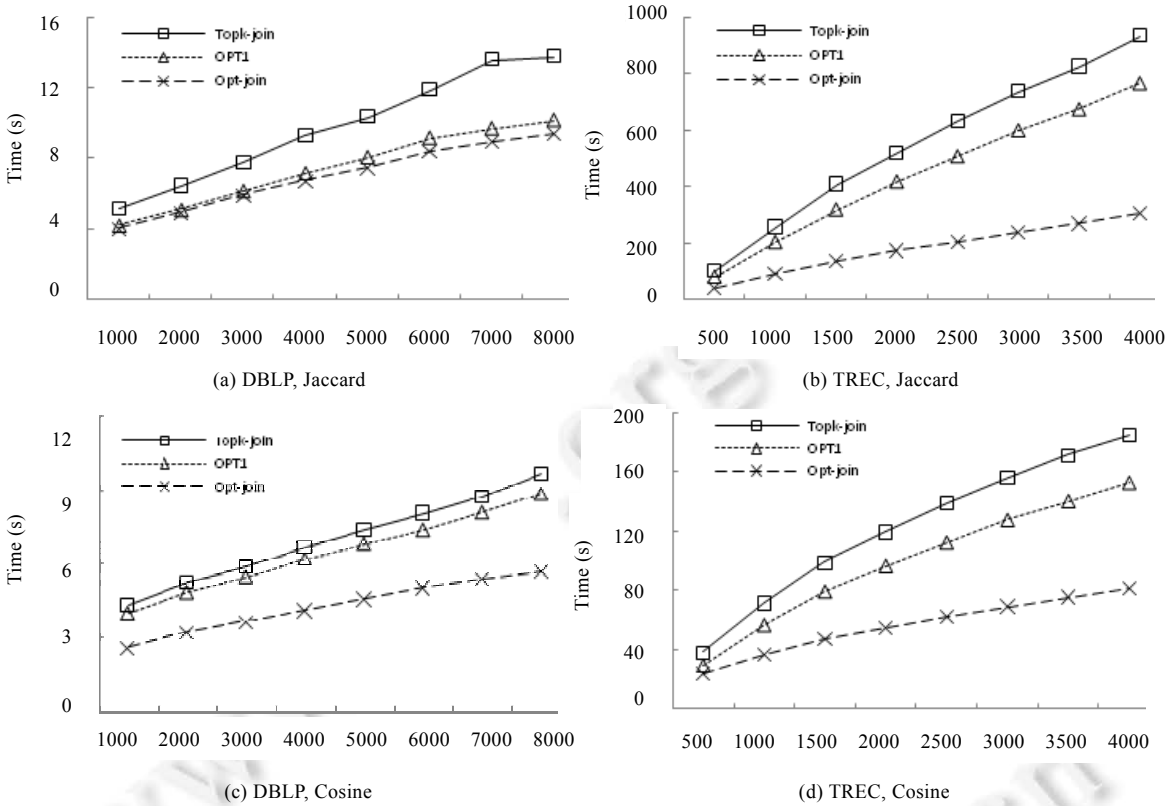


Fig.8 Running time

图 8 程序运行时间

总之,在使用了两个优化技术之后,Opt-join 可以取得 1.28~3.09 的加速比,且随着 k 值的增加或数据集中记录长度的增长,加速比有不断增大的趋势.

5 相关工作

相似连接算法在近几年已经获得广泛的关注.目前的相似连接算法可以归纳为两大类:基于阈值的相似连接和 Top-k 相似连接.

- 基于阈值的相似连接算法已经获得广泛的研究[1,2,4-9,10–23].文献[7]首先提出了前缀过滤算法;文献[1]利用倒排索引和各种过滤技术,对 Cosine 相似度下的相似连接算法进行优化;文献[2]在前缀过滤的基础上提出了位置过滤和后缀过滤,大大降低了需要验证的候选对的数量;文献[11,24]将相似连接算法应用与 Map-Reduce 中,可以处理 P 级以上的大数据集;
- Top-k 相似连接返回记录集中最相似的前 k 个记录对,主要应用于相似度阈值未知的情况[3,10,25].文献[3]中首次使用基于事件驱动的框架来处理 Top-k 相似连接算法;文献[10]使用编辑距(edit-distance)来计算 Top-k 的相似连接;文献[25]使用 Map-Reduce 框架来支持 Top-k 相似连接算法.

6 结 语

本文对 Top-k 相似连接算法进行了研究.在 Top$k$-join 算法的基础上,提出了哈希优化策略和 Token 批处理技术,并将这两个技术与前缀事件处理框架结合,设计了新的 Top-k 相似连接算法 Opt-join.

实验结果表明:Opt-join 在哈希代价、候选对过滤能力、前缀事件堆大小、候选对数量以及运行时间等 5

个方面都优于 Top$k$-join 算法,取得了 1.28~3.09 的加速比.随着 k 值的增加或数据集中记录长度的增长,加速比有不断增大的趋势.

