自适应的 Top-k 重叠集合相似性连接

摘要

集合相似性连接 (set similarity join，简称 SSJ) 是数据清理、近似重复对象检测和数据集成等的核心步骤。Top-k SSJ 算法返回前 k 个最相似的集合对，无需提前给定阈值，从而跳出现有方法必须预设阈值的困境。现有 top-k 算法步长为 1，性能较低，为解决此问题，论文首先提出了一种使用固定步长的算法 (l-ssjoin) 以利用大步长的优势, 接着进一步提出能够自动调整步长的自适应步长算法 (adaptive-ssjoin)，有效减少了冗余验证。本课题实验部分针对论文提及的基础算法和两个改进算法进行复现。实验结果表明，相较步长为 1 的基础算法，固定步长 (大于 1) 和自适应步长两种改进算法存在明显的性能优势。

关键词：重叠集合相似性；相似性连接；top-k 连接；数据湖

1 引言

给定 2 个集合集和相似性函数，集合相似性连接 (set similarity join, 简称 SSJ) 的任务是找出所有相似性满足特定条件 (如超过某个阈值) 的集合对。SSJ 是数据清理[1]、近似重复对象检测[2]和数据集成[3]等的核心步骤。

现有的基于阈值的 SSJ 查询得到了广泛研究[2],[4]-[5]，但由于数据湖系统中，用户可能不具备相关数据集的先验知识，可利用信息很少，这为人工预设阈值带来了较大的困难。预设的阈值太低，得到的集合对数量大；预设过高，则结果不足。由于数据湖有很多不同的数据集，特征和需求不同时，针对每个数据集设定的阈值会有差别。因此，即使有熟练的技能，也很难提前设定阈值，尤其是数据集不断变化的情况下，预设阈值就更加困难。在这个背景下，top-k SSJ[6]被引入到相似集合对的筛选中。由于只需要选择前 k 个相似度较高的集合对，top-k SSJ 很好地越过了必须设定阈值的限制，并且 top-k 得到的结果，也能为后续选择合适的阈值提供参考。

论文研究了 top-k 重叠集合相似连接 (TkOSSJ)，即给定两个集合集，返回前 k 个重叠集合相似性最高的集合对。论文使用交集基数来作为重叠集合相似性[7]的指标。与常见的 Jaccard 和 Cosine 相似度相比，交集基数不受数据集大小影响，不会显著偏向小集合，而且计算和理解起来都十分简单。作者受到论文 top-k SSJ[6]的启发，并在将其作为本论文的基准算法之一。

在对 topk-join 的进一步探索中，作者发现步长 (即算法一次迭代处理的元素个数) 对算法性能会有显著影响。设定的步长不同，算法的运行效率也不同。基准的 topk-join 算法只考虑了步长为 1 的情况，因此性能较差。直观上来看，大的步长有利于在每次迭代中加快剪枝，但同时也带来候选集验证的额外开销。如果额外开销的代价小于剪枝带来的收益，那么算法的整体性能就会得到提升。基于以上的观察，论文提出了一种固定步长的算法，并用大步长来进行实验，说明步长变大的优势。然后进一步提出了自适应步长的方法，通过自动调整步长

来减少固定大步长带来的冗余计算问题。同时，作者基于 top-k 算法的运行流程，对集合验证进行了优化，进一步提升了性能。总体而言，作者的贡献如下：

(1) 观察到查询过程中步长对性能的影响，提出了高效的 TkOSSJ 算法。

(2) 提出一个固定步长的算法，比现有 topk-join 算法效果更好。

(3) 提出一个自适应的算法，能够在算法执行流程中自动调整步长，消除了需要人为界定合适步长的问题。该算法取得了与人工选取最佳固定步长相当的性能。

(4) 对算法进行实证分析，深入探讨改进算法的性质特征，展现他们确实在真实的大数据集上拥有比现有算法更好的性能。

2 相关工作

2.1 相似性连接和搜索的相关研究

2.1.1 基于前缀过滤的方法

Chaudhuri 等人首次提出了前缀过滤技术，并将其作为相似连接的原始算子[8]。Bayardo 等人引入了向量排序来解决内存语境下相似性连接的问题[9]。Xiao 等人通过添加位置过滤和后缀过滤改进了前缀过滤技术[2]。Wang 等人提出了一种自适应前缀过滤框架，通过使用成本模型为每个集合创建合适的前缀过滤器[4]。Bouros 等人提出 GroupJoin，将相同的前缀分组以节省计算成本[10]。Wang 等人通过将相关集合分组到索引中的块并跳过连接中无用的索引探测，进一步改进了工作[11]。Vernica 等人设计了一种基于前缀过滤器的并行算法，使用 MapReduce 来加速集合相似性连接[12]。Mann 等人提供了一项实证研究，包括该领域的所有主要参与者，并广泛涵盖了来自不同应用程序的数据集[13]。

2.1.2 其他方法

论文[14],[15]–[16]应用了基于局部敏感散列 (LSH) 的近似算法。Asymmetric Minwise Hashing[14]和 LSH Ensemble[17]将 MinHash 应用于集合包含搜索。此外，邓栋等人[5]提出了一种基于分区的精确相似连接方法，该方法将集合划分为几个子集，并保证两个集合只有在共享一个公共子集时才相似。Arasu 等人[18]开发了一种用于集合相似性连接的分区和枚举方法。张勇等人[19] 提出了一种基于转换的框架，该框架通过使用树结构索引来搜索相似集。Melnik 等人[20]提出了基于分区的集合包含连接算法。论文[21]、[22]中讨论了相似连接和集合包含连接之间的区别。McCauley 等人[23]提出了一种针对偏斜数据的数据依赖分区。陈刚等人[24]提出了一种适用于任何类型数据的通用相似性连接算法。

2.2 问题定义

论文的符号定义如图 1所示。以下定义一些基础概念。

定义 1 重叠集合相似度 (overlap set similarity). 两个集合 r 和 s 的重叠集合相似度是他

们交集大小的绝对数值。定义如下：

$$sim(r, s) = |r \cap s| \tag{1}$$

使用重叠集合相似度的好处有：(1) 与 Jaccard 和余弦相似度不同，重叠相似度不偏向小集合[14]；(2) 重叠相似度便于理解与计算。同时，作者提出的应用于重叠集合相似度的方法也可以很方便地拓展到其他相似性函数上。

Notation	Definition
$\mathcal{R}$	A collection of input sets
r or x	A set in collection $\mathcal{R}$
e	An element of a set
U	The universe of elements
pos	The position of the event element in the set
t_{ub}	The upper bound overlap similarity
$\langle x, e, pos, t_{ub}\rangle$	A prefix event
R	The result heap
t_k	The similarity of k-th best result
t_k^*	The final similarity of k-th best result
l	The step size
$I(e)$	The inverted list of element e
I_{EC}	The overall number of exceptional cases
X	The collection of sets whose size is larger than t_k^*
Θ_l	The overall cost of the method with step size l

