软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.007209]

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

面向时间有序事务数据的聚簇频繁模式挖掘*

王少鹏[1,2,3], 牛超煜[1]

[1](内蒙古大学 内蒙古大学计算机学院, 内蒙古 呼和浩特 010021)

[2](内蒙古教育部生态大数据工程研究中心, 内蒙古 呼和浩特 010021)

[3](内蒙古云计算与服务工程实验室, 内蒙古 呼和浩特 010021)

通信作者: 王少鹏, E-mail: wangsp@imu.edu.cn

摘　要: 本文首次对时间有序事务数据中聚簇频繁模式的挖掘问题进行研究. 为了解决 Naive 算法处理该问题时存在冗余运算的问题, 提出了一种改进的聚簇频繁模式挖掘算法 ICFPM (Improved Cluster Frequent Pattern Mining, ICFPM). 该算法使用了 2 种优化策略, 一方面可以利用定义的参数 *minCF*, 有效减少挖掘结果的搜索空间, 另一方面可以参考 (n – 1) 项集的判别结果加速聚簇频繁 n 项集的判别过程, 算法还使用了 *ICFPM-list* 结构来减少候选 n 项集的构建开销. 基于 2 个真实世界数据集的仿真实验证明了 ICFPM 算法的有效性, 与 Naive 算法相比, ICFPM 算法在时间和空间效率方面得到了大幅度的提高, 是解决聚簇频繁模式挖掘的有效方法.

关键词: 时间有序事务数据; 聚类; 频繁模式; 数据挖掘; 向下闭包

中图法分类号: TP311

中文引用格式: 王少鹏, 牛超煜. 面向时间有序事务数据的聚簇频繁模式挖掘. 软件学报. http://www.jos.org.cn/1000-9825/7209.htm

英文引用格式: Wang SP, Niu CY. Clustering Frequent Pattern Mining for Time-Ordered Transaction Data. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7209.htm

Clustering Frequent Pattern Mining for Time-Ordered Transaction Data

WANG Shao-Peng[1,2,3], NIU Chao-Yu[1]

[1](College of computer science, Inner Mongolia university, Hohhot 010021, China)

[2](Inner Mongolia Engineering Research Center of Ecological Big Data Ministry of Education, Hohhot 010021, China)

[3](Inner Mongolia Engineering Laboratory for Cloud Computing and Service, Hohhot 010021, China)

Abstract: In this study, the problem of mining cluster frequent patterns in time-ordered transaction data is discussed for the first time. To deal with redundant operations when the Naive algorithm solves this problem, the improved cluster frequent pattern mining (ICFPM) algorithm is proposed. The algorithm uses two optimization strategies. On the one hand, it can use the defined parameter *minCF* to effectively reduce the search space of mining results; on the other hand, it can refer to the discriminative results of (n – 1)-itemsets to accelerate the discriminative process of cluster frequent n-itemset. The algorithm also applies the ICFPM-list structure to reduce the overhead of the candidate n-itemsets construction. Simulation experiments based on two real-world datasets demonstrate the effectiveness of the ICFPM algorithm. Compared with the Naive algorithm, the ICFPM algorithm improves substantially in terms of time and space efficiency, which makes it an effective method for solving clustered frequent pattern mining.

Key words: time-ordered transaction data; clustering; frequent patterns; data mining; downward closure

频繁项集挖掘[1-6]是数据挖掘领域的一个研究热点[7], 旨在发现事务数据集中经常一起出现的项集, 最初是为购物篮分析而设计, 后被广泛应用于许多其他数据挖掘任务. 包括关联规则挖掘[8-10]、顺序模式挖掘[11-13]、聚类[14-15]和分类[16-17]等. 当前有关频繁项集挖掘的研究按照事务数据集中数据是否时间有序, 可分为面向无时间标签事务

* 基金项目: 国家自然科学基金 (62066034, 62262047); 内蒙古科技计划基金 (61862047)
收稿时间: 2023-12-11; 修改时间: 2024-03-13; 采用时间: 2024-04-13; jos 在线出版时间: 2024-06-20

数据和面向时间有序事务数据 2 大类. 其中面向无时间标签事务数据的频繁项集挖掘包括基本频繁项集和衍生出的最大频繁项集[18-20]、频繁闭项集[21-23]、高效用项集[24-26]等相关的研究. 在面向时间有序事务数据集的频繁项集挖掘领域, 相关研究主要关注频繁项集在数据集中发生位置的分布情况. 比如周期模式[27-30]挖掘定期发生在事务数据集中的项集; 局部周期模式[7, 31]挖掘在某几个时间区间内定期发生的项集; 部分周期模式[32, 33]挖掘定期发生在事务数据集中但不一定连续发生的项集; 稀有周期模式[33]挖掘不频繁发生但在事务数据集中定期发生的项集; 规范频繁模式[34-39]挖掘以固定间隔出现在事务数据集中的项集; 周期聚簇模式[40]在事件序列上找到发生位置聚簇状并且任意 2 个相邻簇的间隔满足周期特点的事件. 但总体来看, 当前这一领域还没有针对发生位置呈聚簇状态的频繁项集挖掘问题展开研究, 这使得很多与之相关的应用无法直接使用现有方法进行有效处理. 我们可以通过如下实例来说明这个问题.

