计 算 机 研 究 与 发 展
Journal of Computer Research and Development

DOI: 10.7544/issn1000-1239.202111239
60(12): 2890−2906, 2023

基于 FP-tree 和 MapReduce 的集合相似度自连接算法

冯禹洪[1] 吴坤汉[1] 黄志鸿[1] 冯洋洲[1] 陈欢欢[2] 白鉴聪[1] 明 仲[1]

[1](深圳大学计算机与软件学院 广东深圳 518060)

[2](中国科学技术大学计算机科学与技术学院 合肥 230027)

(yuhongf@szu.edu.cn)

A Set Similarity Self-Join Algorithm with FP-tree and MapReduce

Feng Yuhong[1], Wu Kunhan[1], Huang Zhihong[1], Feng Yangzhou[1], Chen Huanhuan[2], Bai Jiancong[1], and Ming Zhong[1]

[1](*College of Computer Science and Software Engineering*, *Shenzhen University*, *Shenzhen*, *Guangdong* 518060)

[2](*School of Computer Science and Technology* , *University of Science and Technology of China*, *Hefei* 230027)

Abstract Given a collection of sets, the set similarity self-join (SSSJ) algorithm finds all pairs of sets whose similarity is higher than a given threshold, which has been widely used in various applications. The SSSJ adopting the filter-and-verification framework and the parallel and distributed computing framework MapReduce is an active research topic. But existing algorithms produce large candidate sets when the given threshold is low and lead to poor performance. To address this problem, we propose to compute the SSSJ with frequent pattern tree structures and their derivatives FP-tree*, which are used to compress data in memory for computation, targeting at reducing the size of candidate sets. First, based on an investigation on existing FP-tree* and the characteristics of SSSJ computation, we propose a more traversal efficient linear prefix tree structure, i.e., TELP-tree, and an original SSSJ algorithm accordingly, i.e., TELP-SJ, which includes a two-phase filtering strategy: tree-construction oriented filtering algorithm and tree-traversal oriented filtering strategy. These algorithms will reduce the size of TELP-trees and tree-traversals. Then, we design the parallel and distributed SSSJ computation algorithm FastTELP-SJ with MapReduce. Finally, 4 groups of empirical comparison studies have been carried out on 4 sets of real application datasets. The experimental results demonstrate that FastTELP-SJ achieves better performance in term of the execution time, memory usage, disk usage and scalability for similarity joins over large scale collections of sets with high dimension.

Key words similarity join; FP-tree; MapReduce framework; Jaccard function; set

摘 要 利用集合相似度自连接算法找出一个集合集中所有相似度大于给定阈值的集合对有着广泛的应用.基于过滤-验证框架和并行分布式计算框架 MapReduce 的集合相似度连接是近年来的研究热点.但现有算法在阈值低时产生较大规模的候选集,导致性能不理想.针对这一问题,提出采用频繁模式树 FP-tree 及其派生结构 FP-tree*将数据压缩在内存中计算集合相似度自连接以减小候选集规模.首先设计并讨论基于现有 FP-tree*的集合相似度连接计算及其优缺点,提出遍历效率更高的线性频繁模式树结构模型

收稿日期: 2021-12-15; **修回日期**: 2023-02-01

基金项目: 国家自然科学基金项目(62272315, 61836005, 62002230);深圳市基础研究面上项目(JCYJ20210324093212034);广东省自然科学基金项目(2019A1515012053);腾讯“犀牛鸟”-深圳大学青年教师科研基金项目;深圳市稳定支持计划项目(20200814105901001)

This work was supported by the National Natural Science Foundation of China (62272315, 61836005, 62002230), the Shenzhen Fundamental Research General Program (JCYJ20210324093212034), the Natural Science Foundation of Guangdong province (2019A1515012053) , Tencent “Rhinoceros Birds” - Scientific Research Foundation for Young Teachers of Shenzhen University, and the Shenzhen Stable Support Program (20200814105901001).

通信作者: 白鉴聪(szubjc@szu.edu.cn)

TELP-tree 及基于它的算法 TELP-SJ（TELP-tree self join），其包括分别面向构建树和遍历树的 2 阶段过滤算法，这些算法可以减小树规模和减少树遍历. 然后，设计基于 MapReduce 的并行分布式算法 FastTELP-SJ. 最后，基于 4 组真实应用数据集进行 3 组性能比较实验. 实验结果表明 FastTELP-SJ 算法面向高维大规模集合相似度自连接计算时，包括执行时间、内存占用率、磁盘使用量和可扩展性的运行效率最好.

关键词 相似度连接；FP 树；MapReduce 框架；Jaccard 函数；集合

中图法分类号 TP391

集合相似度自连接是大数据分析的重要操作，它是从一个集合集中找出相似度大于给定阈值的集合对，有着广泛的应用，如协同过滤[1]、检测近似重复的网页[2]、社交网络分析[3] 等. 其中，度量 2 个集合的相似度可根据应用选择合适的度量函数，如 Jaccard 函数.

集合相似度自连接需要度量集合集中所有集合对的相似度，时间复杂度为$O(kn^2)$，其中n为集合集的大小，k为任意 2 个集合相似度的平均计算时间. 当集合规模较大时，如 2019 年第 3 季度知名社交网络平台 Facebook 的月活跃用户人数为 24.5 亿[4]，每个用户都有好友集合. Facebook 用户好友集合的自连接计算中$n=2.45\times10^9$，这样的大规模数据集给时间复杂度高的集合相似度自连接计算带来挑战. 基于要处理的数据是静态数据还是动态数据（流数据），现有集合相似度自连接计算方法可以划分为 2 类：批处理式计算和流处理式计算.

首先，批处理式集合相似度自连接计算要处理的是静态数据，运行过程中没有新增数据. 现有提高运行效率的批处理式集合相似度自连接计算方法可以划分为 2 类：近似计算方法和基于过滤-验证框架的准确计算方法. 近似计算方法通过设计合适的集合相似度快速估算方法来加速计算，如基于位置敏感哈希函数的 MSJL[5]、基于局部敏感过滤的 LSF-Join[6]、基于高维数据速写和索引的 CPSJoin[7] 等，但因估算降低了精度导致不能找出所有相似度超过阈值的集合对. 本文将关注准确计算方法.

基于过滤-验证框架的准确计算方法包括 all-pair[8]，PPJoin[2]，MPJoin[9]，AdaptJoin[10]，GroupJoin[11] 等，其计算采用先过滤后验证的 2 阶段计算方法. 第 1 阶段使用合适的过滤策略将检测发现的相似度低于阈值的集合对删除，获得较小的候选集；第 2 阶段基于规模减小后的候选集验证并输出相似度大于给定阈值的集合对，提高了效率. 有效的过滤策略是提高这类算法的重要支撑技术，常用策略包含长度过滤[12]、前缀过滤[8]、位置过滤[2] 等. 最近提出的位图过滤[13] 采用哈希函数创建表达集合特征为固定长度的位图，然后通过位操作推断集合对相似的上界，实现相对使用其他策略最高达 4.50 倍的性能提升.

同时，并行分布式计算方法通过将大规模计算或数据划分为一系列较小的分片，每个分片被分发到不同计算节点上并行运行以提高计算效率. 根据计算节点的体系结构，并行分布式集合相似度自连接计算可以划分为基于 GPU 的计算和基于 CPU 的计算. 基于 GPU 的系列算法有效提升了集合相似度连接的计算性能[14]，但是，基于 GPU 的实现需要对硬件有深入的了解，这往往需要编程者付出很高的学习成本. 而基于 CPU 的集群计算是更为普及的并行分布式计算，现有的基于 CPU 的并行分布式集合相似度连接算法大部分基于 MapReduce[15] 框架. 根据处理过程是否将集合切分为多段，基于 MapReduce 框架和过滤-验证框架的集合相似度连接算法可划分为 2 大类：1）集合整体过滤与验证，这类算法包括 RIDPairsPPJoin（下文简称为 RP-PPJoin，有些文献也称为 VernicaJoin）[16]，MGJoin[17]，SSJ-2R[18] 等；2）集合数据分段过滤与验证，如 FS-Join[19]. 第 1 类算法将集合切分为多段，分段过滤，然后合并验证. 这类算法容易被理解，但存在同一集合多次重复验证的现象. 第 2 算法避免了重复验证. 但当阈值较低时，这 2 类算法产生的候选集规模都比较大，运行效率并不理想[20].

其次，流处理式集合相似度自连接计算处理的是动态数据，该数据指的是随时间依次产生的数据，如网络监控数据、气象测控等. 其数据量随着时间增长，显著特征是数据有时间戳，能快速生成. 集合流处理计算通常基于批处理计算方法，引入集合相似度时间衰减因子、采样降低数据量从而提高计算效率，如串行式计算的基于 PPJoin 设计的流处理计算算法 SSTR[21]、基于更新日志的递增式流处理计算算法 ALJoin[22] 等，分布式计算的流处理计算算法 BundleJoin[23] 和 KNN Join[24] 等.

综上所述，面向静态数据基于 MapReduce 的并行分布式集合相似度自连接计算效率无论对批处理

式计算还是流处理式计算都很重要，本文拟采用频繁模式树(frequent pattern tree, FP-tree)[25]，通过用更紧凑的扩展前缀树结构表示数据并压缩到内存中处理，减小生成的候选集的规模，同时减少 I/O 操作，降低后续计算复杂度. 本文将系统地研究 FP-tree 及其派生结构 FP-tree*，设计针对集合相似度自连接计算更有效的 FP-tree*，并设计相应的基于 MapReduce 框架的并行分布式集合相似度自连接算法. 本文的主要贡献有 3 点：

1)提出面向集合相似度自连接的遍历效率更高的线性频繁模式树 TELP-tree(traversal efficient linear prefix tree)结构模型及基于它的集合相似度自连接算法 TELP-SJ，该算法包括 2 阶段过滤算法，减小树的规模和减少对树结点的遍历，降低集合对相似度计算和验证的时空复杂度；

2)设计基于 TELP-tree 和 MapReduce 的并行分布式集合相似度自连接算法 FastTELP-SJ；

3)基于 4 组真实应用数据集进行 3 组性能比较实验，分别进行 TELP-SJ 算法与其他基于 FP-tree*的算法，如 FastTELP-SJ，TELP-SJ，FastTELP-SJ 等和现有其他基于 MapReduce 算法的比较，本文实验结果展现了 FastTELP-SJ 算法在处理高维大规模集合相似度自连接计算时的性能优于其他算法.

1 基于 FP-tree 的集合相似度自连接算法

给定集合集$x=\{x_1, x_2, \cdots, x_n\}$，任意$x_i, x_j \in x$，$x_i$和$x_j$都是集合，集合相似度度量函数$sim(x_i, x_j)$，阈值$t$，$x$的集合相似度自连接是找出所有满足$sim(x_i, x_j) \geqslant t$的集合对$(x_i, x_j)$.