图 1: 符号定义

定义 2 TkOSSJ. 给定两个集合集 $\mathcal{R}$ 和 $\mathcal{S}$，一个整数 k，top-k 重叠集合相似连接 (TkSSJ) 返回一个集合 R，该集合含具有最高相似度的 k 个集合对 $\langle r, s\rangle$，其中 $r \in \mathcal{R}$，$s \in \mathcal{S}$。

Set	Elements	Size
r_1	$\{e_1, e_2, e_3, e_4, e_6, e_7, e_8, e_9, e_{10}\}$	9
r_2	$\{e_1, e_2, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}\}$	9
r_3	$\{e_3, e_5, e_7, e_8, e_9, e_{10}, e_{11}\}$	7
r_4	$\{e_4, e_7, e_{11}\}$	3
r_5	$\{e_8, e_9, e_{11}\}$	3
r_6	$\{e_{10}, e_{11}\}$	2

图 2: 一个集合集 $\mathcal{R}$

图 2给出一个集合集示例，在后文中会沿用该示例对算法进行过程展示。根据定义 2，算法返回的相似度 top-2 集合对为 $\langle r_1, r_2\rangle$ 和 $\langle r_2, r_3\rangle$，相似度分别为 7 和 6。

作者在论文的论证和例子展示中只考虑了自连接的情形，即 $\mathcal{R} = \mathcal{S}$。自连接的情形也可以很容易地扩展到集合非自连接的情形。

2.3 前缀过滤框架

对于 TkOSSJ 查询问题，首先想到的是蛮力法，可以使用现有基于前缀过滤和阈值的 SSJ 查询，例如 ppjoin+[2]。主要思路是先设定一个较高的阈值，用前缀过滤筛选出高于该阈值的集合对，如果当前获得集合对的数量小于目标数量 k，那么降低该阈值再筛选，依此类推，直到获得 k 个集合对为止。

前缀过滤 令重叠相似度阈值为 t，前缀过滤的基本思路是，将相同元素数量低于 t 的集合对剪枝，再准确验证剩余的集合对的相似度。首先，对集合集的所有元素作全局排序。例如，可以根据元素出现的频率排序。对于每个集合，集合里的元素都根据全局顺序进行排列。设 prefix(r) 表示集合 r 的前缀，包含前 $|r|-t+1$ 个元素。如果集合 r 和 s 满足 $|r \cap s| \geqslant t$，r 和 s 的前缀一定至少有一个元素相同，即 $|\text{prefix}(r) \cap \text{prefix}(s)| \geqslant 1$。如果两个集合的的前缀没有元素相同，那么直接过滤。

倒排索引 元素 e 的倒排索引将其映射到一个包含 e 的集合列表，用 $I(e)$ 表示。论文中作者对所有集合前缀元素建立倒排索引，对满足 $|\text{prefix}(r) \cap \text{prefix}(s)| \geqslant 1$ 的集合对，精确验证是否满足阈值。

2.4 topk-join 算法

在蛮力法中，相似度高的集合对随着阈值下降，被重复计算。为了避免冗余计算，论文[6]提出了一种启发式算法 topk-join。此方法基于前缀过滤策略，并利用不可见集合对的上限相似度 (upper bound similarity)，来减少冗余运算。

在 topk-join 算法中，用一个优先队列 PQ，按降序存储所有集合的当前上限相似度。具体而言，对集合集中的每个集合 x，PQ 保存一个称为前缀事件 (prefix event) 的结构。该结构保存了一个前缀元素 e_i，它在迭代中会被检查，还有 x 从 e_i 的位置到集合末尾能达到多大的上限相似度 (即该集合还有多少个前缀事件待检查)。在每轮迭代中，算法处理集合 x 在 PQ 头部的前缀事件，检测倒排列表 $I(e_i)$，计算集合 x 和 $I(e_i)$ 中所有集合的相似度，最后按需更新结果集 R。倒排列表 $I(e_i)$ 通过插入集合 x 来进行更新，同时 x 的前缀事件也更新到了下一个元素 e_{i+1}，并插入该事件到 PQ 中。设 t_k 为 R 中的第 k 个相似度，t_{ub} 为 PQ 头部事件的上限相似度。当 PQ 为空或 t_k 超过 t_{ub} 时，算法终止。

前缀事件 前缀事件定义为 $\langle x, e_i, pos, t_{ub}\rangle$，其中 e_i 是集合 x 在位置 pos 的元素，t_{ub} 是 x 从位置 pos 到 x 末尾可能达到的上限相似度，即 $t_{ub} = |x| - pos + 1$。PQ 中前缀事件按 t_{ub} 倒序存放。一个前缀事件 $\langle x, e_i, pos, t_{ub}\rangle$ 一经处理，就被更新为 $\langle x, e_{i+1}, pos+1, t'_{ub}\rangle$，插入 PQ。其中，e_{i+1} 是 e_i 的下一个元素，且 $t'_{ub} = t_{ub} - 1$。

算法 1 描述了 topk-join 的算法流程。首先插入所有集合的第一个前缀事件，创建优先队列 PQ。PQ 的初始化阶段插入的前缀事件都按照集合大小降序排列。结果集 R 用一个固定大小的堆来存储 k 个最好的结果。算法迭代处理 PQ 的所有事件，并建立元素的倒排索引。集

合 x 的前缀事件一经处理，x 就被插入到对应的元素倒排列表 $I(e)$ 中。算法执行过程中，用哈希表来存储验证过的集合对，可以避免重复验证。

Algorithm 1: Algorithm Topk-Join

Input: $\mathcal{R}$ is a collection of sets
Output: Top-k set pairs R

```
PQ ← InitializeEvents(R);
R ← InitializeResults(R);
while PQ ≠ ∅ do
    ⟨x, e_i, pos, t_ub⟩ ← PQ.top(); t_k ← R[k].sim;
    if t_ub ⩽ t_k then
        break;
    foreach y ∈ I(e_i) do
        if (x, y) ∉ H then
            sim(x, y) ← Verification(x, y) ;
            H ← H ∪ (x, y) ;
            updatesR ;
    I(e_i) ← I(e_i) ∪ {x} ;
    t'_ub ← SimUpperBound(x, pos + 1) ;
    PQ.insert(⟨x, e_{i+1}, pos + 1, t'_ub⟩) ;
return R ;
```

在 2.5、2.6节介绍作者提出的改进方案：固定步长方法和自适应步长算法。

2.5 固定步长方法

作者基于 2.4节提到的基础 topk-join 算法作了改进，进一步提出一个固定步长的相似性连接方法，称为 l-ssjoin。其中 l 为固定步长大小。在一次迭代中，基础 topk-join 算法只处理队列 PQ 的一个元素。为进一步提升性能，l-ssjoin 在一次迭代中顺序处理队列 $l(l \geqslant 2)$ 个元素。在此意义下，作者提出的步长即为一个前缀事件被处理时，顺序探测到的元素个数。固定步长 l 有两大关键的优点。其一，第 k 个相似度的获取速度加快；其二，后续 prefix event 的相似上限 t_{ub} 也会快速下降。因此，算法的执行效率变高了。