表 1 是一个时间有序的事务数据集, 描述了零售商店中 6 月 6 日-6 月 20 日顾客的购物篮情况. 其中 a、b、c、d、e 和 f 分别代表不同的商品, Timestamp 表示事务数据的发生时间. 如果我们以 5 为频繁阈值基于该数据集挖掘频繁项集, 以 2 个频繁 1 项集 a, f 为例, 它们都会被挖掘出来, 但有时候我们可能只想得到类似 f 这种发生位置呈集中或聚簇状态的频繁项集 (比如端午节前后的粽叶和糯米, 中秋节前后的月饼, 促销期的商品等), 以及这些项集聚簇的位置 (比如 f 的聚簇位置是 6 月 12 日~6 月 14 日和 6 月 18 日~6 月 20 日), 这样可以更好地了解客户需求, 改进营销计划. 但据我们所知, 目前频繁项集相关的研究都无法有效解决这个问题. 直观上该问题可以使用如下步骤来解决, 首先利用现有的频繁模式挖掘方法求出所有频繁模式, 接着找到由每个频繁模式对应的所有发生时间组成的集合, 最后对集合进行聚类并判断, 就可以找到所有聚簇频繁模式 (Naive 算法), 但这种方法的时间和空间复杂度很高. 因此, 在时间有序事务数据集上挖掘聚簇频繁模式的问题仍然没有得到有效解决.

表 1　时间有序事务数据集

Tid	Transaction	Timestamp
1	a, c	2023年6月6日
2	d	2023年6月7日
3	a, b	2023年6月7日
4	b, d	2023年6月8日
5	a, b, f	2023年6月12日
6	b, c, e, f	2023年6月13日
7	b, d, e, f	2023年6月13日
8	a, b, d, e, f	2023年6月14日
9	b, c, d, e , f	2023年6月14日
10	a, d, e	2023年6月15日
11	c, d	2023年6月17日
12	b, f	2023年6月18日
13	a, c, e, f	2023年6月19日
14	a, b, d, e, f	2023年6月20日
15	d, f	2023年6月20日

在本文中, 我们针对时间有序事务数据集中聚簇频繁模式的挖掘问题展开了研究. 主要贡献如下: (1) 给出了聚簇频繁模式的形式化定义. (2) 提出了聚簇频繁模式挖掘算法 ICFPM(Improved Cluster Frequent Pattern Mining, ICFPM), 该算法基于 Naive 方法设计了 2 种有效的优化策略. 并通过使用数据结构 *ICFPM-list* 来减少时空开销. (3) 基于 2 个真实数据集的实验结果表明, ICFPM 在时间和空间成本上都要优于 Naive 算法.

本文的其余部分组织如下. 第 1 节介绍了聚簇频繁模式的相关工作; 第 2 节给出了聚簇频繁模式形式化定义和本文关注问题的具体描述; 第 3 节介绍了聚簇频繁模式的挖掘方法; 第 4 节展示实验结果并对结果进行分析; 第 5 节进行总结.

1 相关工作

本文关注的问题属于时间有序事务数据中模式挖掘这一类别, 所以这部分只对该领域的研究情况进行调研. 总体来看, 当前这个领域的研究分为周期模式相关和规范模式相关 2 类.

1.1 周期模式相关

Tanbeer 等人[27]提出了在事务数据集中挖掘周期性频繁模式的问题, 并设计了一种基于 PF-tree 结构的模式增长挖掘算法. Kiran 等人[28]基于模式增长的算法 PP-growth 挖掘周期模式, 同时考虑了周期性模式的不同周期和最小循环重复次数. Venkatesh 等人[29]提出了周期相关模式, 引入了"周期-全置信度"的测量方法确定模式的周期相关性, 并设计了 EPCP-growth 的模式增长算法. Kiran 等人[30]设计了 PS-tree 结构记录项集时间发生信息, 并基于此结构设计了 PS-growth 算法挖掘周期性频繁模式. Fournier-Viger 等人[7]提出了局部周期模式, 可以发现在一些非预定义的时间区间内具有周期性行为的项集, 在该论文中作者使用 2 个新度量方法来评估时间间隔内模式的周期性和频率, 分别是最大溢出周期 (度量允许检测具有可变长度的时间区间), 最小持续时间 (度量确保这些发生时间区间具有最小持续时间). Krzywicki 等人[31]设计了一种新的算法 AllPat, 可以在不需要指定目标周期的情况下, 识别局部周期性模式. Kiran 等人[32]设计了 3P-growth 算法以及 2 个数据结构 3P-tree 和 3P-list 挖掘部分周期模式. Upadhya 等人[33]设计了 3P-BitVectorMiner 算法挖掘部分周期模式, 并设计了 2 个变种算法 RFPP-BitVectorMiner 和 R3P-BitVectorMiner 挖掘稀有完全周期模式和稀有部分周期模式. Chen 等人[40]提出了周期聚簇模式, 设计了周期聚簇模式算法, 可以在事件序列上挖掘到发生位置聚簇状并且任意 2 个相邻簇的间隔满足周期特点的事件.

1.2 规范模式相关

Tanbeer 等人[34]提出了在事务数据集上挖掘规范模式, 设计了称为 RP-tree 的树结构, 并基于 RP-tree 设计了模式增长算法, 挖掘具有用户定义的最大规范性的规范模式集合. Rashid 等人[35]设计了称为 RF-tree 的树结构, 引入了一种基于同一模式在数据库中发生的间隔方差的新的规范性度量, 并基于 RF-tree 设计了模式增长算法挖掘规范频繁模式. Amphawan 等人[36]设计了 TKRIMPE 算法挖掘 top-k 规范频繁模式, 利用数据库分区和支持度估计技术来挖掘 top-k 规范频繁模式. 该 TKRIMPE 采用最佳优先搜索策略, 且仅需要进行 1 次数据库扫描. Kumar 等人[37]设计了 MaRFI 算法, 使用一对事务标识而非项目标识来挖掘最大规范频繁模式, MaRFI 满足向下闭包性质. Amphawan 等人[38]设计了 TFRC-Mine 算法挖掘 top-k 闭规范频繁模式, 利用紧凑的位向量表示来剪枝不感兴趣的候选集. Rehman 等人[39]设计了 TKIFRPM 算法挖掘 top-k 规范频繁模式, 通过维护 ISI-tables 来存储增量数据的统计信息, 采用了深度优先的搜索模式和渐进式剪枝策略, 减少了不必要的候选集生成和支持度计算.