为方便讨论，本文以 Facebook 开放的大规模应用数据集 Facebook applications dataset II[26]的部分数据为例，演示基于其构建的 FP-tree 派生结构，面向集合相似度自连接计算存在的问题及后续算法的处理过程. 示例数据如图 1 的X所示，其中 *uid* 代表用户标识号，*aid* 代表应用标识号. 在示例数据的应用场景中，*uid* 和 *aid* 可以分别被理解为集合标识号和元素标识号. 对任一用户u_i，其使用的应用可表示为集合$A(u_i)$，如$A(u_1)=\{a_1, a_2, a_3, a_4, a_5\}$. 如式(1)所示，当集合$A(u_i)$和$A(u_j)$的相似度$sim(A(u_i), A(u_j))$用$Jaccard(A(u_i), A(u_j))$来计算时，$sim(A(u_1), A(u_3))=3/(5+3-3)=0.6$.

$$sim(A(u_i), A(u_j))=\frac{|A(u_i)\cap A(u_j)|}{|A(u_i)|+\left|A(u_j)\right|-|A(u_i)\cap A(u_j)|}. \quad (1)$$

FP-tree 数据结构最初用于挖掘频繁模式，为使用它来计算集合相似度自连接，我们首先要将其中集合交集大小的计算转化为基于频繁二项模式挖掘的计算. 为此，转置X的行与列得到X'的数据，X中任意 2 个集合$A(u_i)$和$A(u_j)$的交集大小$|A(u_i)\cap A(u_j)|$的计算就转换成X'中二项集(u_i, u_j)出现的频次$f_{i,j}$[27]的计算. 如X'中二项集(u_1, u_3)出现的频次$f_{1,3}=3$,示例数据中$|A(u_1)\cap A(u_3)|=|\{a_1, a_2, a_3\}|=3$，$f_{1,3}=|A(u_1)\cap A(u_3)|$. 另外，$X'$中一项集$u_i$出现的频次与$X$中的$|A(u_i)|$出现的频次也相等.

	uid	*aid*				
X	u_1	a_1	a_2	a_3	a_4	a_5
	u_2	a_1	a_2	a_3	a_4	a_5
	u_3	a_1	a_2	a_3		
	u_4	a_1	a_4			
	u_5	a_1	a_3	a_5		
	u_6	a_5				

转置

	aid	*uid*				
X'	a_1	u_1	u_2	u_3	u_4	u_5
	a_2	u_1	u_2	u_3		
	a_3	u_1	u_2	u_3	u_5	
	a_4	u_1	u_2	u_4		
	a_5	u_1	u_2	u_5	u_6	

Fig. 1 Transpose of sample data

图 1 示例数据转置

1.1 FP-tree 及其派生结构

FP-tree 是一种扩展前缀树结构，该树保留了所有项集的关联信息，基于 FP-tree 可以递归地挖掘出所有的频繁项集[25]. FP-tree 数据结构模型由 2 部分组成：头表和树结构. 头表由三元组表示(元素，频次，头指针)，其中“头指针”指向树结构中元素相同的第 1 个结点，头表中的三元组按照频次递减排序. 树结构结点由六元组表示(元素，频次，计数，父结点指针，子结点指针，下一个元素相同结点的指针)，其中“计数”指的是共同前缀出现频次的计数，“下一个元素相同结点的指针”用于构建索引链，把树中具有相同元素的结点链接起来.

构建 FP-tree 包括构建头表H和树结构. X'中任一应用a_i的用户集合标记为$U(a_i)$，$U(a_i)$元素按照频次降序排序的序列标记为$S(a_i)$. 首先初始化头表和树结构，将数据集中所有u_i及其一项频次f_i按照频次倒序加入头表中，相应头指针$L(u_i)$初始化为空指针，即索引链为空；而树结构则初始化为元素为空的根结

点. 然后将 X'中任一应用a_i相应的序列$S(a_i)$插入头表和树结构中. 对任一序列$S(a_i)$, 假设当前始于根且与$S(a_i)$有最长共同前缀的路径为P, 则给共同前缀上所有结点的计数都加 1, 将$S(a_i)$中不在共同前缀中的元素依序插入树中, 同时结点u_i链接到对应元素的索引链的末尾. 基于图 1 示例数据构建的 FP-tree 见图 2 (a). 为了简化图 2 表示, 树结点信息仅展示其中的 3 项, 并以三元组格式展示(元素, 频次, 计数).

构建好 FP-tree 后, 简单的集合相似度自连接可以通过以下流程完成: 首先根据头表结点索引链遍历树并统计所有包含这些结点元素的二项集出现频次, 即相应集合交集大小; 然后验证 2 个集合相似度是否大于阈值; 最后输出大于阈值的集合对. 其中统计树二项集的频次时取二项计数值中较小者累加. 例如, 从头表的u_3开始, 索引项的第 1 个树结点是$(u_3, 3, 3)$, 沿分支向根结点方向遍历并统计包含u_3的二项集的频次. 其中$(u_3, 3, 3; u_1, 5, 5)$的二项计数值较小者为 3, $(u_3, 3, 3; u_1, 5, 5)$的计算值为 3, 则$f_{3,1}$暂时记为 3. 向上扫描完第 1 个分支后, 发现u_3索引链只有 1 个结点, 则$f_{3,1}=3$. 即$|A(u_3)\cap A(u_1)|=3$. 而$|A(u_3)|=3$, $|A(u_1)|=5$, 根据式(1)得$sim(A(u_3), A(u_1))=0.6$, 满足阈值要求, 输出集合对(u_1, u_3). 而从u_4开始, 索引项的第 1 个树结点是$(u_4, 2, 1)$, 沿分支向根结点方向遍历得到$(u_4, 2, 1; u_1, 5, 5)$的计算值为 1, 则$f_{4,1}$暂时记为 1. 向上扫描完第 1 个分支后, 沿着u_4索引链找到 FP-tree 的下一个结点$(u_4, 2, 1)$, 同理得到$(u_4, 2, 1; u_1, 5, 5)$的计算值为 1, 则$f_{4,1}=f_{4,1}+1=2$, 即$|A(u_4)\cap A(u_1)|=2$. 而$A(u_4)=2$, $A(u_1)=5$, 根据式(1)得$sim(A(u_4), A(u_1))=0.4<0.6$, 不输出集合对$(u_1, u_4)$.

由于基于 FP-tree 的多项集频次计算或频繁多项集挖掘需要递归建树, 运行期开销大. 为减少这一开销, N-lists[28] 采用垂直数据结构表示 FP-tree 树结点, FP-tree 树结点的构建过程包含 2 步: 1)构建一种类似 FP-tree 的树结构-前序后序编码树(pre-order post-

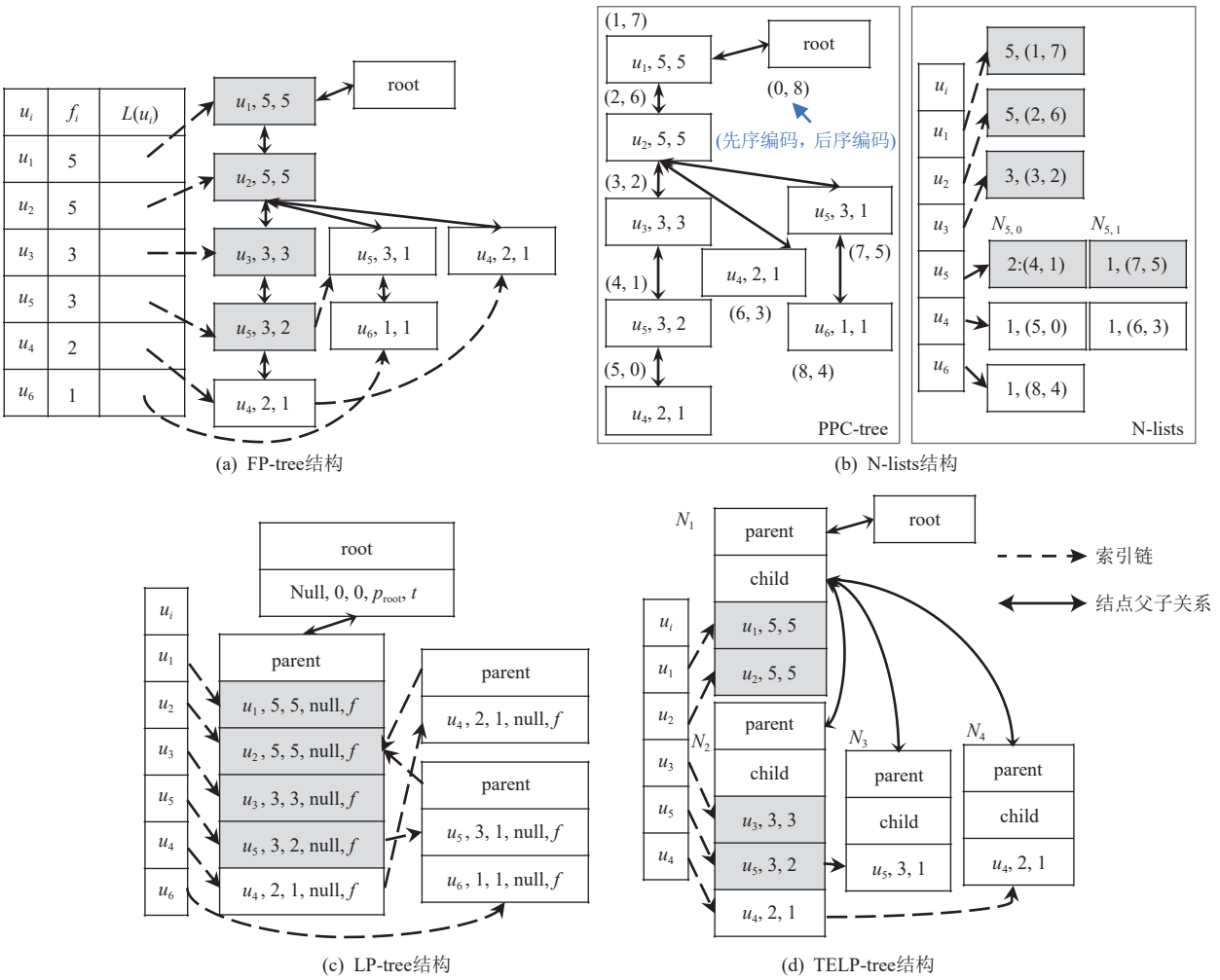


Fig. 2 Construction of the FP-tree, N-lists, LP-tree, and TELP-tree over the sample data

图 2 基于示例数据构建的 FP-tree, N-lists, LP-tree, TELP-tree

order code tree, PPC-tree)；2)构建 PPC-tree 结点列表 N-list. 如图 2(b)所示，PPC-tree 基本与 FP-tree 结构相似，不同之处在于 PPC-tree 不包含索引链，同时每个树结点增加了标识结点在先序树遍历和后序树遍历中位置的先序编码和后序编码. PPC-tree 结点旁边括号中的数字，左边是先序编码，右边是后序编码. 结合 2 个结点的先序编码、后序编码可快速判断 2 个结点之间的祖先后代关系：祖先结点的先序编码大且后序编码小. 为此，PPC-tree 的建树过程增加了 2 次树的遍历：先序遍历和后序遍历. N-lists 是由 PPC-tree 中元素相同的结点按先序编码升序排序的列表，其数据结构包含头表和元素的结点列表 N-list. 头表结构及内容与图 2(a)类似，只是头指针指向相应元素的 N-list 首地址. 本文的图 2(b)中只画出头表中的元素信息部分. N-lists 的结点数据结构为三元组：(计数，(先序编码，后序编码)). 头表元素 u_i的 N-list 的第 1 个结点表项记为 $N_{i,0}$，第 2 个结点表项记为 $N_{i,1}$，依此类推. 如u_5的第 1 个结点 $N_{5,0}=(2,(4,1))$. 特别地，头表元素的 N-list 各项计数之和为元素的单项频次，例如，u_5的 N-list 中 2 项元组的计数之和为 $N_{5,0}$.计数 + $N_{5,1}$.计数 = 2 + 1 = 3 = f_5.

假定二项集(u_i, u_j)都是排序的二项集，u_i的先序编码小于u_j，则(u_i, u_j)的 N-list 也是按先序编码升序排序的列表，可通过将u_i与u_j的 N-lists 按规则相交合并而成：分别取u_i和u_j的 N-lists 上的任意一个结点 $N_{i,k}$和$N_{j,l}$,如果结点之间存在祖先后代关系，则将$(N_{j,l}$.计数, $(N_{i,k}$.先序编码, $N_{i,k}$.后序编码))插入(u_i, u_j)的 N-list 中. 例如，u_3和u_5对应的 N-lists 相交合并可生成(u_3, u_5)的 N-list，首先u_3的 N-list 只包含 1 个结点 $N_{3,0}=(3,(3,2))$，u_5的 N-list 包含 2 个三元组 $N_{5,0}=(2,(4,1))$, $N_{5,1}=(1,(7,5))$. 当比较 $N_{3,0}$和 $N_{5,0}$时，由于 $3<4$且 $2>1$，$N_{3,0}$是 $N_{5,0}$的祖先，即它们间有祖先后代关系，则向(u_3, u_5)的 N-list 中插入 1 个结点 $N_{(3,5),0}=(N_{5,0}$.计数, $(N_{3,0}$.先序编码, $N_{3,0}$.后序编码)) = (2, (3, 2))；当比较 $N_{3,0}$和 $N_{5,1}$时，由于 $3<7$且 $2<5$，它们之间没有祖先后代关系，因此不必向(u_3, u_5)的 N-list 插入新的结点. 同理，频次 $f_{i,j}$的计算可以通过对(u_i, u_j)的 N-list 各项计数求和得到，如 $f_{3,5}=2$，即 $|A(u_3)\cap A(u_5)|=2$，而 $|A(u_3)|=3$，$|A(u_5)|=3$，根据式(1)得 $sim(A(u_3), A(u_5))=0.5$，不满足阈值要求，不输出集合对(u_3, u_5).