以 2-ssjoin 为例，假设在一次迭代中，前缀事件 $\langle x, e_i, pos, t_{ub}\rangle$ 被处理，则从 PQ 顶部顺序探测队列中位于 pos 和 $pos+1$ 的 2 个元素 e_i 和 e_{i+1}。经过验证，将前缀事件 $\langle x, e_{i+2}, pos+2, t'_{ub}\rangle$ 插入 PQ，并更新 t'_{ub} 为 $t'_{ub} = t_{ub} - 2$。

为量化采用固定步长带来的效率提升，作者将算法的计算开销归结为扫描 (scanning)、验证 (verification)、更新 (update) 三个主要方面。扫描开销为 $\frac{1}{2}\sum_{e_i \in U} |I(e_i)||I(e_i) - 1| \cdot cost_s$，其中 $I(e_i)$ 为算法终止时元素 e_i 的倒排列表大小，$cost_s$ 为在倒排列表中定位一个集合的平均开销。验证开销为 $\frac{1}{2}\sum_{e_i \in U} |I(e_i)||I(e_i) - 1| \cdot (cost_h + \alpha \cdot cost_v)$，其中 $cost_h$ 为在哈希表中查找一个集合对的平均开销，$cost_v$ 为计算集合对相似性的验证和与 t_k 对比的总开销。更新开销为 $\frac{1}{2}\sum_{e_i \in U} |I(e_i)| \cdot cost_u$，其中 $cost_u$ 为每轮迭代中更新事件并插入到队列 PQ 的开销。

基于上述分析，作者得到了 topk-join 的开销公式，如式 2所示。

$$\Theta_{l=1} = \frac{1}{2}\sum_{e_i \in U} |I(e_i)_{l=1}|(|I(e_i)_{l=1}| - 1) \cdot cost_{shv} + \sum_{e_i \in U} |I(e_i)_{l=1}| \cdot cost_u \tag{2}$$

其中 $cost_{shv} = cost_s + cost_h + \alpha \cdot cost_v$。

2-ssjoin 的开销公式与 topk-join 类似，除了更新部分有些许不同，如式 3所示。不同点在于，2-ssjoin 的迭代次数是索引条目 (index entries) 数量的一半，因此第二项开销减半。

$$\Theta_{l=2} = \frac{1}{2}\sum_{e_i \in U} |I(e_i)_{l=2}|(|I(e_i)_{l=2}| - 1) \cdot cost_{shv} + \frac{1}{2}\sum_{e_i \in U} |I(e_i)_{l=2}| \cdot cost_u \tag{3}$$

引理 1　算法终止时，若 topk-join 和 2-ssjoin 的索引条目数量相同，即 $\forall e_i \in U, |I(e_i)_{l=1}| = |I(e_i)_{l=2}|$，则 $\Theta_{l=2} < \Theta_{l=1}$。

2-ssjoin 算法 (l-ssjoin 算法同) 可能会产生额外的索引条目。令 $t_k^\star$ 表示算法终止时产生的第 k 个相似度，显然无论步长大小是多少，$t_k^\star$ 都是一样的，所有 $t_{ub} > t_k^\star$ 的事件都必须被处理。算法产生的例外情况定义如下。

定义 3　例外情况. 若一个事件 $\langle x, e_i, pos, t_{ub}\rangle$ 满足 $t_k^\star < t_{ub} < t_k^\star + l$，当且仅当探测位置处于 $[|x| - t_k^\star + 1, |x| - t_{ub} + l]$ 范围内的元素 x 时，例外情况产生。

$l = 1$ 时，由于没有元素满足 $t_k^\star < t_{ub} < t_k^\star + 1$，没有额外事件产生。$l = 2$ 时，产生的额外事件如图 3所示。可以看到，算法处理了 $\langle x, e_6\rangle, \langle y, e_7\rangle, \langle z, e_7\rangle$ 三个额外元素，这些额外元素会导致后续的倒排索引列表变大，处理事件也因此变长了。

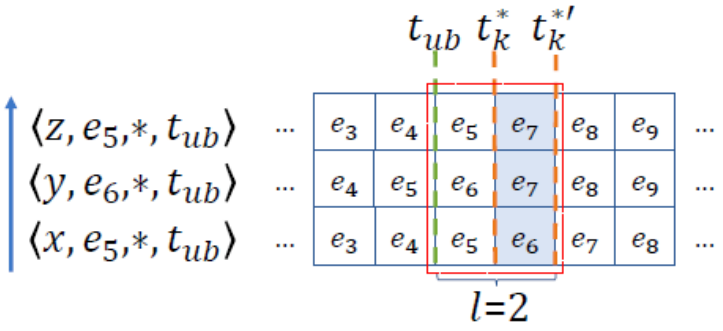

图 3: 2-ssjoin 产生的例外情况

令 N_{EC} 表示例外情况发生的次数，则 $N_{EC} \leqslant |X|$，其中 $X = \{x \in \mathcal{R} | |x| \geqslant t_k^\star\}$。即，$N_{EC} \in [0, |X|]$。topk-join 和 2-ssjoin 的索引条目之差就等于 2-ssjoin 例外情况的出现次数，如式 4所示。

$$\sum_{e_i \in U} |I(e_i)_{l=2}| = \sum_{e_i \in U} |I(e_i)_{l=1}| + N_{EC} \tag{4}$$

N_{EC} 相较于 $\sum_{e_i \in U} |I(e_i)_{l=1}|$ 而言，是很小的。同时，例外情况分布到不同的倒排列表 $I(e_i)$ 中，每个倒排列表内部的变化就十分微小。因此，topk-join 和 2-ssjoin 的索引条目约莫相等。

因此，topk-join 和 2-ssjoin 的开销之差可以用式 5计算。

$$\Theta_{l=1} - \Theta_{l=2} \approx \frac{1}{2}\sum_{e_i \in U} |I(e_i)_{l=1}| + N_{EC} \tag{5}$$

基于以上观察，例外情况部分可做剪枝处理。如果例外情况发生在位置 $pos + j$，同时相似上限满足 $t_{ub} - j \leqslant t_k$，且 $0 \leqslant j < l$，可以剪枝。剪枝后，N_{EC} 会变得更小，式 5的性能也得到了改进。因此，2-ssjoin[1]的总开销要比 topk-join 小。

2-ssjoin 可以拓展到 l-ssjoin。即，对于一个从队列 PQ 头部取出的前缀事件 $\langle x, e_i, pos, t_{ub}\rangle$，在每次迭代中遍历处理位于 $\{pos, pos+1, \ldots, pos+l-1\}$ 的 l 个元素 $\{e_i, e_{i+1}, \ldots, e_{i+l-1}\}$。$l$-ssjoin 的开销可以用式 6表示。

$$\Theta_l = \frac{1}{2}\sum_{e_i \in U} |I(e_i)_l|(|I(e_i)_l| - 1) \cdot cost_{shv} + \frac{1}{l}\sum_{e_i \in U} |I(e_i)_l| \cdot cost_u \tag{6}$$