总体来看, 目前与本文类似的研究只有 1.1 中 Fournier-Viger 等人[7] 提出的局部周期模式和 Chen 等人[40] 提出的周期聚簇模式. 对于局部周期模式来说, 它所描述的是某段时间内周期出现, 且有最短持续时间条件约束的模式, 但聚簇频繁模式要找的是在某段时间内或某个时间点大量出现且发生时间呈簇状的模式, 模式并不需要限制持续发生时间; 对于周期聚簇模式来说, 它只描述 1 项集, 并不考虑多项集的发生情况. 另外对于聚簇发生位置不具有周期特点的项, 便会舍弃. 比较而言, 聚簇频繁模式关注所有满足聚簇频繁条件的项集, 而且不具有周期特点的项集也有可能满足聚簇频繁的条件. 所以综上所述, 现有相关研究都不适用于挖掘聚簇频繁模式, 本文试图为该问题的解决提供一种有效的处理方法.

2 基本概念和问题说明

2.1 基本概念

设 $S_I=\{I_1, I_2, ..., I_m\}$是 m 个项的集合, I_i 是其中第 $i(1\leqslant i\leqslant m)$ 个成员, S_I 中的所有成员按全序>(字典序) 排列. 任给项集 $P\subset S_I$, 令$|P|$表示 P 中项的个数, 如果$|P|=l$, 则称 P 为 l 项集 (或模式), P 的长度为 l.

定义 1.**时间有序事务数据集**. 包含 w 个成员的时间有序事务数据集 $TDS=\{T_1, T_2, ..., T_w\}$, 令$|TDS|$为 TDS 的长

度, 有$|TDS|=w$. TDS 中的每个事务数据 $T_i(1\leqslant i\leqslant w)$ 是一个三元组 (i, t_i, ts_i), 其中 $t_i\subset S_I$;ts_i 表示 t_i 的发生时间, i 是 t_i 的唯一标识 ID. 对于 TDS 中任意 2 个相邻成员 T_i 和 $T_j(1\leqslant i<j\leqslant w)$, 其对应的三元组成员 (i, t_i, ts_i) 与 (j, t_j, ts_j) 间存在如下关系: $j=(i+1)$, $ts_j\geqslant ts_i$. 另设 $TDS.ts_{min}$ 与 $TDS.ts_{max}$ 分别表示 TDS 中第一个和最后一个成员的发生时间, 即有 $ts_{min}=ts_1$ 且 $ts_{max}=ts_w$.

例 1. 考虑表 1 所示的时间有序事务数据集 TDS, 其中共有 15 个成员$\{T_1, T_2, ..., T_{15}\}$, 因此$|TDS| = 15$. 该数据集相关的项集 $S_I=\{a, b, c, d, e, f\}$. 集合$\{a, b\}$是一个包含 2 个项且长度为 2 的 2 项集.T_1 表示在 2023 年 6 月 6 日购买了商品 a 和 c. 另外也可以看出, 该实例中 $TDS.ts_{min}=ts_1$=2023 年 6 月 6 日且 $TDS.ts_{max}=t_{15}$=2023 年 6 月 20 日.

定义 2.**事务数据中项集的出现**[7]. 考虑 TDS 的事务数据 $T_i(1\leqslant i\leqslant w)$ 和项集 $x\subset S_I$, 如果 $x\subset T_i$, 则称 x 出现在 T_i 处, 或出现在时间戳 ts_i 处.

例 2. 考虑表 1 所示的 TDS, 2 项集$\{c, e\}$出现在 T_6、T_9 和 T_{13} 处, 或称出现在时间戳 ts_6、ts_9 和 ts_{13} 处.

定义 3.**项集发生的事务集合**. 考虑包含 w 个事务的 TDS 和项集 $x\subset S_I$, 集合$\{ T_q |1\leqslant q\leqslant w, T_q$ 属于 TDS 且 $x\subset T_q\}$为项集 x 发生的事务集合, 记为 T^x, $|T^x|$是 T^x 中的成员数.

例 3. 考虑表 1 所示的 TDS, 2 项集$\{c, e\}$出现的事务集合 $T^{\{c, e\}}=\{T_6, T_9, T_{13}\}$, $|T^{\{c, e\}}|=3$.

定义 4.**项集发生的时间集合**. 考虑包含 w 个事务的 TDS 和项集 $x\subset S_I$, 集合$\{T_q.ts_q|1\leqslant q\leqslant w, T_q$ 属于 TDS 且 $x\subset T_q\}$为项集 x 发生的时间集合, 记为 TS^x, $|TS^x|$是 TS^x 中的成员数. 考虑到一个时间可能有多个项集发生, 这里的集合设定为多重集合 (即允许重复的时间戳成员存在), 所以满足$|TS^x|=|T^x|$.

例 4. 接着考虑例 3, $\{c, e\}$的发生时间集合 $TS^{\{c, e\}}=[ts_6, ts_9, ts_{13}]$=[13, 14, 19], 由于时间戳长度较长, 本文都使用数字简写为发生在哪一天, $|TS^x|=|T^x|$=3.

定义 5.**项集关于发生时间的 *Eps*-邻域**[41]. 考虑包含 w 个事务的 TDS 和项集 $x\subset S_I$, x 关于发生时间 $p(\ p\in TS^x)$ 的 Eps-邻域定义为多重集合$\{q \mid q\in TS^x$, dist$(p, q)\leqslant Eps\}$, 记为 $N^x_{Eps}(p)$. 其中, Eps 是设置的邻域半径阈值, dist$(p, q)=|p-q|$.$|N^x_{Eps}(p)|$表示 $N^x_{Eps}(p)$ 中成员数.