基于 N-lists 的多项集频次计算是基于 N-lists 进行交叉合并的，不需要递归建树，提升了计算性能，但在只计算二项频次时转变为 2 个 N-lists 的所有元素两两对比的计算，性能反而不理想. 而面向大规模集合构建的 FP-tree 的树结点数量多，指针结点的寻址时间长，缓存访问命中率低，降低了遍历树的效率. 线性前缀树(linear prefix tree, LP-tree)[29]，在 FP-tree 结构的基础上将其结点的元素字段改用数组表示，存储多个元素信息，减少内存使用并加快寻址. 如图 2(c)所示，LP-tree 的头表结构及内容与图 2(a)完全一样，所以本文只画出头表中的元素信息部分. 而结点信息改为 2 项内容：一个是存储元素信息的数组，另一个是父结点. 该数组每一项的数据域用五元组表示(元素，频次，计数，下一个元素相同结点的指针，分支标志). 其中，“下一个元素相同结点的指针”用于构建索引链，指向下一个元素相同的项在另一个结点的数组中的位置. 父结点指向其前缀最后一个元素所在的结点及其在数组的位置. 其中，当一个元素没有下一个元素相同的结点时，相应指针为Null. 分支标志为t时表示有大于等于 2 个以上的序列片段的最长共同前缀止于该元素，f表示不存在分支. LP-tree 结点结构合并了多个元素结点的信息，减少了 FP-tree 树结点数量，图 2(a)中的 FP-tree 有 9 个结点，而图 2(c)中的 LP-tree 只有 4 个结点. LP-tree 通过连续存储元素信息提高了计算的效率，但结点内部存在的分支、复杂的遍历流程带来运行期开销.

1.2 TELP-tree 数据结构

为进一步提高基于 FP-tree*的集合相似度自连接的计算效率，我们提出了一种遍历效率更高的线性前缀树 TELP-tree(traversal efficient LP-tree)的数据结构. 与线性前缀树 LP-tree 相似，TELP-tree 的树结点也采用线性数组存储多个元素信息以减少树结点数量，提高树的遍历效率；而不同的是，TELP-tree 树结点内部没有分支，避免结点内分支带来复杂的运行期遍历流程管理，从而减少相应的运行期开销.

定义 1. TELP-tree. TELP-tree，数据结构包含头表H和树结构，头表H与 FP-tree 一样结构. TELP-tree 树结点(TELP-tree node)N包含建树和遍历树所需的 3 项内容：父结点 P、子结点 C 和存储元素信息的数组 E，即一个包含k个元素的 N_i表示为 $N_i=\{P, C, E=\{E_1, E_2, \cdots, E_k\}\}$. 而数组元素$E_j$可用三元组(元素，频次，计数)表示. 一个由$c$个树结点组成的 TELP-tree 可以表示为$\{H, N_1, N_2, \cdots, N_c\}$.

TELP-tree 的头表与 FP-tree 一样，因此，构建 TELP-tree 头表与构建 FP-tree 的方式是一样的. 而树结点间存在差异，因此 TELP-tree 树结构的构建与 FP-tree, LP-tree 的构建也存在差别. 对任一序列$S(a_i)$，与 FP-tree, LP-tree 相似，始于根结点，找到 TELP-tree 中

与$S(a_i)$有最长共同前缀的路径P，将P相应共同前缀上结点中所有元素的计数加 1，然后将这些元素从$S(a_i)$中移去，形成$S(a_i)'$. 如果$|S(a_i)'|=0$，对该序列的处理完成. 否则，设P上最后 1 个结点中与$S(a_i)'$有最长公共前缀的后继结点为N_{succ}，根据该公共前缀长度是否为 0，树结点的构建和树结点插入到树结构有所不同：1）长度为 0 时，构建一个新的结点，将$S(a_i)'$中的元素依序存进其数组中，并将该结点插入树结构和链接到对应元素的索引链的末尾，这种情况与 LP-tree 的树结构构建是一致的；2）长度不为 0 时，将N_{succ}所有公共前缀上的元素计数加 1. 然后将$S(a_i)'$中在公共前缀上的元素移除，形成$S(a_i)''$，如果$|S(a_i)''|=0$，对该序列的处理完成，这种情况与 LP-tree 的树结构构建是一致的. 如果$|S(a_i)''|=0$，将构建一个新的结点N'，将$S(a_i)''$中的元素都依序存进其数组中. 同时将N_{succ}中不是公共前缀的元素都移出，重构一个新的结点N''. 最后将新构建的结点N'和N''插入树中，即N_{succ}是N'和N''的父结点，并将N'和N''分别链接到对应元素的索引链的末尾. 这一步是 TELP-tree 与 LP-tree 因为树结点结构不同带来的树结构构建的不同之处，结点内部没有分支，简化了树结点的同时也简化了树结构的构建.

基于图 1 数据构建的 TELP-tree 如图 2(d)所示，其中$(u_1,5;u_2,5)$是所有输入序列的公共前缀，基于构建树的规则，$(u_1,5)$和$(u_2,5)$将分别成为 TELP-tree 的 1 个结点里的 2 个数组元素. 而序列$(u_1,5;u_2,5;u_3,3;u_5,3;u_4,2)$移去公共前缀$(u_1,5;u_2,5)$之后的后续片段为$(u_3,3;u_5,3;u_4,2)$，$(u_1,5;u_2,5;u_3,3;u_5,3)$移除公共前缀$(u_1,5;u_2,5)$后是$(u_3,3;u_5,3)$，$(u_1,5;u_2,5;u_3,3)$移除公共前缀$(u_1,5;u_2,5)$后是$(u_3,3)$，$(u_3,3;u_5,3)$是$(u_3,3;u_5,3;u_4,2)$的子序列，而$(u_5,3)$和$(u_4,2)$没有公共前缀，所以$(u_1,5)$，$(u_2,5)$，$(u_3,3)$这 3 个数组元素组成 1 个结点，计数值分别是 3，2，1，且结点内部没有分支. 最终构建的 TELP-tree 有 5 个结点.

相比图 2(a)中 9 个结点的 FP-tree，图 2(d)中 5 个结点的 TELP-tree 结点少 4 个，结点内数组元素寻址快. 相比图 2(c)的 4 个结点的 LP-tree，TELP-tree 的结点多了 1 个，但结点数组中存储的信息少，且前缀所在结点直接指向后续片段所在结点，避免了结点内部分支，简化并加速树的遍历.

1.3 TELP-SJ 算法

1.1 节描述了简单的基于 FP-tree*的集合相似度自连接计算流程，但该计算过程没有采用过滤-验证框架. 为进一步提高基于 FP-tree*的集合相似度自连接计算效率，本节设计基于过滤-验证框架和 TELP-tree 的集合相似度自连接算法 TELP-SJ.

TELP-SJ 使用长度过滤策略来设计其过滤算法. 长度过滤策略被应用于多个集合相似度自连接算法中，如 FS-Join. 当集合相似度函数为 *Jaccard*()时，长度过滤策略如定理 1[19] 所示.

定理 1. 给定 2 个集合A和B，$|A|<|B|$，阈值t，如果$|A|<t\times|B|$，那么$sim(A,B)<t$.

文献 [19] 给出定理 1 的数学证明，基于定理 1，2 个相似的集合应有相似的大小，反之，集合大小不相似的集合对的相似度低. 与集合A的相似度大于等于t的所有集合，其集合大小一定小于等于$|A|/t$. 因此，可提前过滤集合大小大于$|A|/t$的集合. 根据 1.1 节和 1.2 节的建树过程可知，序列的长度将影响树的深度，长度越大，深度越深，树的遍历代价就越高. 为减小构建的树的规模，根据定理 1，我们首先设计面向构建树的过滤算法，该算法检测、发现并删除序列中不会与其他集合形成相似度高于阈值的集合对的集合. 给定至少有 2 个元组的序列$S(a_i)$，在将它插入到 TELP-tree 时前先应用算法 1 描述的面向构建树的过滤算法对其进行预处理：从右向左扫描$S(a_i)$中的元素，假设最右边的 2 个相邻元素分别是u_k和u_j，如果$|A(u_k)|>t\times|A(u_j)|$，将$S(a_i)$按照 2.2 节描述的建树过程插入 TELP-tree. 如果$|A(u_k)|<t\times|A(u_j)|$，根据定理 1，我们有$sim(A(u_k),A(u_j))<t$，即u_k与u_j的相似度低于阈值t. 序列片段按元素频次降序排列，此时，u_k对应的元组$(u_k,A(u_k))$将从序列$S(a_i)$中删除. 删除后如果$|S(a_i)|\leqslant 1$，意味着a_i对应原序列中所有元素相似度都低于阈值t，原序列被舍弃，否则继续如上述过程迭代处理$S(a_i)$. 面向构建树的过滤算法如算法 1 所示. 函数$getrightmost(S(a_i))$表示获取$S(a_i)$的最右边元素.

算法 1. 面向构建树的过滤算法.

输入：序列$S(a_i)$；

输出：过滤后序列$S'(a_i)$.

① $S'(a_i)\leftarrow S(a_i)$;

② while $|S'(a_i)|>1$ do

③ $(u_k,A(u_k))\leftarrow getrightmost(S'(a_i))$;

④ $S''(a_i)\leftarrow S'(a_i)-(u_k,A(u_k))$;

⑤ $(u_j,A(u_j))\leftarrow getrightmost(S''(a_i))$;

⑥ if $|A(u_k)|\geqslant t\times|A(u_j)|$

⑦ break;

⑧ else

⑨ $S'(a_i)\leftarrow S''(a_i)$;

⑩ end if

⑪ end while

⑫ return $S'(a_i)$.

例如，针对图 1 的示例数据，$S(a_5)=(u_1,5;u_2,5;u_5,3;u_6,1)$，从右向左扫描$S(a_5)$，最右边的$|A(u_6)|=1$，而过滤算法根据构建树(算法 1)，构建树的过程就是逐步插入树节点的过程$|A(u_5)|=3$，$1/3<0.6$，根据面向构建树的过滤算法删除元组$(u_6,1)$，得到$S'(a_5)=(u_1,5;u_2,5;u_5,3)$. 此时最右边的$|A(u_5)|=3$，而$|A(u_2)|=5$，$3/5=0.6$，将$S'(a_5)$插入到 TELP-tree. 应用面向构建树的过滤算法构建的 TELP-tree 如图 3 所示，与图 2(d)中没有应用该过滤算法构建的 TELP-tree 相比，图 3 中的 TELP-tree 减少了 1 个树结点N_4，树的结点N_3的数组元素减少了 1 个，头表元素$|H|$也减少了 1 个，即少了u_6相应的信息.

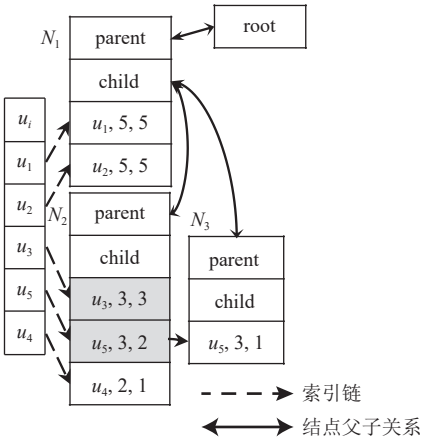


Fig. 3 A TELP-tree is constructed based on filtering algorithm

图 3 基于过滤算法构建的 TELP-tree

构建好 TELP-tree 后，通过沿着头表结点索引链遍历树并统计所有以这些结点为后缀的二项集频次，根据树的特征，树结点元素频次越高越靠近根结点. 树结点内数组元素按频次倒序排序. 树的遍历代价跟遍历的结点数和结点内数据元素的个数有关. 沿着索引链从远离根结点的树结点向着根结点的方向遍历树，统计索引链结点元素与其他元素的二项频次. 同样，根据定理 1，非根结点元素u_i及其祖先结点元素u_j如果满足$|A(u_i)|<t\times|A(u_j)|$，可得$sim(A(u_i),A(u_j))<t$，即这 2 个元素相似度低于阈值，此时，可以停止沿着这条分支向着根结点方向统计，即过滤对应的集合对以减少遍历. 面向遍历树的过滤算法见算法 2，其中，$Ancestor(N)$表示树中N的祖先结点；$N.x$中的“.”表示取结点N的相应数据域x，如$N.E[l]$表示取结点N的元素数组E的第l个元素.