步长 l 越大，对应的额外情况带来的开销 N_{EC} 也越大。由于步长 l 较小时，额外情况发生的次数也较少，l-ssjoin 的开销要比 topk-join 小，且随着步长 l 的增大不断下降。然而，当步长 l 增大到一定程度时，无法忽视额外情况带来的开销。就式 6而言，l 增大到一定程度时，第 1 项会显著增加，第 2 项带来的性能优化也将显著变小。因此，需要在增加步长和由此带来的额外开销之间做一个折衷 (trade-off)。即，需要寻找一个最优的步长 l，使性能达到最佳。

算法 2 描述了 l-ssjoin 的执行流程。在初始化阶段，步长 l 是一个预设好的定值。算法迭代处理前缀事件。在一次迭代中，每处理一个前缀事件，就顺序从集合 x 中取出 l 个元素。当该事件满足 $t_{ub} \leqslant t_k$，算法终止，返回结果集合对 R。对于每个 x 中的元素 e_{i+j}，计算他的相似上限 $t_{ub} - j$，并将该值与 t_k 对比。如果 $t_{ub} - j \leqslant t_k$，跳该前缀事件后续未处理的元素，减少例外情况的产生。对没有跳过的元素，遍历他的倒排列表，去找候选对。对不在哈希表里的候选对进行验证，并更新结果集。最后，如果 $t'_{ub} > t_k$，将下一个前缀事件 $\langle x, e_{i+l}, pos+l, t'_{ub}\rangle$ 插入队列 PQ，其中，$t'_{ub} = t_{ub} - l$。

2.6 自适应步长方法

对于不同数据集而言，一个合适的步长 l 很难提前得出。直接使用 l-ssjoin 去逐个试出步长，实际效率不够高。而且，一个固定长度的算法很难保证一直有最好的性能。即使一个大步长的算法性能很好，他带来的额外开销也很高，不能直接忽略不计。作者想到，如果在算法一开始使用较大的步长，在例外情况即将到来时，减小步长，如此一来，既能继续保证大步长的优势，又进一步减少了例外情况的发生。为了平衡 l-step 的性能提升和冗余验证带来的额外开销，论文提出了一种自适应步长的相似性连接方法。

算法 3 给出了自适应步长的算法实现细节。在此算法中，步长不是无止境增加的，他的上限由当前 PQ 的上限相似度和结果集中 R 中的第 k 个相似度共同决定。令 l_{ub} 表示 l 的上界，则它根据 $t_{ub} - t_k$ 动态更新。在每轮迭代中，l 如果小于 l_{ub}，就不断增加。否则，就被置

[1]例子过程详述见http://seamoon.lol/2022/11/13/topk-join/11/37/11/1/ah-papers/seamoon/

Algorithm 2: Algorithm l-ssjoin

Input: $\mathcal{R}$ is a collection of sets
Output: Top-k set pairs R

$PQ \leftarrow$ InitializeEvents($\mathcal{R}$);
$R \leftarrow$ InitializeResults($\mathcal{R}$);
$l \leftarrow$A fixed step size; /* For example $l \leftarrow 2$ */;
while $PQ \neq \emptyset$ **do**
  $\langle x, e_i, pos, t_{ub} \rangle \leftarrow PQ.top()$;
  $t_k \leftarrow R[k].sim$;
  if $t_{ub} \leqslant t_k$ **then**
    return R ;
  for $0 \leqslant j < l$ **do**
    if $t_{ub} - j \leqslant t_k$ **then**
      break;
    foreach $y \in I(e_{i+j})$ **do**
      if $(x, y) \notin H$ **then**
        $sim(x, y) \leftarrow Verification(x, y)$;
        $H \leftarrow H \cup (x, y)$;
        updates R;
      $I(e_{i+j}) \leftarrow I(e_{i+j}) \cup \{x\}$;
    $t_{ub} \leftarrow t_{ub} - 1$;
  $t'_{ub} \leftarrow SimUpperBound(x, pos + l)$;
  if $t'_{ub} > t_k$ **then**
    $PQ.insert(\langle x, e_{i+l}, pos + l, t'_{ub} \rangle)$;
return R ;

成 l_{ub}。在初始阶段，$l < l_{ub}$，步长不断增大，算法加速获得前 k 个相似度。当算法即将要终止时，$l = l_{ub}$，有效减少了例外情况的发生。

2.7 基于事件特性的集合对验证方法

在基于事件的验证中，作者获取倒排列表中最后看到的重叠位置。它可以跳过该位置之前的元素。与论文提出的基于事件的验证相比，基于位置的验证方法存在比较集合对的冗余成本，而高级验证方法需要记录两组中最后看到的重叠的位置。

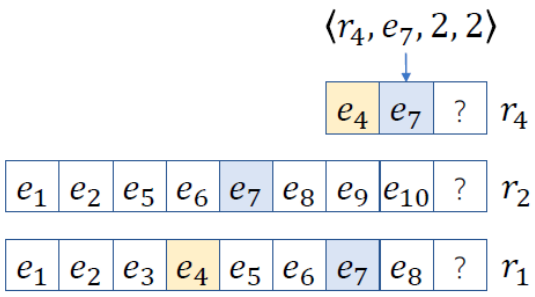

图 4: 基于事件的集合对验证示例

图 4展示了一个基于事件的集合对验证示例。假设当前在队列 PQ 头部的前缀事件为

Algorithm 3: Adaptive Step Size Join

Input: $\mathcal{R}$ is a collection of sets
Output: Top-k set pairs R

```
1  PQ ← InitializeEvents(R);
2  R ← InitializeResults(R);
3  l ← 1;
4  while PQ ≠ ∅ do
5      ⟨x, e_i, pos, t_ub⟩ ← PQ.top();
6      t_k ← R[k].sim;
7      if l < t_ub − t_k then
8          l ← l + 1 ;
9      else
10         l ← t_ub − t_k ;
11     same as Line 7-21 in Algorithm 2;
12 return R ;
```

$\langle r_4, e_7, 2, 2\rangle$，步长 $l = 1$。e_7 是 r_4 中最后一个可见元素，此时 $I(e_7) = \{r_2, r_1\}$。算法不会重复验证同一个集合对，以 r_1 为例，在 e_7 之前，即 $r_4.e_4$ 时，(r_1, r_4) 已经被验证，所以不会重复验证。而 (r_2, r_4) 未被验证，说明在 e_7 之前，r_2, r_4 没有相同的元素。因此，对 r_2, r_4 而言，e_7 是可见部分第一个也是最后一个相同的元素[2]。因此，可以直接从 $r_2.e_7$ 和 $r_4.e_7$ 开始比较，减少了前面元素的冗余验证。

3 复现细节

3.1 与已有开源代码对比

本研究报告没有参考任何相关源代码。

3.2 实验环境搭建

真实环境：Intel 2.9GHz CPU, 32GB memory on Ubuntu virtual machine

原论文环境：Intel 3.4GHz CPU, 32GB memory on a Linux machine

所有算法代码均采用 C++ 编写，并加入-O3 选项进行编译优化。

3.3 实验数据

由于年份差异，数据集可能有所更新，以及作者未说明部分数据集的具体版本或裁剪方式，给数据集的准确获取带来较大困难。表 1选取了与原论文数据最接近的 3 个数据集（除 Enron 数据集外，其余与原论文统计数据均有细微差异），分别是 Enron[3]，kosarak[4]，和 orkut[5]。

对于单个数据集，其数据的条数成为 cardinality，一条数据为一个集合，这个集合里面的元素根据数据集的不同，可以是字符串或数字等。数据集的分解和使用见表 2。

[2] r_2 的可见部分为 $\{e_1, e_2, e_5, e_6, e_7\}$，$r_4$ 的可见部分为 $\{e_4, e_7\}$.