例 5. 考虑表 1 所示数据集 TDS, 给定 Eps=2(天), 2 项集$\{b, f\}$对应 $TS^{\{b, f\}}$=[12, 13, 13, 14, 14, 15, 18, 20]. 以时间戳 14 为例, $N^{\{b, f\}}_{Eps}(14)$=[12, 13, 13, 14, 15], $|N^{\{b, f\}}_{Eps}(14)|$=5.

定义 6.**项集关于发生时间的密度**[41]. 考虑包含 w 个事务的 TDS 和项集 $x\subset S_I$, x 关于发生时间 $p(\ p\in TS^x)$ 的密度记为 $\rho_x(p)$, 可表示为 $\rho_x(p)=|N^x_{Eps}(p)|$.

例 6. 接着考虑例 5, $\rho_{\{b, f\}}(14)=|N^{\{b, f\}}_{Eps}(14)|$=5.

定义 7.**项集 *x* 发生时间的核心点**[41]. 考虑包含 w 个事务的 TDS 和项集 x, 给定 $p\in TS^x$, 如果 $\rho_x(p)\geqslant minPts$ 成立, 则 p 是项集 x 发生时间的核心点, 否则 p 是非核心点. 其中, $minPts$ 是设置的密度阈值.

例 7. 接着考虑例 7, 给定 $minPts$=3(天). 由于 $\rho_{\{b, f\}}(14)\geqslant minPts$, 则 14 是$\{b, f\}$发生时间的核心点.

定义 8.**边界点**[41]. 考虑包含 w 个事务的 TDS 和项集 x, $\forall p\in TS^x$, 如果点 p 是非核心点, 并且在 TS^x 中存在成员 q 是项集 x 发生时间的核心点, 使得它们间存在 $p\in N^x_{Eps}(q)$ 的关系, 则 p 是边界点.

例 8. 考虑表 1 所示数据集 TDS, 给定 Eps=1(天), $minPts$=3(天).1 项集$\{f\}$对应 $TS^{\{f\}}$=[12, 13, 13, 14, 14, 18, 19, 20, 20], 以时间戳 18 为例, $N^{\{f\}}_{Eps}(18)$=[19], $\rho_{\{f\}}(18)=1<minPts$, 则 18 是非核心点. 而 $N^{\{f\}}_{Eps}(19)$=[18, 20, 20], $\rho_{\{f\}}(19)=3\geqslant minPts$, 19 是核心点, 且满足 $18\in N^{\{f\}}_{Eps}(19)$, 则 18 是边界点.

定义 9.**项集 *x* 发生时间的噪声点**[41]. 考虑包含 w 个事务的 TDS 和项集 x, $\forall p\in TS^x$, 如果 p 不是核心点, 同时也不是边界点, 则它是噪声点.

例 9. 接着考虑例 5, $N^{\{b, f\}}_{Eps}(18)$=[20], $N^{\{b, f\}}_{Eps}(20)$=[18], 由于 $\rho_{\{b, f\}}(18)$, $\rho_{\{b, f\}}(20)<minPts$, 则 18 和 20 是噪声点.

定义 10.**直接密度可达**[41]. 考虑包含 w 个事务的 TDS 和项集 x, $\forall p, q\in TS^x$, 如果 $p\in N^x_{Eps}(q)$, 且 q 是核心点, 则 q 到 p 直接密度可达.

例 10. 接着考虑例 5, 由于时间戳 $12\in N^{\{b, f\}}_{Eps}(14)$, 则 12 到 14 直接密度可达.

定义 11.**密度可达**[41]. 考虑包含 w 个事务的 TDS 和项集 x, $\forall p, q\in TS^x$, 如果在 TS^x 中存在一系列数据 $p_1, p_2, ...,$

p_i, ..., p_n(p_1=q 且 p_n=p), 且这系列数据中任意 2 个相邻成员 p_i 到 p_{i+1} 直接密度可达, 那么 q 到 p 密度可达, 其中, $p_i \in TS^x$, $1 \leqslant i \leqslant n$.

例 11.接着考虑例 5, 考虑 $TS^{\{b,f\}}$中一系列数据 [12, 13, 13, 14, 14, 15], 通过计算可得任 2 个相邻成员都直接密度可达, 则 12 到 15 密度可达.

定义 12.密度相连[41]. 考虑包含 w 个事务的 *TDS* 和项集 x, $\forall p, q, s \in TS^x$, 如果存在 s 到 p 和 q 皆密度可达, 则 q 到 p 密度相连.

例 12. 接着考虑例 5, 考虑 $TS^{\{b,f\}}$中成员 12, 13 和 15, 可得 13 到 12 和 15 皆密度可达, 则 12 到 15 密度相连.

定义 13.项集 *x* 发生时间的聚簇集合[41]. 考虑包含 w 个事务的 *TDS* 和项集 x, x 发生时间的聚簇集合 $C^x=\{c_1, ..., c_g\}$, $g \geqslant 1$, 其中每个 c_i($1 \leqslant i \leqslant g$) 是非空, 且满足如下条件的 TS^x 子集:

(1) $\forall p, q \in TS^x$, 如果 $p \in c_i$, p 到 q 密度可达, 则 $q \in c_i$.(最大性)

(2) $\forall p, q \in c_i$, p 与 q 密度相连.(连通性)

$|C^x|$表示聚簇内容数, 其值为每个簇中所含 TS^x 中成员数的总和, $|c_i|$是 c_i 中成员数.

例 13. 接着考虑例 12, 可得 $C^{\{b,f\}}=\{c_1\}=\{[12, 13, 13, 14, 14, 15]\}$, $|C^{\{b,f\}}|=6$, $|c_1|=6$.