算法 2. 面向遍历树的过滤算法.

输入：元素u_i，树结点N，频次统计f；

输出：频次统计f.

① while N 不为根结点 do

② $l\leftarrow|N.E|-1$;

③ while $l\geqslant 0$ do

④ if $|A(u_i)|<t\times|A(N.E[l])|$

⑤ goto ⑫;

⑥ else

⑦ $f_{i,N.E[l]}+=N.E[l]$.计数

⑧ endif

⑨ end while

⑩ $N\leftarrow Ancestor(N)$;

⑪ end while

⑫ return f.

基于面向构建树的过滤算法构建 TELP-tree，按规则遍历树并计算集合相似度自连接：1)遍历头表H中所有元素u_i，并初始化f；2)通过u_i的索引链，遍历其所有树结点，对每个树结点调用面向遍历树的过滤算法，并更新f；3)根据f计算出集合相似性，并输出相似性大于阈值的集合对. 例如从图 3 的头表的$(u_4,2)$开始，顺着索引项第 1 个树结点N_2的数组元素$(u_4,2,1)$开始统计包含u_4的二项集的频次. 如$(u_4,2,1;u_5,3,2)$的二项计数值较小者为 1，则$f_{4,5}=0+1=1$. 同时注意到$|A(u_4)|=2$，$|A(u_2)|=5$，$2<0.6\times5$，因此，针对$(u_4,2,1)$，从N_2向着根结点的方向遍历树时，只需要遍历N_2的元素信息数组，不需要遍历N_1. 因此$f_{4,5}=1$，即$|A(u_4)\cap A(u_5)|=1$. 而$|A(u_4)|=2$，$|A(u_5)|=3$，根据式(1)得$sim(A(u_4),A(u_5))=0.25<0.6$，不输出集合对$(u_4,u_5)$. 图 2 和图 3 中灰色框的元素表示统计包含$u_4$的二项集频次时要遍历的元素，没有填充灰色的框的结点和数组元素表示没有遍历的点或元素. 由此，针对 TELP-tree，统计包含u_4的二项集的频次要遍历的结点数和数组元素分别是 1 和 2，而在使用过滤算法的情况下，要遍历的结点数和数组元素的分别是 3 和 6，遍历次数大大减少，提高了遍历效率.

以上面向构建树和树遍历的 2 个过滤算法同样可以使用于其他基于 FP-tree*的集合相似度自连接算法. 综上，TELP-SJ 的算法的伪代码见算法 3. 基于 FP-tree 的集合相似度自连接算法 FP-SJ、基于 LP-tree 的 LP-SJ 均与 TELP-SJ 的计算思路基本一致，不同之处在于算法 3 中行⑪，即：如果是 FP-SJ，则构建 FP-tree；如果是 LP-SJ，则构建 LP-tree. 遍历树的一些细节有稍许区别，如$Ancestor(n_k)$的实现和结点的定位等操作，这里不赘述. 而基于 N-lists 的集合相似度自

连接算法 PPC-SJ, 则是构建 PPC-tree 后, 构造 N-lists, 然后对任意 2 个 N-lists 的所有元素两两对比验证完成计算. FP-SJ, LP-SJ, TELP-SJ, PPC-SJ 这 4 种基于 FP-tree*的集合相似度自连接算法统称为 FP-tree*-SJ.

算法 3. TELP-SJ 算法.

输入: 数据集$D=\{\langle u_i,\{a_{i1},a_{i2},\cdots,a_{ik}\}\rangle\}$, 阈值$t$;

输出: 满足阈值的集合对集合P.

① $U\leftarrow\varnothing$, $S\leftarrow\varnothing$, $S'\leftarrow\varnothing$, $P\leftarrow\varnothing$;
② for each u_i, $A(u_i)$ in D
③ for each $a_{ij}\in A(u_i)$
④ $U(a_{ij})\leftarrow U(a_{ij})+(u_i:|A(u_i)|)$;
⑤ end for
⑥ end for
⑦ for $U(a_i)$ in U
⑧ 对$U(a_i)$按频次降序排序得到$S(a_i)$;
⑨ 对$S(a_i)$调用算法 1 得到$S'(a_i)$;
⑩ end for
⑪ 根据S'建树$tree$;
⑫ for each $u_i\in tree.H$
⑬ 初始化频次 f;
⑭ for each N_k in u_i 的索引链
⑮ 对u_i, N_k, f调用算法 2;
⑯ end for
⑰ 根据f计算出$sim(u_i,u_j)$;
⑱ $P\leftarrow P+\{(u_i,u_j)\text{s.t.}sim(u_i,u_j)>t\}$;
⑲ end for
⑳ return P.

2 基于 MapReduce 的集合相似度自连接算法 FastTELP-SJ

MapReduce 是一种高效的并行分布式集群计算框架. 一个 MapReduce 任务的运行主要包含 2 个阶段: 映射(Map)阶段和规约(Reduce)阶段. Map 阶段首先将存储在分布式文件系统 HDFS 上的输入数据划分为多个分片; 然后就近分发至集群多个计算节点独立并行运行函数 *map*, 处理数据并以〈*Key,Value*〉对的形式输出中间结果. 之后中间结果按照 *Key* 值进行合并(combine)、重新分配(shuffle)、排序(sort)后再分发至 1 个或多个节点进行 Reduce 阶段运算, 独立并行运行函数 *reduce* 进行汇总. 最终各个函数 *reduce* 的结果仍以〈*Key,Value*〉对的方式输出并保存在 HDFS 上.

第 1 节设计了系列基于 FP-tree 及其扩展结构的集合相似度自连接算法 FP-tree*-SJ, 这些算法的特征是建树后, 整棵树需要保留在内存中进行计算, 当集合规模增大时, 树的规模增大, 对内存的要求增大, 树的遍历计算代价也增大. 为进一步提高计算性能, 本节设计基于 TELP-tree 和 MapReduce 的集合相似度自连接算法 FastTELP-SJ. 图 4 展示了 FastTELP-SJ 的计算过程. FastTELP-SJ 包含 2 个 MapReduce 任务, 分别完成数据集转置、建树和集合相似度自连接的计算. 1.3 节设计的 2 阶段过滤算法, 分别应用于任务 1 和任务 2.

2.1 转置数据集

任务 1 的函数 *map* 计算每个u_i的频次$|A(u_i)|$、输出$\langle Key=aid, Value=(u_i:|A(u_i)|)\rangle$对, 实现了原数据集的转置. 与*Key*值相同的*Value*将被合并和发送到同一个函数 *reduce*. 任一应用a_i的*Value*集合标记为$U(a_i)$. 函数 reduce 将$U(a_i)$元素按照频次降序排序得到序列$S(a_i)$, 调用算法 1 得到$S'(a_i)$, 输出键值对$\langle Key=a_i, Value=S'(a_i)\rangle$. 例如, 图 4 任务 1 的 Reduce 阶段中$S(a_5)=(u_1,5;u_2,5;u_5,3;u_6,1)$, 函数 *reduce* 对$S(a_5)$应用算法 1, 从右向左扫描$S(a_5)$, 最右边的$|A(u_6)|=1$, 而$|A(u_5)|=3$, $1/3<0.6$, 删除数据$(u_6,1)$. 同时$|A(u_2)|=5$, $3/5\geqslant 0.6$, 因此输出结果$\langle a_5,(u_1,5;u_2,5;u_5,3)\rangle$. 任务 1 的函数 *map* 与 *reduce* 的功能分别与算法 3 中行②~⑥和行⑦~⑩类似, 故不再赘述.

当所有 *reduce* 函数完成计算后, 节点 *master* 将启动一个独立进程对元素分组, 按照频次将元素排序, 函数$seq(u_i)$表示u_i的序号. 假设$gNum$表示组数, g_i表示u_i的组标识号gid. 基于 FP-tree 树的并行分布式计算有多种分组策略[30], 这里采用基于均衡预估遍历负载的策略, 即$g_i=seq(u_i)\bmod gNum$.

2.2 构建树并计算集合相似度自连接

FastTELP-SJ 采用集合数据分段过滤与验证的设计思路, 为此, 任务 2 首先切分序列, 按分段划分数据集, 然后根据数据集分片并行独立构建 TELP-tree 计算集合相似度自连接. 其中函数 *map* 完成数据集划分工作. 对任一$S(a_i)$, 函数 *map* 的工作流程为: 1)根据$S(a_i)$最右边元素*gid*输出键值对$\langle Key=gid, Value=S(a_i)\rangle$; 2)从右向左遍历$S(a_i)$中的元素$u_k$, 如果与*gid*即$g_k$相等的键值对已输出, 则将该$u_k$从$S(a_i)$中删去; 如果$S(a_i)$不为空, 跳回 1)继续迭代执行. 例如, 图 4 任务 2 的函数 *map* 对序列$S(a_5)=(u_1,5;u_2,5;u_5,3)$将输出 2 个结果: $\langle 1,(u_1,5;u_2,5)\rangle$和$\langle 0,(u_1,5;u_2,5;u_5,3)\rangle$. 任务 2 中切分序列的伪代码如算法 4 所示.

算法 4. 序列切分算法.

输入：已排序序列$S(a_i)$，元素分组gid；

输出：划分结果$R=\{\langle gid, S\rangle\}$.

① $R\leftarrow\varnothing, G\leftarrow\varnothing$；

② while $|S(a_i)|>0$

③ $(u_{ik}, |A(u_{ik})|)\leftarrow getrightmost(S(a_i))$；

④ if $g_{ik}\in G$

⑤ $R\leftarrow R\cup\{\langle g_{ik}, S(a_i)\rangle\}$；

⑥ $G\leftarrow G\cup\{g_{ik}\}$；

⑦ end if

⑧ $S(a_i)\leftarrow S(a_i)-(u_{ik}, |A(u_{ik})|)$；

⑨ end while

⑩ return R.

所有*Key*即*gid*相同的序列片段将被分配给同一个节点的函数*reduce*. 因为集合相似度自连接仅需要计算单项频次和二项频次，结合面向MapReduce计算而设计的精简FP-tree(reduced FP-tree)[29]的结构特征，函数*reduce*在构建TELP-tree时，仅为*gid*与*Key*值相同的元素构建头表和索引链，减少内存空间并减少建树时间，相应的TELP-tree称为精简TELP-tree. 除此之外，树的构建过程与1.2节相同.

构建好精简TELP-tree后，通过对头表结点索引链应用面向遍历树的过滤规则遍历树，并统计所有以这些结点为后缀的二项集出现频次. 其中树遍历过程与1.3节描述的相同，而二项集频次统计仅对*gid*与*reduce*的*Key*值相同的结点进行统计. 任务2的函数*reduce*与算法4的行⑪~⑳类似，故不再赘述.

综上，FastTELP-SJ算法的计算流程示例见图4. 其他FP-tree*-SJ的MapReduce计算可采用类似的设计，相应地，基于MapReduce的FP-tree*-SJ算法被命名为FastFP-tree*-SJ.