[3] https://www2.cs.sfu.ca/ jnwang/projects/adapt/

[4] http://fimi.ua.ac.be/data/

[5] https://snap.stanford.edu/data/com-Orkut.html

Dataset	Cardinality	avg-len	max-len	# of elements
Orkut	2,723,360	44.03	33,008	3,072,441
Kosarak	99,0002	8.1	2,498	41,270
Enron	517,422	133.58	3,162	1,113,231

表 1: 数据集统计

Dataset	Description	Set	Element
Orkut	Data from ORKUT social network	A user	A group member-ship of the user
Kosarak	Click-stream data	The recorded behavior of a user	A link clicked by the user
Enron	Real e-mail data	An email	A word from the subject or the body field

表 2: 数据集的使用

3.4 界面分析与使用说明

VS Code 配置 C++ 环境和 Run Code 组件，直接运行代码即可。在 main 函数中可以手动调整输入参数值，得到一个算法下不同数据集对应的统计结果。运行界面和一个示例运行结果如图 5、图 6所示。

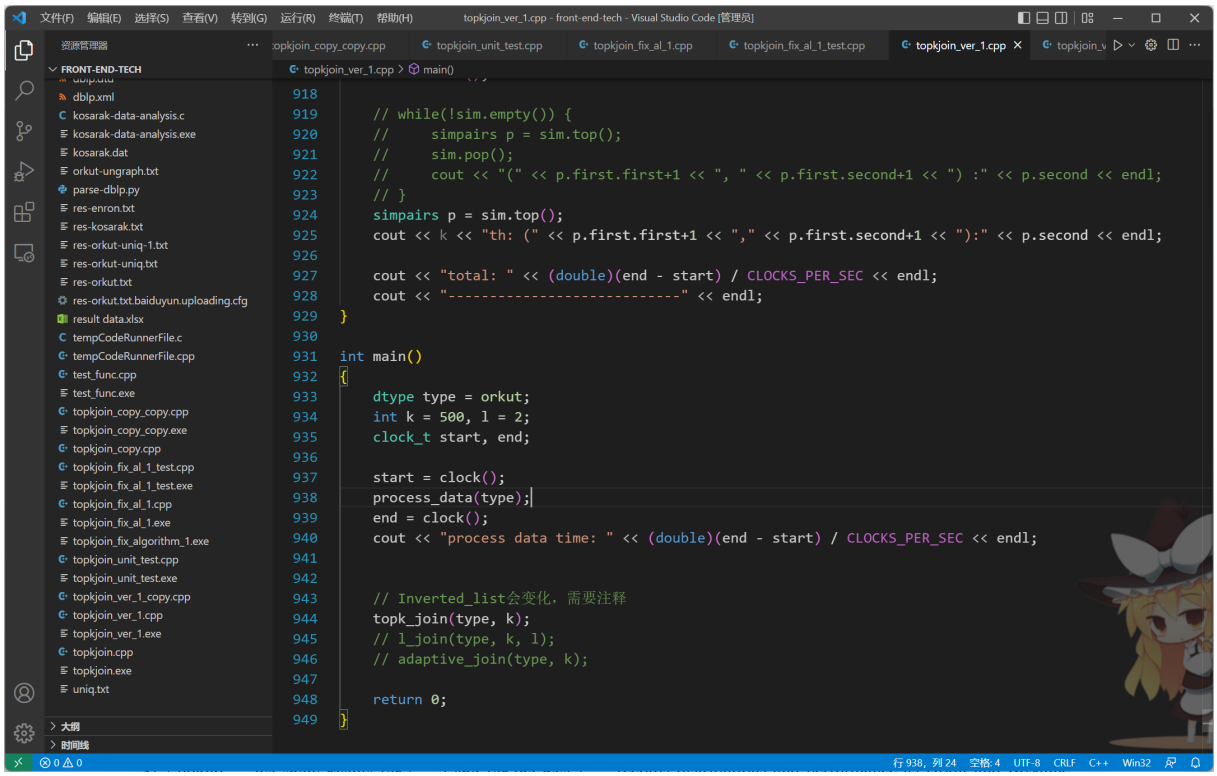

图 5: 运行界面

```
PS C:\Users\dell\Desktop\course-works\front-end-tech> cd "c:\Users\dell\Desktop\course-works\front-end-tech\"
 if ($?) { g++ topkjoin_ver_1.cpp -O3 -o topkjoin_ver_1 } ; if ($?) { .\topkjoin_ver_1 }
process data time: 60.143
TopK-Join(k=500,data=orkut)
Number of candidates: 0
Number of Index Entries: 1982858
500th: (42152,301638):2773
running time: 15.377
----------------------------
```

图 6: 示例运行结果 (dataset=orkut, k=500, l=2, algorithm=top-k)

4 实验结果分析

本章给出原文和复现的比对结果，并分析异同。

4.1 第 k 个结果的准确性对比

如图 7所示，其中同一个数据集的第 1 行是原论文结果，第 2 行是复现结果，标红部分为不匹配的结果。第 3.3小节指出，本次实验使用的数据集因年份差异，以及作者并未明确提及测试数据所用版本，所用数据集部分数据有所变动，因此第 k 个结果有极小部分不同，但在误差允许范围内。随着 k 值增大，top-k 结果变小，总趋势与论文一致。

k	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500
Orkut	4168	3551	3231	2980	2773	2620	2486	2379	2291	2212	2134	2072	2017	1962	1917
	4168	3550	3230	2980	2773	2620	2486	2379	2291	2211	2134	2071	2016	1961	1917
Kosarak	722	643	611	598	588	580	575	569	564	561	557	552	548	545	541
	725	645	612	598	588	580	575	569	565	561	557	552	548	545	541
Enron	2910	2887	2794	2794	2728	2627	2601	2599	2530	2514	2497	2487	2392	2377	2286
	2910	2887	2794	2794	2728	2627	2601	2599	2530	2514	2497	2487	2392	2377	2286

图 7: 第 k 个结果的比较

4.2 不同数据集上的运行时间 ($k = 500$)