定义 14.对项集发生时间集合的划分[40]. 考虑包含 w 个事务的 *TDS* 和项集 x, TS^x 是 x 的发生时间集合, 给定参数 *Eps*(半径), *minPts*(最小点数量), TS^x 关于 *Eps* 和 *minPts* 划分得到的 $P^x=C^x \cup Noi^x$ 会将 TS^x 的所有元素分为 (g+1) 组. 其中, $C^x=\{c_1, ..., c_g\}$包含 g 个聚簇;Noi^x 包含所有噪声点, 簇和噪声点分别满足定义 13 和定义 9. 定义噪点比 $NR^x=|Noi^x|/|TS^x|$.

例 14. 接着考虑例 13, 对 $TS^{\{b,f\}}$进行划分, 得到 $P^{\{b,f\}}=C^{\{b,f\}} \cup Noi^{\{b,f\}}$. 其中, $C^{\{b,f\}}=\{c_1\}=\{[12, 13, 13, 14, 14, 15]\}$, $|C^{\{b,f\}}|=6$, $Noi^x=[18, 20]$, $NR^x=2/8=0.25$.

定义 15.频繁项集. 考虑包含 w 个事务的 *TDS* 和项集 x, x 在 *TDS* 中支持度为 *TDS* 中包含 x 的事务个数与 *TDS* 中事务总数$|TDS|$的比值, 记为 $Sup^x=|T^x|/|TDS|$. 根据定义 4, $|TS^x|=|T^x|$, 所以, 也可表示为 $Sup^x=|TS^x|/|TDS|$. 给定频繁阈值 *minSup*, 若 $Sup^x \geqslant minSup$($0 \leqslant minSup \leqslant 1$), 则 x 为频繁项集.

例 15. 接着考虑例 6, 给定 *minSup*=0.2, $\{b, f\}$对应 $Sup^{\{b,f\}}$= 7/15=0.53>*minSup*, 则$\{b, f\}$是一个频繁项集.

定义 16.聚簇频繁模式. 考虑包含 w 个事务的 *TDS* 和项集 x, 给定参数 *Eps*、*minPts* 以及阈值 *minSup* 和 *maxNR*, 其中 *maxNR* 是最大噪点比阈值. 如果项集 x 及对应分区 $P^x=C^x \cup Noi^x$ 满足以下条件:

(1) $Sup^x \geqslant minSup$

(2) $NR^x \leqslant maxNR$

那么 x 就是一个聚簇频繁模式 (Cluster Frequent Pattern, CFP), C^x 的发生区间集合 $L^x=\{l_1, ..., l_g\}$, 其中, g 为 C^x 包含簇的总数, $l_z=[begin(c_z), end(c_z)]$, $begin(c_z)$ 表示簇 c_z 的开始发生时间, $end(c_z)$ 表示簇 c_z 的最后发生时间, $1 \leqslant z \leqslant g$. 注意, 此处的 [] 表示发生区间.

例 16. 接着考虑例 14 和例 15, 给定 *maxNR*=0.3. 由于 $Sup^{\{b,f\}}>minSup$ 且 $NR^{\{b,f\}}<maxNR$, 则$\{b, f\}$是聚簇频繁模式, 对应聚簇频繁模式发生区间集合 $L^{\{b,f\}}=\{l_1\}=\{[12, 15]\}$.

2.2 问题说明

定义 16 给出了本文聚焦问题的形式化定义, 通过最小频繁阈值 (*minSup*) 来保证所找的项集的频繁特征, 通过邻域半径阈值 (*Eps*), 密度阈值 (*minPts*) 以及最大噪点比阈值 (*maxNR*) 来保证所找项集的聚簇特性. 本文接下来的工作就是设计出一种有效的方法, 以便在给定一个 *TDS*, 以及最小频繁阈值, 邻域半径阈值, 密度阈值, 最大噪点比阈值的情况下, 有效挖掘出 *TDS* 中所有频繁且发生时间呈聚簇状态的项集, 并将满足条件的项集及对应聚簇发生区间呈现给用户.

3 聚簇频繁模式挖掘算法

本部分针对聚簇频繁模式的具体挖掘方法展开研究.3.1 节给出了从定义出发挖掘聚簇频繁模式的 Naive 算

法.3.2 节针对 Naive 算法存在的冗余运算, 设计了 2 种优化策略和一种新的数据结构进行优化.3.3 节介绍了优化 Naive 算法后得到的新聚簇频繁模式挖掘算法.

3.1 Naive 算法

3.1.1 算法描述

由定义 16 可知, 聚簇频繁模式的挖掘可基于如下 2 个阶段来完成. 首先, 得到时间有序事务数据集中所有的频繁项集; 接着基于邻域半径阈值 (*Eps*), 密度阈值 (*minPts*) 以及最大噪点比阈值 (*maxNR*) 过滤这些频繁项集, 保证最终得到项集的聚簇特性. 在这 2 个阶段本文分别使用了 Apriori-TID[1] 算法和 DBSCAN 算法[41], 其中 Apriori-TID 是经典的频繁模式挖掘方法, DBSCAN 算法是一种经典的密度聚类算法, 能够保证最终结果的正确性和完整性. 这样的方法被称为 Naive 算法. 具体描述如下:

算法 1. Naive 算法.

输入:*TDS*: 包含 *w* 个事务的时间有序事务数据集, *minSup*, *Eps*, *minPts*, *maxNR*;
输出: 所有聚簇频繁模式.