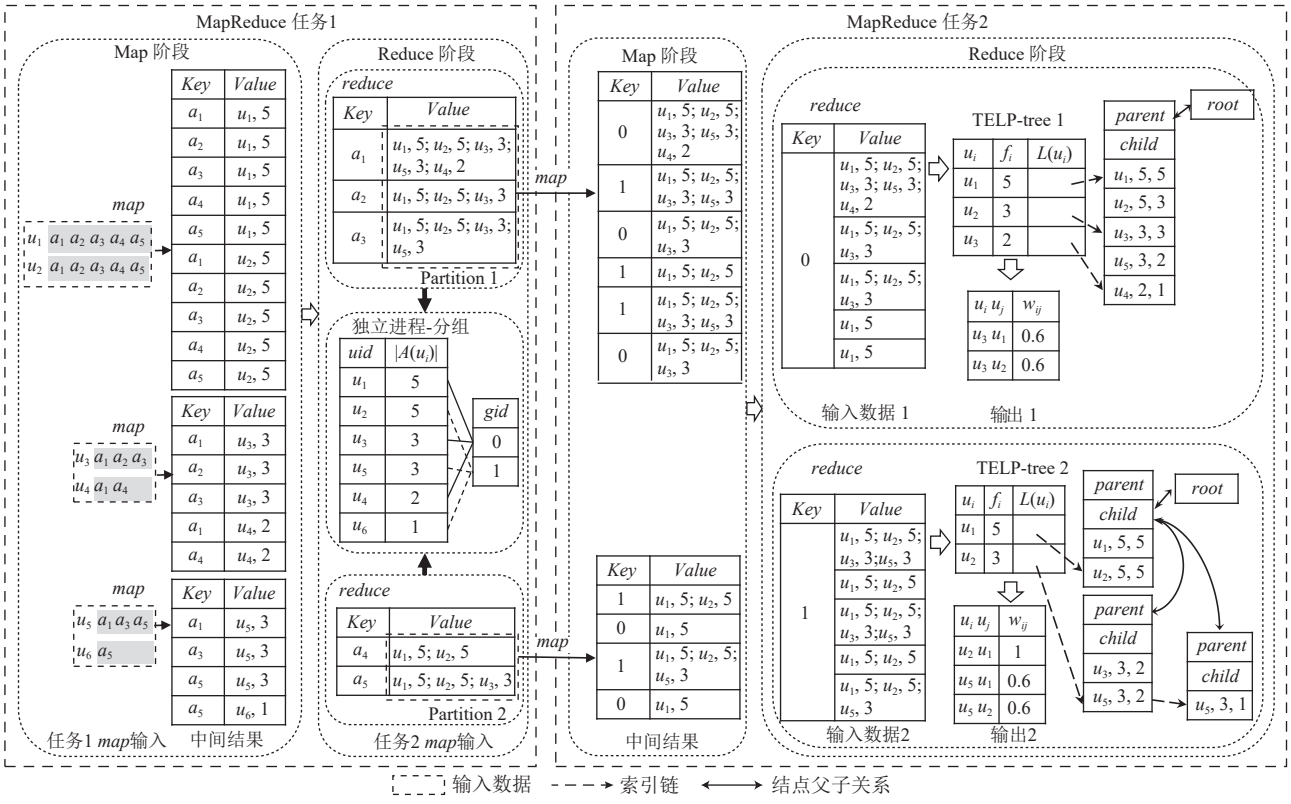


Fig. 4 An illustration of the computation process of the parallel and distributed set similarity self-join of FastTELP-SJ with MapReduce

图4 基于MapReduce的并行分布式集合相似度自连接FastTELP-SJ的计算流程示意图

3 性能评估

基于FP-tree*的算法通过将数据压缩在内存中进行计算，减少I/O操作，减小候选集，提高验证两两集合相似度是否大于阈值的运行效率. 但这些算法也增加了动态构建树等运行期开销，同时，整棵树需被保留在内存中进行计算，对内存需求比较大. 我们将从运行时间、内存占用率、磁盘使用量这3类指标通过3组比较实验评估FastTELP-SJ算法的性能：1) TELP-SJ算法与基于其他FP-tree*-SJ算法的比较；2) FastTELP-SJ算法与TELP-SJ算法的比较；3) FastTELP-

SJ 与现有经典算法的比较.

Apache Hadoop[31] 和 Apache Spark[32] 是 2 个最知名和最广泛使用的 MapReduce 编程模型的开源实现. 两者最大的区别是：Apache Hadoop 的中间计算结果保存在 Apache Hadoop 分布式文件系统中，而 Apache Spark 可将中间数据结果缓存在内存中. 因此，对多次数据依赖的迭代算法、多数据依赖的计算任务，使用 Apache Spark 可以减少 I/O 从而提高运行效率. 从第 2 节的描述可见，集合相似度自连接计算不包含多次数据依赖的迭代计算. 另外，还有些特定面向多核或众核优化的 MapReduce 实现，如 Phoenix[33], Mars[34] 等. 考虑到现有经典算法都是基于 Apache Hadoop 实现，因此，本文的对比实验首先在 Apache Hadoop 上复现经典算法，然后实现本文设计的算法.

实验使用一个包含 17 个曙光 TC4600 刀片节点搭建的 Apache Hadoop 集群，其中 1 个作为主节点，其他作为从节点. 每个刀片都有 2 个 10 核的 Intel® Xeon® E5-2 680 v2 2.8 GHz 的处理器、2 个 1 TB 的磁盘、64 GB 的内存和 1 Gbps 的以太网. 刀片上安装 CentOS 6.5 操作系统，OpenJDK 1.8.0 和 Apache Hadoop 2.7.1. 相关 Apache Hadoop 参数配置见表 1，其他均采用缺省配置.

实验数据来自 4 组真实的应用数据集：Facebook 应用数据集 II(简记为 facebook)[35]、匈牙利在线新闻门户点击流数据集 kosarak[36]、2015 年 Enron 电子邮件数据集 Enron[37] 和交通事故数据集 accidents[38]. 我们将从各组数据集中提取一定量的数据分别进行实验，相应数据子集命名为数据集名称后加提取的数量，如从 facebook 数据集中提取 10 万条数据，记为 facebook10w. 4 组数据集为各取近 30 万条数据组成的子集，其相应数据特征总结在表 2 中，其中，子集 $x'=\{x'_1, x'_2, \cdots, x'_{n'}\}$，子集大小即 $|x'|=n'$ 为数据子集元素总数，$A(x')=A\left(\bigcup_{i=1}^{n'} x'_i\right)=\{a'_1, a'_2, \cdots, a'_{m'}\}$ 表示数据集所有元素属性合集，$X(a'_i)$ 表示拥有属性 a'_i 的元素，$\max\limits_{1\leqslant i\leqslant m'}(|X(a'_i)|)$ 表示拥有该属性的元素的最大个数，$avg\limits_{1\leqslant i\leqslant m'}(|X(a'_i)|)$ 表示所有属性元素的平均个数，$\max\limits_{1\leqslant i\leqslant n'}(|A(x'_i)|)$ 表示元素拥有属性的最大个数，$avg\limits_{1\leqslant i\leqslant n'}(|A(x'_i)|)$ 表示元素拥有属性的平均个数，$dev\limits_{1\leqslant i\leqslant n'}(|A(x'_i)|)$ 表示元素拥有属性个数的方差. 每次计算都将运行 3 次，采用平均运行时间、内存占用率峰值和磁盘使用量用于展示实验结果和分析.

3.1 TELP-SJ 与基于其他 FP-tree*-SJ 的比较

本文分别实现 FP-SJ, PPC-SJ, LP-SJ, TELP-SJ 这 4 种基于 FP-tree 及其派生结构的集合相似度自连接算法，并将这些算法分别运行在 facebook30w, kosarak30w, Enron30w, accidents30w 这 4 组数据上，阈值 t 的取值范围为0.6 ~ 0.9. 基于 FP-tree*-SJ 的集合相似度自连接计算没有产生中间结果，最后输出结果是一样的，因此这 4 种算法的磁盘使用量没有差异. 这 4 组算法的区别在于树结构的差异及因此带来的建树和遍历树流程的差异，我们将比较这 4 种算法的运行时间和内存占用率峰值.

1）运行时间

Table 1 Apache Hadoop Parameter Configuration

表 1 Apache Hadoop 参数配置

参数	取值	描述
mapreduce.map.java.opts/MB	4 096	Java 虚拟机启动参数，为函数 *map* 设置最大堆内存大小
mapreduce.reduce.java.opts/MB	20 240	Java 虚拟机启动参数，为函数 *reduce* 设置最大堆内存大小
yarn.scheduler.maximum-allocation-mb/MB	40 000	单个任务可申请的最大物理内存量
yarn.scheduler.minimum-allocation-mb/MB	5 120	单个任务可申请的最小物理内存量
yarn.nodemanager.resource.memory-mb/MB	41 440	节点最大可用物理内存总量
yarn.nodemanager.resource.cpu-vcores	20	该节点上 YARN 可使用的虚拟 CPU 个数

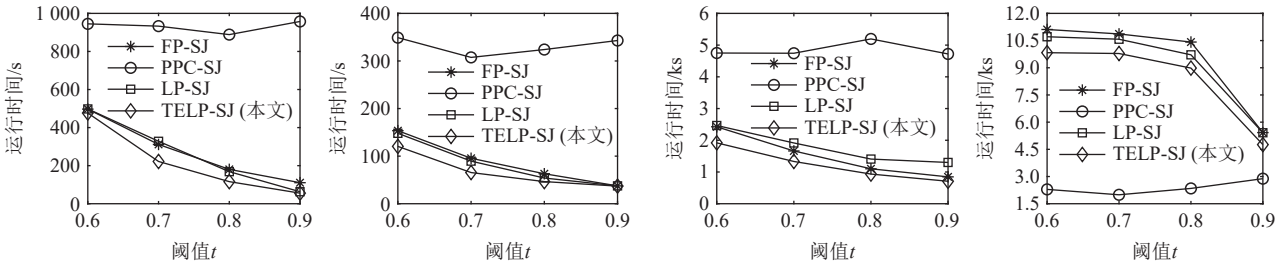
Table 2 Characteristics of Experimental Datasets

表 2 实验数据集特征

数据集	$\lvert x'\rvert$	$\lvert(Ax')\rvert$	$\max\limits_{1\leqslant i\leqslant m'}(\lvert X(a'_i)\rvert)$	$avg\limits_{1\leqslant i\leqslant m'}(\lvert X(a'_i)\rvert)$	$\max\limits_{1\leqslant i\leqslant n'}(\lvert A(x'_i)\rvert)$	$avg\limits_{1\leqslant i\leqslant n'}(\lvert A(x'_i)\rvert)$	$dev\limits_{1\leqslant i\leqslant n'}(\lvert A(x'_i)\rvert)$
facebook30w	297 474	13 604	253 963	428.8	774	19.6	591
kosarak30w	300 000	31 976	182 107	76.0	2 498	8.1	556
Enron30w	300 000	791 165	239 119	53.4	3 162	140.8	34 588
accidents30w	300 000	458	299 969	22 131.3	51	33.7	9.2

根据第 1 节的描述, 基于 FP-tree*的集合相似度计算的第 1 步是转置数据集. 数据子集x'转置后的数据集x''的大小$|x''|$为x'所有元素属性合集大小$|A(x')|$, 即$|x''|=|A(x')|$. FP-SJ, LP-SJ, TELP-SJ 等算法的计算复杂度即树遍历的复杂度$O(T)=|A(x')|\times\max\limits_{1\leqslant i\leqslant m'}(|X(a'_i)|)\times\max\limits_{1\leqslant i\leqslant n'}(|A(x'_i)|)$. 而对 PPC-SJ 来说, 基于构建好 N-lists 的集合相似度自连接计算是对任意 2 个 N-lists 的比较, 其计算复杂度$O(N_{\text{lists}})=|x'|\times|A(x')|^2$.

图 5 展示 4 种 FP-tree*-SJ 算法的运行时间随着阈值 t 提高的变化, 我们可以看到在 facebook30w, Enron30w, kosarak30w 这 3 组数据集上 FP-SJ, LP-SJ, TELP-SJ 的运行时间比 PPC-SJ 的运行时间少很多. 如表 2 所示, 这 3 组数据的$|A(x')|$很大, 说明原数据子集元素属性合集很大, 即是高维大数据. 该实验数据说明这 3 种算法比 PPC-SJ 更适合处理高维大规模集合相似度自连接计算, 其中 TELP-SJ 的运行时间最少.



(a) facebook30w (b) kosarak30w (c) Enron30w (d) accidents30w

Fig. 5 Relationship between FP-tree*-SJ's running time and the threshold

图 5 FP-tree*-SJ 的运行时间与阈值关系

在 accidents30w 数据集上 PPC-SJ 的运行效率最好, 其次是 TELP-SJ, 最差是 FP-SJ. accidents30w 的$|A(x')|=458$, 而其他 3 组的$\max\limits_{1\leqslant i\leqslant m'}(|X(a'_i)|)$和$|A(x')|$都很大, 所以 PPC-SJ 的运行效率最好. 该实验数据说明 PPC-SJ 比其他 3 种算法适合处理低维大规模集合相似度自连接计算. 此种情况下, TELP-SJ 比 FP-SJ, LP-SJ 的运行效率也要高.