如图 8所示，其中同一个数据集的第 1 行是原论文结果，第 2 行是复现结果，横线标注的是最优运行时间。可以发现，topk-join 和 2-ssjoin 在 Orkut 数据集的复现运行速度比原论文要快 1-3 倍，在其他数据集上则慢于论文结果。原因可能是代码的具体实现有差异。总体而言，对同一个数据集，运行时间从 topk-join 到 2-ssjoin、l-ssjoin、A-ssjoin，主要呈减少趋势。

Dataset	topk-join	2-ssjoin	l-ssjoin	A-ssjoin
Orkut	68.347	33.644	5.134	4.746
	20.093	18.619	9.224	8.241
Kosarak	0.921	0.525	0.147	0.154
	0.397	0.360	0.291	0.295
Enron	0.512	0.290	0.069	0.153
	0.717	0.683	0.605	0.606

图 8: $k = 500$ 时不同数据集上的运行时间 (s)

4.3 运行时间随 k 的变化

图 9展示了运行时间随 k 的变化。上方的图片为复现结果，下方为原论文结果（下同）。结果基本吻合。可以看到，从基准 topk-join 算法到后面的固定步长和自适应步长，作者提出的优化算法具有明显的效能提升，l-ssjoin 算法和 A-ssjoin 算法的运行时间在大多数情况都十分接近。运行时间随 k 的增长随数据集的不同而有所区别。对 Orkut 和 Kosarak 数据集，随着 k 的增长，运行时间大约呈线性增长，而 Enron 数据集在 k 较大时呈指数增长。

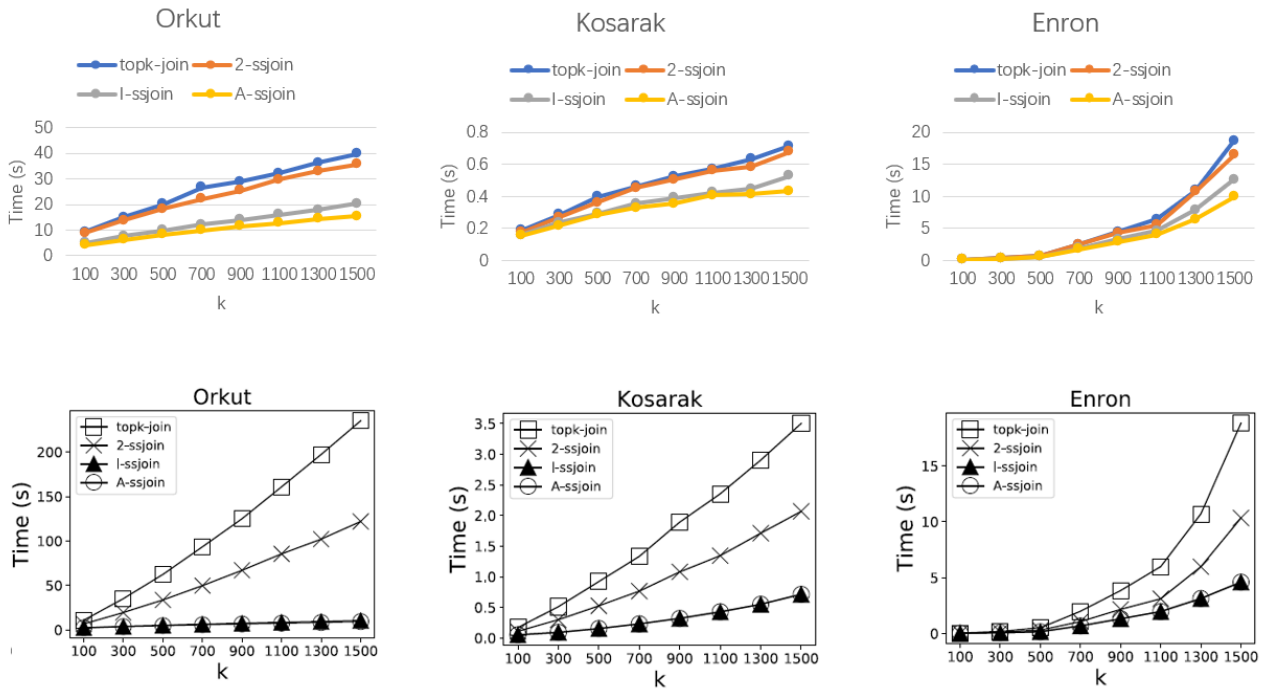

图 9: 运行时间随 k 的变化

4.4 运行时间随步长 l 的变化

图 10展示运行时间随步长 l 的变化。随步长的增大，运行时间有所下降，与原论文趋势相同，符合预期。与原论文稍有不同的是，Kosarak 数据集和 Enron 数据集的结果在初始阶段下降的趋势不太明显。

4.5 索引条目的数量随 k 的变化

索引条目总和是所有元素倒排列表里的集合总量，用式 7计算。

$$IndexNum = \sum_{1 \leqslant i \leqslant n} |I(e_i)| \tag{7}$$

图 11展示了索引条目的数量变化。可以看到，随着 k 的增大，索引条目总数呈明显的上升趋势。A-ssjoin 的索引条目总数最多，这是由于其不断变化步长的缘故。但其运行时间却比其他算法要低，是因为增加步长而带来的增益大于例外情形带来的损耗（即索引条目的增加）。复现结果和原论文结果高度一致。