1. 基于 *TDS* 得到哈希表 *R*, *R* 的结构为 (*x*, *tid*(*x*)). 其中, 键 *x* 是 1 项集, *tid*(*x*) 表示 *x* 在 *TDS* 中的发生事务 *ID* 集合, *R*(*x*) 表示 *tid*(*x*);
2. $Candidate_k=\varnothing$, $k=1$; //$Candidate_k$ 以 *R* 的结构形式存储所有频繁 *k* 项集;
3. 将 *TDS* 中包含的所有 1 项集放入 *itemsets* 中;
4. **foreach** $x\in itemsets$ **do**
5. $TS^x=map(TDS, R(x))$;
6. $Sup^x=|TS^x|/|TDS|$;
7. **if** $Sup^x\geqslant minSup$ **then**
8. 将 (*x*, *R*(*x*)) 加入到 $Candidate_k$;
9. $P^x=C^x\cup Noi^x$=DBSCAN(TS^x, *Eps*, *minPts*); //对 TS^x 进行划分;
10. $NR^x=|Noi^x|/|TS^x|$;
11. **if** $NR^x\leqslant maxNR$ **then** 通过 C^x 计算得到 L^x 并且输出 *x* 及对应 L^x;
12. **end if**
13. **end foreach**
14. $Candidates=Candidate_{k;}$
15. **while** |*Candidates*|>1 **do** //|*Candidates*|表示包含频繁项集总数;
16. $Candidate_{k+1}$=Naive-Generate(*Candidates*, *minSup*, *Eps*, *minPts*, *maxNR*, *TDS*);
17. $Candidates=Candidate_{k+1}$;
18. **end**

如算法 1 所示, Naive 算法首先基于 *TDS* 得到每个 1 项集的发生事务 *ID* 集合, 接着以哈希表 *R* 的形式进行存放. 这里 *R* 的结构形式为 (*x*, *tid*(*x*)), 其中, 项集 *x* 是键, *tid*(*x*) 为 *x* 发生事务 *ID* 集合, 可表示为 *R*(*x*). 初始化 $Candidate_k$ 为空, 其中, $Candidate_k$ 用来存储所有频繁 *k* 项集, 结构与 *R* 相同. 之后, 将 *TDS* 中包含的所有 1 项集放入 *itemsets* 中 (行 1~3). 对 *itemsets* 中每个 1 项集 *x* 对应的 *R* (*x*), 通过 *TDS* 映射得到 TS^x, 并计算得到 Sup^x(行 5~6). 如果 $Sup^x\geqslant minSup$, 根据定义 15, *x* 是一个频繁项集, 将 (*x*, *R*(*x*)) 加入到 $Candidate_k$ 中 (行 8)(**第 1 阶段**). 对 TS^x 根据 *Eps* 和 *minPts* 使用 DBSCAN 进行划分并得到 P^x, 计算 NR^x. 如果 $NR^x\leqslant maxNR$ 那么 *x* 是一个聚簇频繁模式, 通过 C^x 得到 L^x 并输出 *x* 和 L^x(行 9~11)(**第 2 阶段**). 最后, 将 $Candidate_k$ 复制到 *Candidates* 中 (行 14). 算法通过 while 循环重复调用 Naive-Generate 生成更大项集, 每次调用 Naive-Generate 时, 它都会组合 *k* 项集 ($k\geqslant 1$) 来生成 (k+1) 项集. Naive-Generate 调用结束后会将满足频繁条件的项集存储到 *Candidates* 中, while 结束条件是

|*Candidates*|≤1(行 15 ~18).

算法 2 给出了 Naive-Generate 算法的执行情况, 具体描述如下:

算法 2. Naive-Generate 算法.

输入:*Candidates*, *minSup*, *Eps*, *minPts*, *maxNR*, *minCF*, *TDS*;
输出: 长度为 (*k*+1) 的聚簇频繁模式.

1. $Candidate_{k+1}$=∅; //用于存储长度为 (*k*+1) 频繁项集;
2. 将 *Candidates* 中包含的所有 *k* 项集放入 *itemsets* 中;
3. **foreach** *Px*∈*itemsets* **do**
4. **foreach** *Py*∈*itemsets* 且 *Px*≠*Py* **do**
5. *tid*(*Pxy*)=*Candidates*(*Px*)∩*Candidates*(*Py*); //$Candidate_k$(*Px*) 和 $Candidate_k$(*Py*) 表示 *Px*, *Py* 对应事务 *ID* 集合;
6. TS^{Pxy}=*map*(*TDS*, *tid*(*Pxy*));
7. Sup^{Pxy}=|TS^{Pxy}|/|*TDS*|;
8. **if** Sup^{Pxy}≥*minSup* **then**
9. 将 (*Pxy*, *tid*(*Pxy*)) 加入到 $Candidate_{k+1}$ 中;
10. P^{Pxy}=C^{Pxy} ∪ Noi^{Pxy}=DBSCAN(TS^{Pxy}, *Eps*, *minPts*);
11. NR^{Pxy}=|Noi^{Pxy}|/|TS^{Pxy}|;
12. **if** NR^{Pxy}≤*maxNR* **then** 通过 C^{Pxy} 计算得到 L^{Pxy} 并且输出 *Pxy* 及对应 L^{Pxy};
13. **end if**
14. **end foreach**
15. **end foreach**
16. **return** $Candidate_{k+1}$