References:

[1] Bayardo RJ, Ma Y, Srikant R. Scaling up all pairssimilarity search. In: Proc. of the 16th Int'l Conf. on World Wide Web. 2007. 131−140. [doi: 10.1145/1242572.1242591]

[2] Xiao C, Wang W, Lin X, Yu JX. Efficient similarityjoins for near duplicate detection. In: Proc. of the 17th Int'l Conf. on World Wide Web. 2008. 131−140. [doi: 10.1145/2000824.2000825]

[3] Xiao C, Wang W, Lin X, Shang H. Top-k setsimilarityjoins. In: Proc. of the 25th Int'l Conf. on Data Engineering (ICDE). 2009. 916−927. [doi: 10.1109/ICDE.2009.111]

[4] Wang J, Li G, Feng J. Can we beat the prefix filtering: An adaptive framework for similarity join and search. In: Proc. of the SIGMOD Conf. 2012. 85−96. [doi: 10.1145/2213836.2213847]

[5] Arawagi S, Kirpal A. Efficient set joins onsimilarity predicates. In: Proc. of the SIGMOD Conf. 2004. 743−754. [doi: 10.1145/1007568.1007652]

[6] Deng D, Li G, Feng J. A pivotal prefix based filteringalgorithm for string similarity search. In: Proc. of the SIGMOD Conf. 2014. 673−684. [doi: 10.1145/2588555.2593675]

[7] Chaudhuri S, Ganti V, Kaushik R. A primitive operator for similarity joins in data cleaning. In: Proc. of the Int'l Conf. on Data Engineering (ICDE). 2006. [doi: 10.1109/ICDE.2006.9]

[8] Li C, Wang B, Yang X. Vgram: Improving performanceof approximate queries on string collections usingvariable-length grams. In: Proc. of the VLDB. 2007. 303−314.

[9] Qin J, Wang W, Lu Y, Xiao C, Lin X. Efficient exactedit similarity query processing with the asymmetric signaturescheme. In: Proc. of the SIGMOD Conf. 2011. 1033−1044. [doi: 10.1145/1989323.1989431]

[10] Deng D, Li G, Feng J, Li WS. Top-k string similaritysearch with edit-distance constraints. In: Proc. of the Int'l Conf. on Data Engineering (ICDE). 2013. 925−936. [doi: 10.1109/ICDE.2013.6544886]

[11] Deng D, Li G, Hao S, Wang J, Feng J. Massjoin: Amapreduce-Based method for scalable string similarity joins. In: Proc. of the Int'l Conf. on Data Engineering (ICDE). 2014. 340−351. [doi: 10.1109/ICDE.2014.6816663]

[12] Kusumoto M, Maehara T, Kawarabayashi K. Scalable similarity search for SimRank. In: Proc. of the SIGMOD Conf. 2014. 325−336. [doi: 10.1145/2588555.2610526]

[13] Li G, He J, Deng D, Li J. Efficient similarity join and search on multi-attribute data. In: Proc. of the SIGMOD Conf. 2015. 1137−1151[doi: 10.1145/2723372.2723733]

[14] Li G, Deng D, Wang J, Feng J. Pass-Join: Apartition-Based method for similarity joins. PVLDB, 2011,5(3):253−264. [doi: 10.1145/2588555.2610526]

[15] Xiao C, Wang W, Lin X. Ed-Join: An efficient algorithmfor similarity joins with edit distance constraints. PVLDB, 2008,1(1): 933−944. [doi: 10.14778/1453856.1453957]

[16] Wang W, Qin J, Xiao C, Lin X, Shen HT. Vchunkjoin: An efficient algorithm for edit similarity joins. IEEE Trans. on Knowlogy Data Engineer, 2013,25(8):1916−1929. [doi: 10.1109/TKDE.2012.79]

[17] Sarawagi S, Bhamidipaty A. Interactive deduplication using activelearning. In: Proc. of the KDD. 2002. 269−278. [doi: 10.1145/775047.775087]

[18] Wang J, Li G, Feng J. Trie-Join: Efficient trie-basedstring similarity joins with edit-distance constraints. PVLDB, 2010,3(1): 1219−1230. [doi: 10.14778/1920841.1920992]

[19] Jiang Y, Li G, Feng J. String similarity joins: An experimental evaluation. In: Proc. of the VLDB. 2014. 625−636. [doi: 10.14778/2732296.2732299]

[20] Bocek BST, Hunt E. Fast similarity search in large dictionaries. Technical Report, ifi-2007.02, Department of Informatics, University of Zurich, 2007.

[21] Lam HT, Dung DV, Perego R, Silvestri F. An incremental prefix filtering approach for theall pairs similarity search problem. In: Proc. of the 12th Asia Pacific Web (APWEB). 2010. 188−194. [doi: 10.1109/APWeb.2010.30]

[22] Lin XM, Wang W. Set and string similarity queries: A survey. Chinese Journal of Computers, 2011,34(10):1853−1861 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.01853]

[23] Arasu A, Ganti V, Kaushik R. Efficient exactset-similarity joins. In: Proc. of the VLDB. 2006. 918−929.

[24] Sarma AD, He Y, Chaudhuri S. Clusterjoin: Asimilarity joins framework using map-reduce. PVLDB, 2014,7(12):1059−1070. [doi: 10.14778/2732977.2732981]

[25] Kim Y, Shim K. Parallel top-*k* similarity join algorithmsusing mapreduce. In: Proc. of the Int'l Conf. on Data Engineering (ICDE). 2012. 510−521. [doi: 10.1109/ICDE.2012.87]

附中文参考文献:

[22] 林学民,王炜.集合和字符串的相似度查询.计算机学报,2011,34(10):1853−1862. [doi: 10.3724/SP.J.1016.2011.01853]

王洪亚(1976－),男,河南开封人,博士,教授,博士生导师,CCF 会员,主要研究领域为数据库系统与理论,实时计算,移动计算.

杨利宏(1988—),男,硕士,主要研究领域为数据挖掘,数据清理.

刘晓强(1968—),女,博士,教授,主要研究领域为数据管理,信息系统,语义 Web,分布式计算,软件设计与评测.