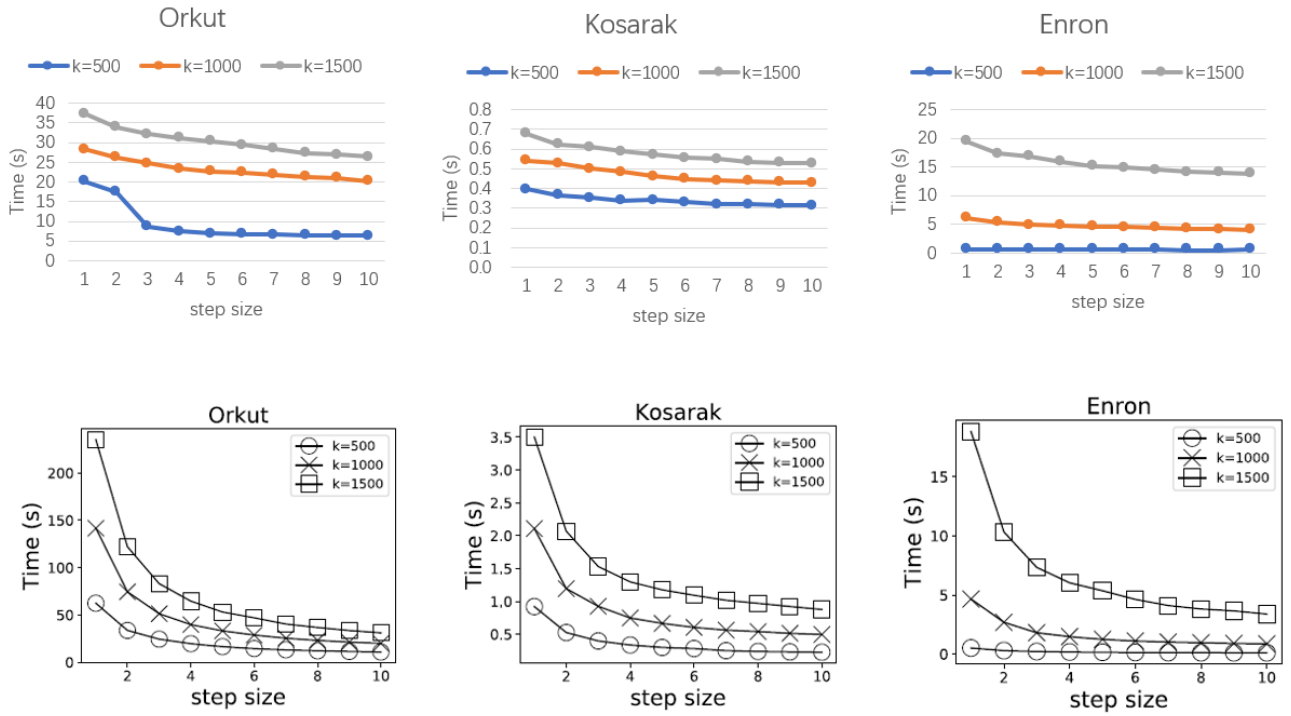

图 10: 运行时间随 l 的变化

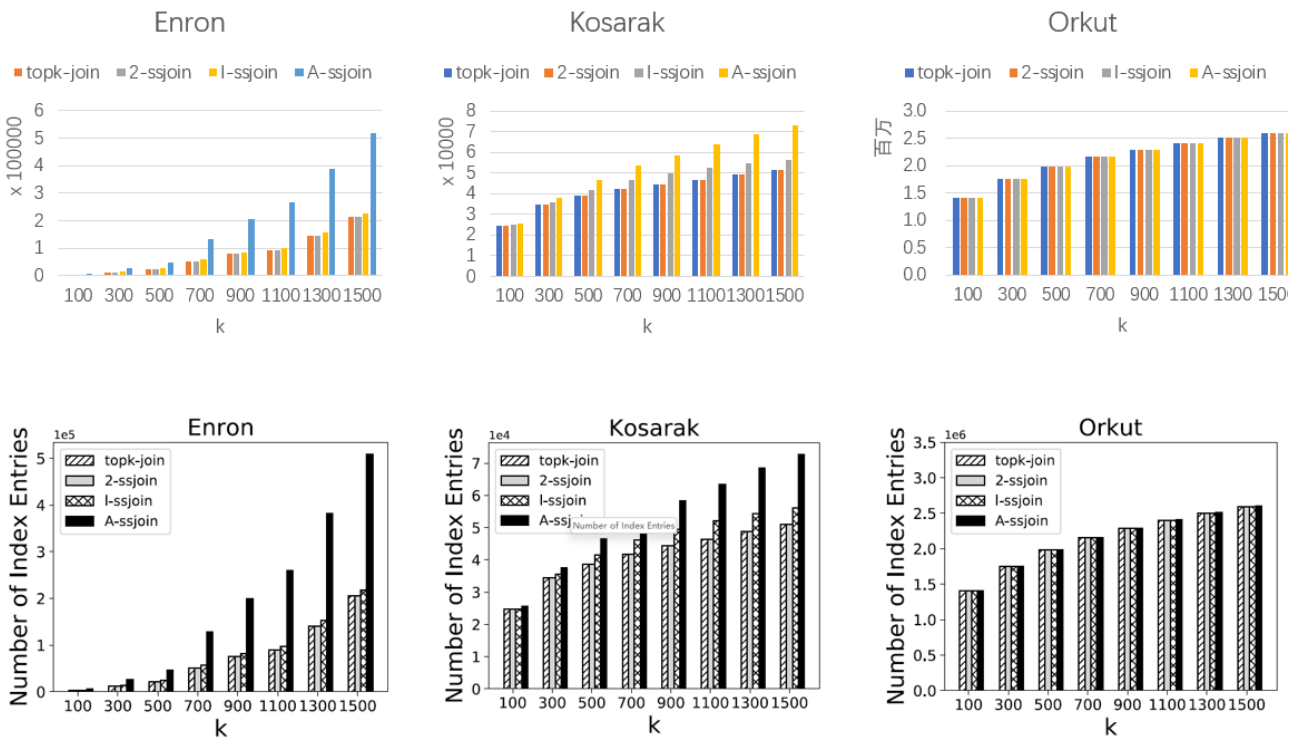

图 11: 索引条目数量随 k 的变化

4.6 例外情况的数量随 k 的变化

图 12展示了例外情况随 k 和 l 的变化情况。这个数量随着步长 l 的增加而增加，证实了2.5节中的陈述。观察到当 l 变大时，Enron 上例外情况发生的数量迅速增加。在其他两个数据集上，尤其是在 Orkut 上，它增加缓慢。增长率在很大程度上取决于数据分布。Orkut 和 Kosarak