表 3 展示了阈值 $t=0.6$ 时 4 种 FP-tree*-SJ 算法处理 Enron30w 数据集时各分步的运行时间, 包括建树时间、遍历树时间、总耗时. 表 4 展示了 TELP- SJ 与其他 3 种算法的相对性能提升率. 由表 3 和表 4 可见, 4 种算法遍历树的时间在总耗时中占比较高, 而 TELP-SJ 和 LP-SJ 的遍历时间少于 FP-SJ 和 PPC-SJ, 这与第 1 节的分析一致. 其中, TELP-SJ 的遍历时间最少, 相比于 FP-SJ 和 PPC-SJ 性能分别提高了 32% 和 83%. 同时, 我们也看到, TELP-SJ 的建树时间和遍历树时间都比 LP-SJ 少, 其中建树时间性能提高了 45%, 遍历时间性能提高了 17%, 这也与第 1 节的分析一致. 随着阈值 t 的增大, 因为大部分数据在第 1 阶段过滤中被过滤掉, 候选集规模小, 建树和遍历树的代价差异减小, 相应地 TELP-SJ 与 FP-SJ 和 LP-SJ 的运行时间差异减小.

Table 3 Detailed Execution Time of Enron30w Dataset handled by FP-tree*-SJ

表 3 FP-tree*-SJ处理Enron30w数据集的细分运行时间 s

耗时	FP-SJ	PPC-SJ	LP-SJ	TELP-SJ
建树时间	393	517	895	488
遍历树时间	2 087	8 239	1 704	1 419
总耗时	2 496	8 772	2 616	1923

Table 4 Relative Performance Improvement of TELP-SJ Compared with Other FP-tree*-SJ

表 4 TELP-SJ 相对于其他 FP-tree*-SJ 的性能提升 %

相对性能提升	FP-SJ	PPC-SJ	LP-SJ
建树时间	−24	6	45
遍历树时间	32	83	17
总耗时	23	78	26

我们以 Enron 数据集为例, 阈值 t=0.6, 从 5 万条数据起, 每次每组增加 5 万条, 共 6 组数据, 分别运行 4 种算法以观察它们的运行时间与数据子集大小的关系. 如图 6 所示, 随着数据子集大小的增加, TELP-SJ 的运行时间增长速度低于另外 3 种算法, 可扩展性最好.

2) 内存占用峰值

图 7 展示了阈值 $t=0.6$ 时 4 种算法运行期的内存占用率. 由图 7(a)(b)可见, 4 种 FP-tree*-SJ 算法运行期在 facebook30w 和 kosarak30w 数据子集上的内存占用率峰值相近, 图 7(c)(d)则显示在 Enron30w

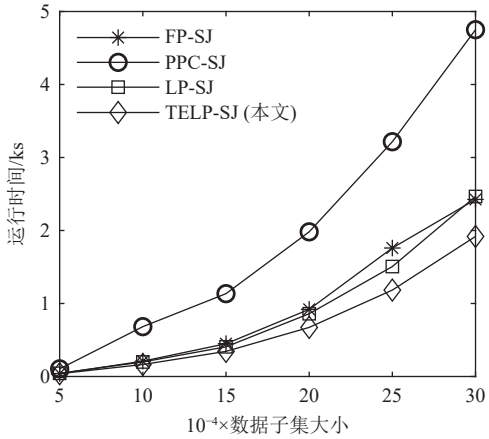


Fig. 6 Relationship between FP-tree*-SJ's running time and the size of Enron's sub-datasets

图 6 FP-tree*-SJ 的运行时间与 Enron 数据子集大小的关系

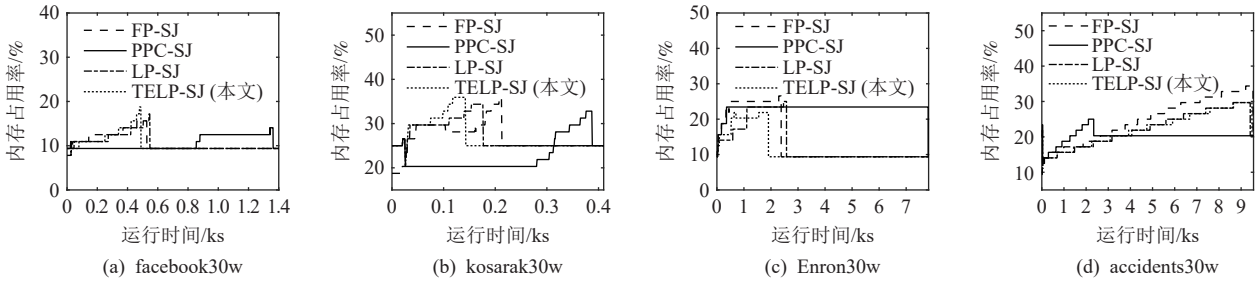
和 acciden30w 数据集上，FP-SJ 的内存占用率峰值最高，PPC-SJ 相对较低和 TELP-SJ 最低，这说明 TELP-SJ 使用数组存储了更多的元素，具有较好的压缩比，而 PPC-tree 因为使用 N-lists 进行计算，不需要建立相同元素结点间的索引链，减少了内存开销.

3.2 FastTELP-SJ 算法与 TELP-SJ 算法的比较

本节我们以 TELP-tree 为例，以 FastTELP-SJ 相对 TELP-SJ 的运行时间加速比及内存占用峰值评估 MapReduce 对基于 FP-tree*的集合相似度自连接算法的加速效果.

实验将 FastTELP-SJ 和 TELP-SJ 分别运行在 4 个数据集的子集上，图 8 描绘了 2 种算法的相对加速比. 其中数据子集大小始于 5 万条，每次每组增加 5 万条，共 6 组数据，阈值 t 分别设置为 0.6 和 0.9. 由图 8 可见，当阈值 $t=0.6$ 时，随着数据子集大小的增加，相对加速比加大. FastTELP-SJ 相对于 TELP-SJ 的相对加速比在 accidents 数据子集上高达 10，且在 facebook 和 Enron 数据子集上相对加速比的增长近似于线性. 当阈值 $t=0.9$ 时，当数据子集大小等于 3 万条时，相对加速比在 Enron 和 accidents 数据子集上分别约为 2 和 7，可见采用并行分布式计算框架 MapReduce 可加速基于 FP-tree*的集合相似度自连接计算.

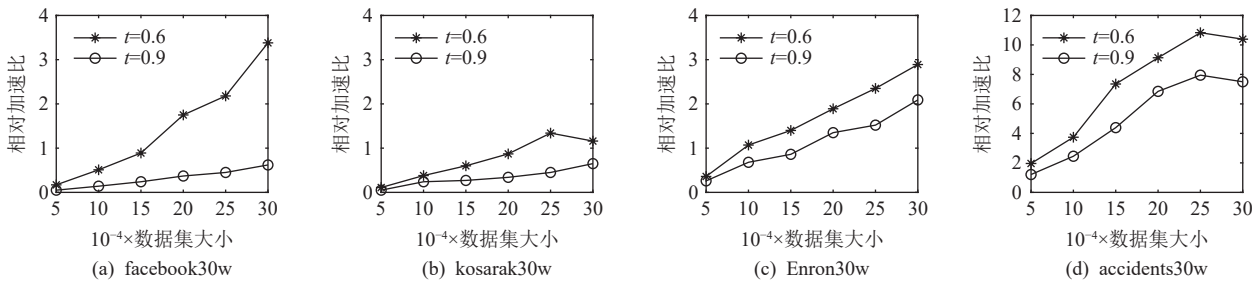
当数据子集比较小时，如图 8 中数据集大小小于 15 万条时，FastTELP-SJ 和 TELP-SJ 算法的相对加速比在 facebook, kosarak, Enron 等数据子集上都小于 1. 同时，在所有 kosarak 数据子集上的相对加速比增长都缓慢，且小于 1.5. 这是因为该数据集的 $|A(x')|$ 和 $\max\limits_{1\leqslant i\leqslant m'}(|X(a'_i)|)$ 偏低，如图 9 所示，转置后数据子集的规模和维度都不大，相当于小数据集. 同理，当阈值 $t=0.9$ 时，大部分数据在第 1 阶段过滤中被过滤掉，规模减小后的候选集变成了小数据集，因此，2 种算法的相对加速比在 facebook 和 kosarak 数据子集中都小于 1. 这是因为基于并行分布式计算框架 MapReduce 的计算有额外的运行期开销，包括数据划分、合并、重新分配、排序等，当数据集比较小时，额外地运行其



(a) facebook30w (b) kosarak30w (c) Enron30w (d) accidents30w

Fig. 7 Memory usage of FP-tree*-SJ ($t=0.6$)

图 7 FP-tree*-SJ 的内存占用率（$t=0.6$）



(a) facebook30w (b) kosarak30w (c) Enron30w (d) accidents30w

Fig. 8 Relative speedup of FastTELP-SJ and TELP-SJ

图 8 FastTELP-SJ 与 TELP-SJ 的相对加速比

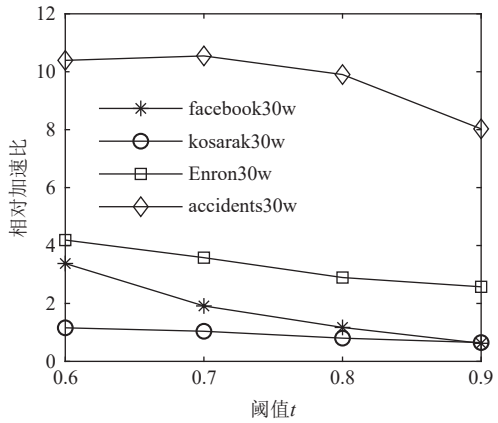


Fig. 9 Relationship between relative speedups of FastTELP-SJ and TELP-SJ, and the threshold

图 9 FastTELP-SJ 和 TELP-SJ 的相对加速比与阈值的关系

开销抵消甚至超过降低切分数据并行计算带来的提升. 此时, TELP-SJ 比 FastTELP-SJ 的运行效率高.

另外, 图 9 描绘了 2 种算法的相对加速比与阈值 t 的关系. 数据子集大小均为 30 万条, 阈值 t 设置为0.6 ~ 0.9. 从图 9 可以看出 FastTELP-SJ 和 TELP-SJ 的相对加速比在 4 个数据集上都大于 1, 且随着阈值的减小, 相对加速比在不断增大. 这是因为在阈值减小时, 第 1 阶段过滤后得到的候选集规模变大, 需要进行两两交集大小计算的集合对个数增多, 利用并行分布式计算框架 MapReduce 的 FastTELP-SJ 能使用更多计算资源完成计算, 因而减小计算耗时.

图 10 描绘的是阈值 $t = 0.6$时 FastTELP-SJ 和 TELP-SJ 在 4 个数据子集上的运行期内存占用率, 从图 10 可见, 除了在 enron30w 数据子集上两者的内存占用率峰值基本相等外, 在其他 3 个子集上 FastTELP-SJ 的内存占用率峰值接近于 TELP-SJ 的1/2, 即采用并行分布式计算框架 MapReduce 可减少单个节点的内存负载.

3.3 FastTELP-SJ 与现有经典算法的比较

从 3.1 节和 3.2 节的实验结果可见, FastTELP-SJ 在处理大规模集合相似度自连接计算时相较于基于其他 FP-tree*-SJ 算法运行效率最高. RP-PPJoin 和 FS-Join 是基于 MapReduce 的集合相似度自连接算法的代表. 另外, 最近提出的位图过滤[12] 在单机和 GPU 系统上运行均展现了它的有效性. 在本节, 为了更全面地评估 FastTELP-SJ 的性能, 本文同时设计并实现基于位图过滤策略的 RP-PPJoin, 记为 RP-PPJoin + Bitmap. 我们将通过 FastTELP-SJ 与 RP-PPJoin, RP-PPJoin + Bitmap, FS-Join 这 4 种算法的比较实验评估 FastTELP-SJ 的性能.