Naive-Generate(算法 2), 将 *Candidates*, *minSup*, *Eps*, *minPts*, *maxNR*, *minCF*, *TDS* 作为输入, 输出长度为 (*k*+1) 的聚簇频繁模式. 算法首先初始化 $Candidate_{k+1}$ 为空, $Candidate_{k+1}$ 用来存储频繁 *k*+1 项集, $Candidate_{k+1}$ 的结构与算法 1 中的 *R* 相同. 之后, 将 *Candidates* 中的项集放入 *itemsets* 中 (行 1~2). 接下来, 执行双重循环组合 2 个不同的 *k* 项集以生成 (*k*+1) 项集 (行 3~4). 将 *Px* 和 *Py* 对应的 *Candidates*(*Px*) 和 *Candidates*(*Py*) 取交集后得到 *tid*(*Pxy*), 对 *tid*(*Pxy*) 通过 *TDS* 映射得到 TS^{Pxy}, 并计算得到 Sup^{Pxy}(行 5~7). 接下来, 如果 Sup^{Pxy}≥*minSup*, 根据定义 15, *Pxy* 是一个频繁项集, 将 *Pxy* 及 *tid*(*Pxy*) 转为哈希结构 (*Pxy*, *tid*(*Pxy*)) 并加入到 $Candidate_{k+1}$ 中 (**第 1 阶段**)(行 8~9). 下一步, 对 TS^{Pxy} 根据 *Eps* 和 *minPts* 使用 DBSCAN 进行划分并得到 P^{Pxy}, 并计算 NR^{Pxy}. 如果 NR^{Pxy}≤*maxNR*, 那么 *Pxy* 是一个聚簇频繁模式, 通过 C^{Pxy} 得到 L^{Pxy} 并输出 P^{Pxy} 和 L^{Pxy}(**第 2 阶段**)(行 10~12). 最后算法返回 $Candidate_{k+1}$, 用来下一次调用 Naive-Generate 时生成 (*k*+2) 项集 (行 16).

3.1.2 算法复杂度分析

(1) 时间复杂度

算法首先基于包含 *m* 个事务的时间事务数据集 *TDS* 得到 *R*, 对应的时间复杂度为 O(*m*×*w*). 其中, *w* 为每个事务平均长度. 之后, 对 *TDS* 中每个 1 项集做聚簇频繁模式判断. 对项集 *x*∈*itemsets*, 令 *R*(*x*) 长度为 *r*, 则映射为 TS^x 的时间复杂度为 O(*r*); 求 Sup^x 对应的时间复杂度为 O(1); 如果项集满足频繁, 则可继续向下判断, DBSCAN 算法在最坏情况下的时间复杂度为 O(r^2); 由于计算 L^x 须遍历 1 遍 C^x, 令|C^x|=*s*, 则对应的时间复杂度为 O(*s*). 综上, 对每个 1 项集做聚簇频繁模式判断的时间复杂度为 O(r+r^2+s). 之后, 调用 Naive-Generate 生成 (*k*+1) 项集, 直到无法生成更多项集. 其中, *k*≥1. 如果 *itemsets* 中包含 *g* 个 *k* 项集, 那么所有 *g* 个 *k* 项集将组合为 (*g*×(*g*-1))/2 对项集以生成 (*k*+1) 项集. 对每一对项集 *Px* 和 *Py*, 将其对应的 *Candidates* (*Px*) 和 *Candidates* (*Py*) 取交集得到 *tid*(*Pxy*), 对应的时间复杂度为 O(r^2). 接下来的判断过程与 1 项集类似, 可得对每个 (*k*+1) 项集做聚簇频繁模式判断的时间复杂度

为 $O(r^2+r+r^2+s)$. 算法利用了上述过程探索了项集的搜索空间. 在最坏情况下, 所有 1 项集及生成的 (k+1) 项集都满足频繁, 最终会生成 $2^{(n-1)}$ 个项集, 总的时间复杂度为 $O(m\times w+n\times((r+r^2+s)+(2^{(n-1)}-n)\times(r^2+r+r^2+s))$. 其中, n 为 *TDS* 包含不同 1 项集总数. 算法可以根据参数的设置, 减少搜索空间, 进而减少时间复杂度.

(2) 空间复杂度

算法首先基于 *TDS* 得到 R, 占用空间为 $O(m)$; *itemsets* 占用空间为 $O(n)$. 对项集 $x\in itemsets$, TS^x 占用的空间为 $O(r)$, 如果项集满足频繁, 则可继续向下判断, P^x 占用的空间为 $O(r)$; L^x 只需要存储 C^x 的发生区间, 其占用的空间为 $O(d)$. 其中, d 为 C^x 包含簇总数. 综上, 对每个 1 项集做聚簇频繁模式判断的空间复杂度为 $O(r+r+d)$. 之后, 调用 Naive-Generate 算法求 (k+1) 项集. 在 Naive-Generate 算法生成 (k+1) 项集过程中, 算法只需在内存中保留 *Candidates* 和 $Candidate_{k+1}$. 在最坏情况下, 生成的所有 (k+1) 项集都可以作为聚簇频繁模式候选项集, 假设 *Candidates* 包含 g 个 k 项集, *Candidates* 占用空间为 $O(g\times r)$; *itemsets* 占用空间为 $O(g)$; $Candidate_{k+1}$ 中包含 $(g\times(g-1))/2$ 个 (k+1) 项集, 其占用空间为 $O(g^2\times r)$. 对每个 (k+1) 项集进行聚簇频繁模式判断与 1 项集过程类似, 空间复杂度都为 $O(r+r+d)$. 在最坏情况下, 所有 1 项集及生成的 (k+1) 项集都满足频繁, 最终会生成 $2^{(n-1)}$ 个项集, 总的时间复杂度为 $O(m+n+2^{(n-1)}\times(r+r+d)+(n-1)\times(g\times r+g+g^2\times r))$. 其中, $(n-1)\times(g\times r+g+g^2\times r)$ 的 (n-1) 表示 Naive-Generate 算法迭代次数. 算法可以根据参数的设置, 减少搜索空间, 进而减少空间复杂度.