的集合大小服从幂律分布，它们在例外情况发生次数上的增长率缓慢。相比之下，Enron 的记录大小分布不同（见原论文 P8 顶部），增长速度较快。

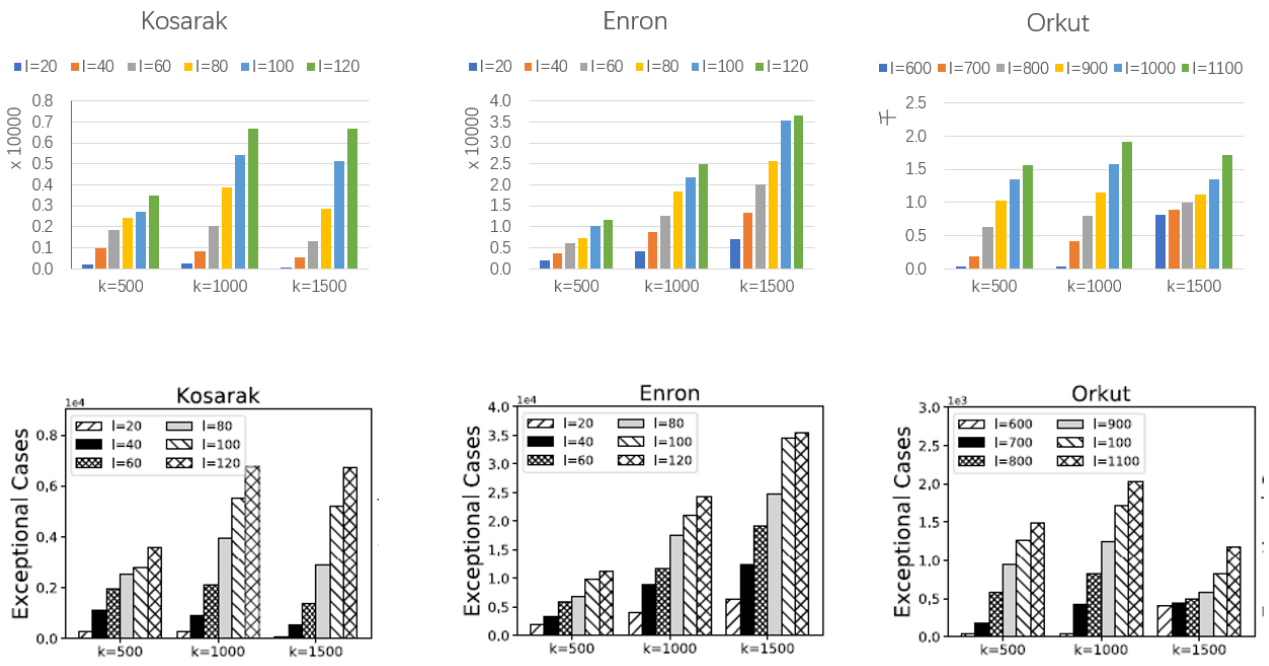

图 12: 例外情况数量随 k 的变化

5 总结与展望

本文中深入研究和复现了 Topk 重叠集合相似性连接 (TkOSSJ) 查询。作者从固定大小的步长算法 l-ssjoin 开始分析，从理论和实验 2 个方面证明了该算法优于 topk-join。此外，为了减少例外情况的影响，避免手动设置步长，作者进一步提出了一种自动调整步长的自适应大小步长算法 A-ssjoin。该算法在实践中更具可行性。大规模真实数据集的实验表明，所提出的算法优于基线算法一个数量级。

本次课程研究耗费了许多时间阅读相关论文，寻找数据集和编写代码、测试等等。最终复现结果与论文基本一致。由于一些数据集无法获得，或没有彻底明白应如何运用，只复现了其中三个数据集的大部分运行结果。剩下待完成的工作可以是：

(1) 研究数据集分布对 Topk 结果的影响；

(2) 在 DBLP，MovieLens，LiveJ 数据集上进行实验；

(3) 实现对比算法 (如 topk 应用到 ppjoin 的算法 pptopk 等)；

(3) 尝试了很多方法，均无法得出作者对候选集 (candidates) 数量的测试结果，等待补全。

此番论文复现历经艰辛，不禁让人感叹论文发表的不易。在未来的工作中，可研究如何将步长应用于其他相关问题，例如 top-k 重叠集合相似性搜索、基于阈值的集合相似性搜索和算法并行化。后续可以思考如何将此论文应用到现有的研究方向 (如，FP-tree 集合相似性连接的 Top-k 实现) 上。

参考文献

[1] YU M, LI G, DENG D, et al. String similarity search and join: a survey[J]. Frontiers of Computer Science, 2016, 10(3): 399-417.

[2] XIAO C, WANG W, LIN X, et al. Efficient similarity joins for near-duplicate detection[J]. ACM Transactions on Database Systems (TODS), 2011, 36(3): 1-41.

[3] COHEN W W. Integration of heterogeneous databases without common domains using queries based on textual similarity[C]//Proceedings of the 1998 ACM SIGMOD international conference on Management of data. 1998: 201-212.

[4] WANG J, LI G, FENG J. Can we beat the prefix filtering? An adaptive framework for similarity join and search[C]//Proceedings of the 2012 ACM SIGMOD international conference on management of data. 2012: 85-96.

[5] DENG D, LI G, WEN H, et al. An efficient partition based method for exact set similarity joins [J]. Proceedings of the VLDB Endowment, 2015, 9(4): 360-371.

[6] XIAO C, WANG W, LIN X, et al. Top-k set similarity joins[C]//2009 IEEE 25th International Conference on Data Engineering. 2009: 916-927.

[7] DENG D, TAO Y, LI G. Overlap set similarity joins with theoretical guarantees[C]// Proceedings of the 2018 International Conference on Management of Data. 2018: 905-920.

[8] CHAUDHURI S, GANTI V, KAUSHIK R. A primitive operator for similarity joins in data cleaning[C]//22nd International Conference on Data Engineering (ICDE'06). 2006: 5-5.

[9] BAYARDO R J, MA Y, SRIKANT R. Scaling up all pairs similarity search[C]//Proceedings of the 16th international conference on World Wide Web. 2007: 131-140.

[10] BOUROS P, GE S, MAMOULIS N. Spatio-textual similarity joins[J]. Proceedings of the VLDB Endowment, 2012, 6(1): 1-12.

[11] WANG X, QIN L, LIN X, et al. Leveraging set relations in exact set similarity join[J]. Proceedings of the VLDB Endowment, 2017.

[12] VERNICA R, CAREY M J, LI C. Efficient parallel set-similarity joins using mapreduce[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. 2010: 495-506.

[13] MANN W, AUGSTEN N, BOUROS P. An empirical evaluation of set similarity join techniques[J]. Proceedings of the VLDB Endowment, 2016, 9(9): 636-647.

[14] SHRIVASTAVA A, LI P. Asymmetric minwise hashing for indexing binary inner products and set containment[C]//Proceedings of the 24th international conference on world wide web. 2015: 981-991.

[15] BRODER A Z, GLASSMAN S C, MANASSE M S, et al. Syntactic clustering of the web[J]. Computer networks and ISDN systems, 1997, 29(8-13): 1157-1166.

[16] CHRISTIANI T, PAGH R. Set similarity search beyond minhash[C]//Proceedings of the 49th annual ACM SIGACT symposium on theory of computing. 2017: 1094-1107.

[17] ZHU E, NARGESIAN F, PU K Q, et al. LSH ensemble: Internet-scale domain search[J]. arXiv preprint arXiv:1603.07410, 2016.

[18] ARASU A, GANTI V, KAUSHIK R. Efficient exact set-similarity joins[C]//Proceedings of the 32nd international conference on Very large data bases. 2006: 918-929.

[19] ZHANG Y, LI X, WANG J, et al. An efficient framework for exact set similarity search using tree structure indexes[C]//2017 IEEE 33rd International Conference on Data Engineering (ICDE). 2017: 759-770.

[20] MELNIK S, GARCIA-MOLINA H. Adaptive algorithms for set containment joins[J]. ACM Transactions on Database Systems (TODS), 2003, 28(1): 56-99.

[21] YANG J, ZHANG W, YANG S, et al. Tt-join: Efficient set containment join[C]//2017 IEEE 33rd International Conference on Data Engineering (ICDE). 2017: 509-520.

[22] DENG D, YANG C, SHANG S, et al. LCJoin: set containment join via list crosscutting[C]// 2019 IEEE 35th International Conference on Data Engineering (ICDE). 2019: 362-373.

[23] MCCAULEY S, MIKKELSEN J W, PAGH R. Set similarity search for skewed data[C]//Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. 2018: 63-74.

[24] CHEN G, YANG K, CHEN L, et al. Metric similarity joins using MapReduce[J]. IEEE Transactions on Knowledge and Data Engineering, 2016, 29(3): 656-669.