1) 运行时间

首先将这 4 种算法运行在 facebookb30w, kosarak 30w, Enron30w, accidents30w 这 4 组数据集上. 阈值 t 的取值范围为0.6 ~ 0.9. 图 11 展示 4 种算法随着阈值 t 的提高运行时间的变化, 可以看到通过使用位图过滤, RP-PPJoin + Bitmap 在 facebookb30w, kosarak30w,

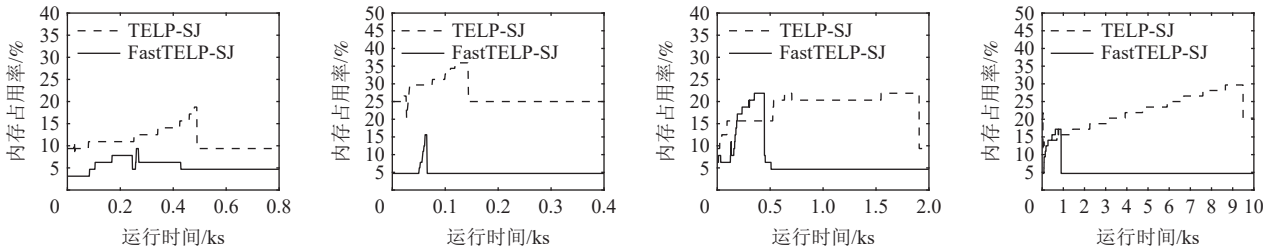


Fig. 10 Memory usage of FastTELP-SJ and TELP-SJ ($t = 0.6$)

图 10 FastTELP-SJ 与 TELP-SJ 的内存占用率 ($t = 0.6$)

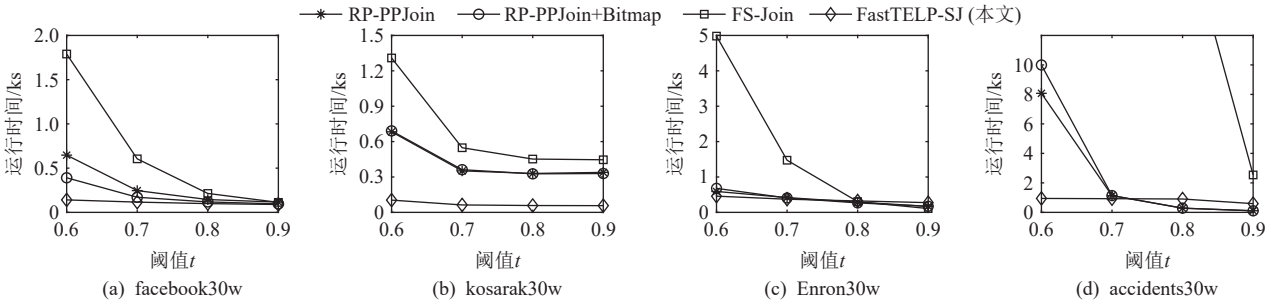


Fig. 11 Relationship between running time of FastTELP-SJ, RP-PPJoin, RP-PPJoin+Bitmap and FS-Join, and the threshold

图 11 FastTELP-SJ 和 RP-PPJoin, RP-PPJoin+Bitmap, FS-Join 的运行时间与阈值的关系

Enron30w 上的运行效率都得到提高. 而 FastTELP-SJ 的运行时间在这 3 个数据集上都优于其他 3 种算法，尤其在阈值 t<0.7 时运行时间明显降低很多. 这主要是 TELP-SJ 将数据压缩在内存中进行计算，减少了 I/O 次数，减小了候选集和降低了计算复杂度. 而此时 RP-PPJoin, RP-PPJoin + Bitmap, FS-join 算法产生较大候选集，其$O\left(kn^2\right)$的高复杂度验证计算延长了运行时间. 尤其对 accidents30w，当$t \leqslant 0.85$时，FS-Join 无法在当前计算环境中完成计算，当$t < 0.7$时，RP-PPJoin 和 RP-PPJoin+Bitmap 的运行时间大幅度升高，而 FastTELP-SJ 的运行时间增长不多，运行效率最高.

图 12 描绘了这 4 种算法的运行时间与数据子集大小的关系，以 accidents 数据集为例，阈值 t 分别设置为 0.6 和 0.9. 其中，0.6 代表了低阈值情况，而 0.9 代表了较高阈值的情况. 数据子集大小始于 5 万条、每次每组增加 5 万条，取 6 组数据，分别运行这 4 种算法并观察它们的运行时间与数据子集大小的关系. 如图 12(a)所示，当$t = 0.6$时，FastTELP- SJ 的运行时间低于另外 3 种算法，且其运行时间随数据增长呈线性增长，可扩展性优于另外 3 种算法. 当$t = 0.9$时，高阈值下各算法应用过滤策略过滤了大量相似度低于阈值的候选集合对. 此时，FastTELP-SJ 的运行期开销如建树时间等导致它的总运行时间高于 RP-PPJoin 和 RP-PPJoin + Bitmap，但仍然低于 FS-Join，如图 12(b)所示，这个结果也与图 10 中t值较大时的结果一致.

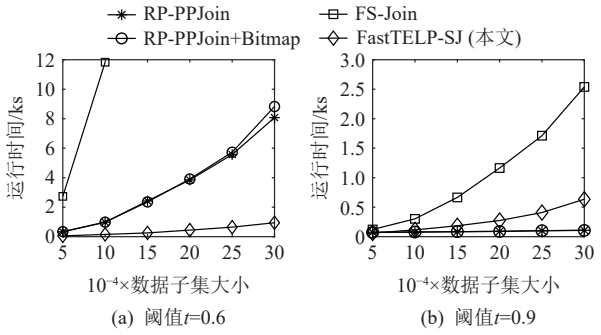


Fig. 12 Relationship between running time of FastTELP-SJ, RP-PPJoin, RP-PPJoin+Bitmap and FS-Join, and the size of accidents' sub-datasets

图 12 FastTELP-SJ, RP-PPJoin, RP-PPJoin + Bitmap, FS-Join 的运行时间与 accidents 数据子集大小的关系

最后，我们评估这 4 种算法的运行时间与集群规模的关系，其中，集群规模为集群节点个数. 实验以 facebook30w 数据集为例，阈值 t 设置为 0.6，集群规模起始计算节点为 2，依次增加 2 个直至 16 个，分别运行 4 种算法并观察它们的运行时间与集群规模的关系. 图 13 描绘了实验结果，随着集群节点个数的增加，相比于RP-PPJoin,RP-PPJoin+Bitmap,FSJoin 和FastTELP-SJ 运行时间的变化速率下降得更慢. 这说明相较于其他 3 种算法，FastTELP-SJ 的扩展性最好.

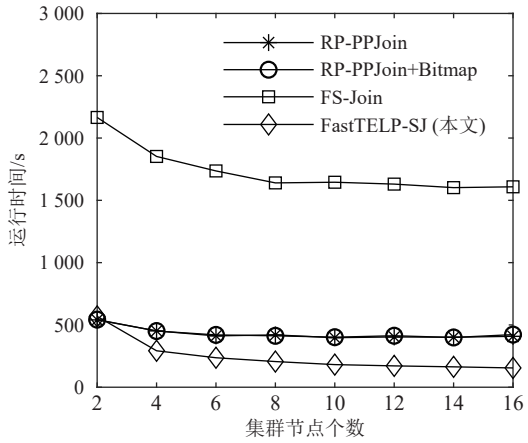


Fig. 13 Relationship between running time of FastTELP-SJ, RP-PPJoin, RP-PPJoin+Bitmap and FS-Join, and the cluster's size

图 13 FastTELP-SJ, RP-PPJoin, RP-PPJoin + Bitmap, FS-Join 的运行时间与集群规模的关系

2) 内存占用率

图 14 描绘了 4 种算法在 facebookb30w, kosarak 30w, Enron30w, accidents30w 这 4 组数据子集进行集合相似度自连接计算过程中的内存占用率，其中，FastTELP-SJ 在处理 kosarak30w 和 Enron30w 时内存占用率峰值分别是其他 3 种算法的 2~3 倍，因为 TELP-tree 在其任务 2 的函数 *reduce* 运行过程中常驻内存，这跟第 1, 2 节的分析一致. 但 RP-PPJoin 和 FS-Join 的候选集大，也有可能出现内存占用率高的情况，如图 14(a)(d). 同时 RP-PPJoin+Bitmap 的内存占用率峰值始终最大，这是因为该算法需要为每个集合构造相应的位图并保存在内存中进行计算.

3) 磁盘使用量

图 15 描绘了 4 种算法在 facebookb30w, kosarak 30w, Enron30w, accidents30w 这 4 组数据子集上进行集合相似度自连接计算过程中的磁盘使用量. 当阈值 $t = 0.6$ 时，FastTELP-SJ 在处理 facebook30w, kosarak-30w, accidents30w 这 3 个数据子集时的磁盘读写量最低，因为它的验证计算都在内存中完成，减少了 I/O，此结论与第 1, 2 节的分析和 3.3.1 节运行时间相关的实验结果分析相一致. 尤其是 RP-PPJoin 和 RP-PPJoin+Bitmap, 因它们可能产生重复验证的冗余候选集，磁盘使用量最大，见图 15(c)(d). 而当阈值t增大时，如

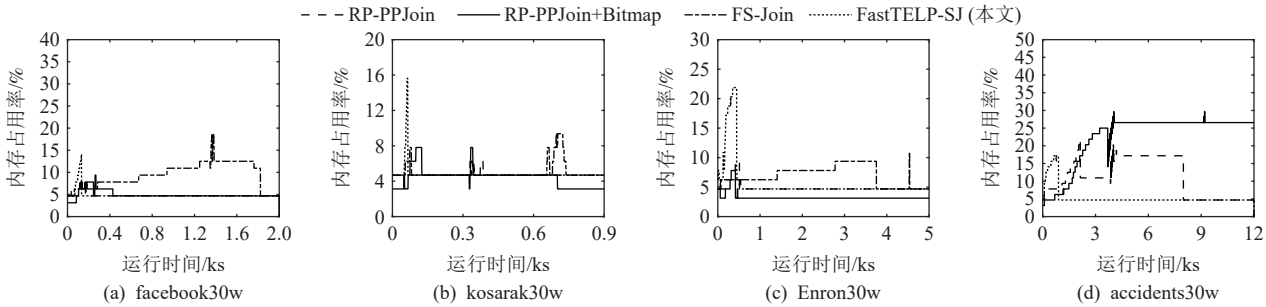


Fig. 14 Memory usage of four algorithms ($t = 0.6$)

图 14 4 种算法的内存占用率($t = 0.6$)

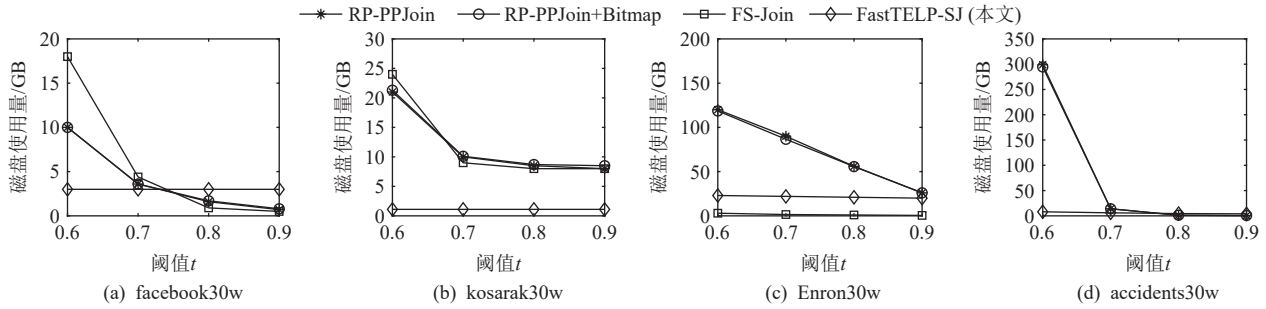


Fig. 15 Disk usage amount of four algorithms

图 15 4 种算法的磁盘使用量

$t = 0.9$时, 由于过滤策略的应用, 大量数据被过滤, 各个算法的磁盘使用量都降低, 差别也减少.

3.4 小 结

基于 3.1~3.3 节的实验结果比较, 当阈值 t 比较高时, 如在我们的实验环境和数据下, $t > 0.8$时, 4 种算法的执行性能差别不大. 当阈值 t 比较低时, 我们有 4 点实验结论:

1) 高维大规模集合相似度自连接计算中 TELP-SJ 的运行效率最好且具有较好的可扩展性.

2) 低维大规模集合相似度自连接计算中 PPC-SJ 的运行效率最好.

3) 高维大规模集合相似度自连接计算中基于并行分布式计算模型 MapReduce 和 TELP-tree 设计的算法 FastTELP-SJ 相对于 TELP-SJ 有较好的加速比.

4) 大规模集合相似度自连接计算中 FastTELP-SJ 的运行效率优于现有基于 MapReduce 的 3 类算法 RP-PPJoin, RP-PPJoin + Bitmap, FS-Join.

4 结束语

数据的增长、集合规模的加大给现有集合相似度自连接算法带来挑战. 基于 MapReduce 的并行分布式集合相似度自连接加速了计算, 但当阈值较低时现有算法生成了较大规模的候选集, 执行效率依然不理想.

为此, 本文提出并设计采用频繁模式树及其派生结构 FP-tree*加速集合相似度自连接算法: 1)设计面向基于 FP-tree*的集合相似度自连接算法及加速其计算的 2 阶段过滤规则, 减小 FP-tree*的规模并提高树遍历效率; 2)提出并构建面向集合相似度自连接的遍历效率更高的 FP-tree 派生结构-线性频繁模式树结构 TELP-tree 及基于其的算法 TELP-SJ; 3)设计基于 MapReduce 和 TELP-tree 的集合相似度自连接算法 FastTELP-SJ. 基于 4 组真实应用数据集, 通过 3 组比较实验从运行时间、内存占用率、磁盘使用量等方面进行对比实验来评估 FastTELP-SJ 的性能, 实验结果展现了 FastTELP-SJ 较好运行性能和可扩展性.

FastTELP-SJ 算法将数据压缩在内存中进行计算, 带来构建树和销毁树等运行期内存管理开销, 同时, FastTELP-SJ 算法包含 2 次 MapReduce 任务, 带来运行期流程管理开销. 经调研, 文献 [39−40] 设计了一种基于生命周期的内存管理框架 Deca, 通过自动分析用户定义的函数和数据结构, 预测其生命周期. 基于该预测分配和释放空间可提高系统垃圾回收效率, 进而提高内存使用率. 另外, 文献 [41−42] 提出了一种面向流数据和迭代任务流程管理降低能耗的资源共享框架 F3C. 我们将进一步研究如何结合有效的内

存管理、流程管理框架构建绿色高效的集合相似度连接计算.

作者贡献声明：冯禹洪发现问题、定义问题、提出算法思路和实验方案；吴坤汉、黄志鸿、冯洋洲负责完成算法设计、实现、性能评估；陈欢欢、白鉴聪、明仲优化算法和实验方案. 所有作者共同完善国内外研究现状调研、完成论文的撰写和修改.

参 考 文 献

[1] Das A S, Datar M, Garg A, et al. Google news personalization: Scalable online collaborative filtering [C] // Proc of the 16th Int Conf on World Wide Web. New York: ACM, 2007: 271–280

[2] Xiao Chuan, Wang Wei, Lin Xuemin, et al. Efficient similarity joins for near-duplicate detection[J]. ACM Transactions on Database Systems, 2011, 36(3): 1−41

[3] Dong Wei, Moses C, Li Kai. Efficient k-nearest neighbor graph construction for generic similarity measures [C] //Proc of the 20th Int Conf on World Wide Web. New York: ACM, 2011: 577–586

[4] Statista Research Department. Facebook MAU worldwide 2021 statista [EB/OL]. [2021-11-16].https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/

[5] Sohrabi M K, Azgomi H. Parallel set similarity join on big data based on locality-sensitive hashing[J]. Science of Computer Programming, 2017, 145(12): 1−12

[6] Rashtchian C, Sharma A, Woodruff D. LSF-Join: Locality sensitive filtering for distributed all-pairs set similarity under skew [C] //Proc of the 29th Web Conf. New York: ACM, 2020: 2998−3004

[7] Christiani T, Pagh R, Sivertsen J. Scalable and robust set similarity join [C] //Proc of the 34th IEEE Int Conf on Data Engineering. Piscataway, NJ: IEEE, 2018: 1240–1243

[8] Bayardo R J, Ma Yiming, Srikant R. Scaling up all pairs similarity search [C] //Proc of the 16th Int Conf on World Wide Web. New York: ACM, 2007: 131–140

[9] Ribeiro L A, Härder T. Generalizing prefix filtering to improve set similarity joins[J]. Information Systems, 2011, 36(1): 62−78

[10] Wang Jiannan, Li Guoliang, Feng Jiahua. Can we beat the prefix filtering? An adaptive framework for similarity join and search [C] //Proc of the 2012 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2012: 85–96

[11] Bouros P, Ge S, Mamoulis N. Spatio-textual similarity joins[J]. Proceedings of the VLDB Endowment, 2012, 6(1): 1−12

[12] Gravano L, Ipeirotis P G, Jagadish H V, et al. Approximate string joins in a database (almost) for free [C] //Proc of the 27th Int Conf on Very Large Data Bases. Berlin: Springer, 2001: 491–500

[13] Sandes E F, Teodoro G L, Melo A C. Bitmap filter: Speeding up exact set similarity joins with bitwise operations[J]. Information Systems, 2020, 88(2): 101449−1014463

[14] Bellas C, Gounaris A. An empirical evaluation of exact set similarity join techniques using GPUs[J]. Information Systems, 2020, 89(3): 101485−101503

[15] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107−113

[16] Vernica R, Carey M J, Li Chen. Efficient parallel set-similarity joins using MapReduce [C] //Proc of the 2010 ACM SIGMOD Int Conf on Management of Data. New York: ACM: 2010: 495–506

[17] Rong Chutian, Lu Wei, Wang Xiaoli, et al. Efficient and scalable processing of string similarity join[J]. IEEE Transactions on Knowledge and Data Engineering, 2012, 25(10): 2217−2230

[18] Baraglia R, Morales G D F, Lucchese C. Document similarity self-join with MapReduce [C] //Proc of the 26th IEEE Int Conf on Data Mining. Piscataway, NJ: IEEE, 2010: 731–736

[19] Rong Chuntian, Lin Chunbin, Silva Y N, et al. Fast and scalable distributed set similarity joins for big data analytics [C] //Proc of the 33rd IEEE Int Conf on Data Engineering. Piscataway, NJ: IEEE, 2017: 1059–1070

[20] Fier F, Augsten N, Bouros P, et al. Set similarity joins on MapReduce: An experimental survey[J]. Proceedings of the VLDB Endowme, 2018, 11(10): 1110−1122

[21] Pacífico L, Ribeiro L A. Streaming set similarity joins [C] //Proc of the 15th Int Conf on Enterprise Information Systems. Berlin: Springer, 2020: 24−42

[22] Yang Chengcheng, Chen Lisi, Wang Hao, et al. Dynamic set similarity join: An update log based approach [J/OL]. IEEE Transactions on Knowledge and Data Engineering, 2021[2023-01-18].https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9609684

[23] Yang Jianye, Zhang Wenjie, Wang Xiang, et al. Distributed streaming set similarity join [C] //Proc of the 36th IEEE Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2020: 565−576

[24] Shahvarani A, Jacobsen H A. Distributed stream KNN join [C] //Proc of the 2021 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2021: 1597−1609

[25] Han Jiawei, Pei Jian, Yin Yiwen. Mining frequent patterns without candidate generation[J]. ACM SIGMOD Record, 2000, 29(2): 1−12

[26] Nancy P, Ramani R G. Frequent pattern mining in social network data (facebook application data)[J]. European Journal of Scientific Research, 2012, 79(4): 531−540

[27] Feng Yuhong, Wang Junpeng, Zhang Zhiqiang, et al. The edge weight computation with MapReduce for extracting weighted graphs[J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(12): 3659−3672

[28] Deng Zhihong, Wang ZhongHui, Jiang Jiajian. A new algorithm for fast mining frequent itemsets using N-lists[J]. Science China Information Sciences, 2012, 55(9): 2008−2030

[29] Pyun G, Yun U, Ryu K H. Efficient frequent pattern mining based on linear prefix tree[J]. Knowledge-Based Systems, 2014, 55(1): 125−139

[30] Feng Yuhong, Guo Meihong, Lu Kezhong, et al. Optimize the FP-Tree based graph edge weight computation on multi-core MapReduce

clusters [C] // Proc of the 23rd IEEE Int Conf on Parallel and Distributed Systems. Piscataway, NJ: IEEE, 2017: 400–409

[31] The Apache Software Foundation. Apache Hadoop [EB/OL]. [2022-11-05].https://hadoop.apache.org

[32] The Apache Software Foundation. Apache Spark [EB/OL]. [2022-11-05].https://spark.apache.org

[33] Ranger C, Raghuraman R, Penmetsa A, et al. Evaluating MapReduce for multi-core and multiprocessor systems [C] //Proc of the 13th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2007: 13–24

[34] He Bingsheng, Fang Wenbin, Luo Qiong, et al. Mars: A MapReduce framework on graphics processors [C] // Proc of the 17th Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2008: 260–269

[35] Nazir A, Raza S, Gupta D, et al. Network level footprints of facebook applications [C] //Proc of the 9th ACM SIGCOMM Conf on Internet Measurement. New York: ACM, 2009: 63–75

[36] Bart G. Frequent itemset mining implementations repository [EB/OL]. [2022-11-05].https://fimi/uantwerpen.be/data

[37] William W. Enron email dataset [EB/OL]. [2022−11-05].https://www.cs.cmu.edu/~enron

[38] Malik A J, Khan F A. A hybrid technique using binary particle swarm optimization and decision tree pruning for network intrusion detection[J]. Cluster Computing, 2018, 21(1): 667−680

[39] Lu Lu, Shi Xuanhua, Zhou Yongluan, et al. Lifetime-based memory management for distributed data processing systems[J]. Proceedings of the VLDB Endowment, 2016, 9(12): 936−947

[40] Shi Xuanhua, Ke Zhixiang, Zhou Yongluan, et al. Deca: A garbage collection optimizer for in-memory data processing[J]. ACM Transactions on Computer Systems, 2019, 36(1): 1−47

[41] Luo Siqi, Chen Xu, Zhou Zhi. F3C: Fog-enabled joint computation, communication and caching resource sharing for energy-efficient IoT data stream processing [C] //Proc of the 39th IEEE Int Conf on Distributed Computing Systems (ICDCS). Piscataway, NJ: IEEE, 2019: 1019−1028

[42] Luo Siqi, Chen Xu, Zhou Zhi, et al. Fog-enabled joint computation, communication and caching resource sharing for energy-efficient IoT data stream processing[J]. IEEE Transactions on Vehicular Technology, 2021, 70(4): 3715−3730

Feng Yuhong, born in 1975. PhD, associate professor. Member of CCF. Her main research interests include parallel and distributed computing, middleware, system software, and data analysis.

冯禹洪, 1975 年生. 博士, 副教授. CCF 会员. 主要研究方向为并行分布式计算、中间件、系统软件、数据分析.

Wu Kunhan, born in 1997. Master. His main research interests include parallel and distributed computing, and data analysis.

吴坤汉, 1997 年生. 硕士. 主要研究方向为并行分布式计算、数据分析.

Huang Zhihong, born in 1994. Master. His main research interests include parallel and distributed computing, and data analysis.

黄志鸿, 1994 年生. 硕士. 主要研究方向为并行分布式计算、数据分析.

Feng Yangzhou, born in 1998. Bachelor. His main research interests include parallel and distributed computing, and data analysis.

冯洋洲, 1998 年生. 学士. 主要研究方向为并行分布式计算、数据分析.

Chen Huanhuan, born in 1982. PhD, professor, PhD supervisor. Member of CCF. His main research interests include machine learning, data mining, computational intelligence, underground pipeline network detection and data fusion, and evolutionary computing.

陈欢欢, 1982 年生. 博士, 教授, 博士生导师. CCF 会员. 主要研究方向为机器学习、数据挖掘、计算智能、地下管网探测与数据融合、演化计算.

Bai Jiancong, born in 1977. PhD, lecturer. Member of CCF. His main research interests include intelligent optimization algorithm, big data, and e-commerce.

白鉴聪, 1977 年生. 博士, 讲师. CCF 会员. 主要研究方向为智能优化算法、大数据、电子商务.

Ming Zhong, born in 1967. PhD, professor, PhD supervisor. Senior member of CCF. His main research interests include software engineering, ontology, Internet of things, and workflow.

明　仲, 1967 年生. 博士, 教授, 博士生导师. CCF 高级会员. 主要研究方向为软件工程、本体、物联网、工作流.