3.2 算法优化

3.2.1 优化策略

Naive 算法的第 2 阶段需要无差别地对每一个频繁项集对应的时间集合使用 DBSCAN 算法, 这极大地影响了算法的时间效率, 有必要对该操作进行优化. 使用向下闭包性是频繁模式进行优化的首选策略, 但聚簇频繁模式并不具有向下闭包的特点 (即一个项集如果不是聚簇频繁项集, 则其超集必不是聚簇频繁项集), 我们可以通过如下实例来说明这点: 考虑表 1 所示 *TDS*, 给定 *minSup*=0.3, *maxNR*=0.3, *Eps*=2(天), *minPts*=3(天). 对 1 项集{b}求得 $Sup^{\{b\}}$=0.6>*minSup*, 之后, 对 $TS^{\{b\}}$进行划分得到 $C^{\{b\}}=\{c_1\}=\{[12, 13, 13, 14, 14]\}$, $Noi^{\{b\}}=[7, 8, 18, 20]$. 计算得 $NR^{\{b\}}$=0.44>*maxNR*, 所以, {b}不是聚簇频繁项集. 同理对 2 项集{b,f}进行判断, 可得该项集是聚簇频繁模式, 所以向下闭包性并不成立. 聚簇频繁模式不具有向下闭包的特点使我们无法严格从该角度出发对 Naive 算法进行优化. 通过该算法执行过程进行研究, 我们得到了如下结论.

引理 1. 任给项集 x', x, 考虑对 TS^x 和 $TS^{x'}$划分后得到的 C^x 和 $C^{x'}$, 如果 $x'\subset x$, 则有$|C^x|\leqslant|C^{x'}|$成立.

证明:由于 $x'\subset x$, 根据 Apriori 性质, 任何包含 x'的事务也必然包含 x, 可得 $T^x\subset T^{x'}$. 根据定义 4, 可推得 $TS^x\subset TS^{x'}$, 且$\forall t\in TS^x$, 都有 $t\in TS^{x'}$. 根据定义 14 可知簇的形成是基于 *Eps* 和 *minPts*, 且 C^x 和 $C^{x'}$中的簇分别由 TS^x 和 $TS^{x'}$中的时间戳构成, 所以, $\forall c2\in C^x$, $\exists c1\in C^{x'}$, 使得 $c2\subset c1$. 因此, 可知$|c2|\leqslant|c1|$, 进而可得$|C^x|\leqslant|C^{x'}|$. **证毕**.

引理 2. 令 *minCF*=*minSup*×|*TDS*|×(1-*maxNR*), 任给频繁项集 x, 我们有如下结论成立: 如果$|C^x|<$*minCF*, 那么 x 不是聚簇频繁模式.

证明:由于$|C^x|<$*minCF*, 根据 *minCF*=*minSup*×|*TDS*|×(1-*maxNR*), 可得$|C^x|<$*minSup*×|*TDS*|×(1-*maxNR*), 由于 x 是频繁项集, 根据定义 15 可知, $Sup^x\geqslant$*minSup*, 所以可以得到$|C^x|<Sup^x\times|TDS|\times(1-maxNR)$. 对上式比较符右边括号展开, 并重新组合比较符左右两边, 可得 $Sup^x\times|TDS|-|C^x|>maxNR\times Sup^x\times|TDS|$. 根据定义 14 和定义 15, 将比较符左右两边进行替换, 可得$|Noi^x|>maxNR\times|TS^x|$. 根据定义 14, 将比较符左右两边化简, 可得 $NR^x>maxNR$. 由定义 16 可知, 当 $NR^x>maxNR$ 时, x 不是聚簇频繁模式. **证毕**.

引理 1 给出了项集 x'与包含 x'的项集 x 在划分发生时间后得到的聚簇内容数$|C^x|$与$|C^{x'}|$间的关系, 引理 2 得到了一个与新变量 *minCF* 相关的非聚簇频繁模式的判断条件, 具体内容可通过如下实例说明.

考虑表 1 所示 *TDS*, 给定 *minSup*=0.3, *maxNR*=0.1, *Eps*=2(天), *minPts*=3(天), 计算得到 *minCF*=4.5. 对 1 项集{b}求得 $Sup^{\{b\}}$=0.6>*minSup*, 之后对 $TS^{\{b\}}$进行划分得到 $C^{\{b\}}=\{c_1\}=\{[12, 13, 13, 14, 14]\}$, 对 2 项集{$b$, e}求得 $Sup^{\{b, e\}}$=0.3$\geqslant$*minSup*, 之后对 $TS^{\{b, e\}}$进行划分得到 $C^{\{b, e\}}=\{c_1\}=\{[13, 13, 14, 14]\}$, 可得$|C^{\{b, e\}}|\leqslant|C^{\{b\}}|$. 对 2 项集{$b$,$f$}求得 $Sup^{\{b,f\}}$=0.47$\geqslant$*minSup*, 之后对 $TS^{\{b,f\}}$进行划分得到 $C^{\{b,f\}}=\{c_1\}=\{[12, 13, 13, 14, 14]\}$, 可得$|C^{\{b,f\}}|\leqslant|C^{\{b\}}|$(**引理 1**). 对 2 项集{$b$,

e}, 求得 $Noi^{\{b,e\}}=[ts_{14}]$, $NR^{\{b,e\}}=0.2>maxNR$, 可知{b, e}不是聚簇频繁模式, 同时计算得到$|C^{\{b,e\}}|<minCF$(**引理 2**).

由引理 1 和引理 2 不难推得下面定理 1 和定理 2 成立.

定理 1. 对于频繁项集 $x'\subset x$, 如果$|C^{x'}|<minCF$, 则任何超集 $x\supset x'$都不是聚簇频繁模式.

证明:由于$|C^{x'}|<minCF$, 根据引理 2, 可得 x'不是聚簇频繁模式. 根据引理 1, 可知$|C^{x}|\leqslant|C^{x'}|$, 所以, 可得$|C^{x}|<minCF$. 根据引理 2, 可得 x 不是聚簇频繁模式. 因此, 如果 x'不是聚簇频繁模式, 则任何超集 $x\supset x'$都不是聚簇频繁模式. **证毕**.

定理 2. 对项集 $x'\subset x$, 如果 $\exists x''\subset x'$且$|C^{x''}|<minCF$, 则 x'不是聚簇频繁模式且任何超集 $x\supset x'$都不是聚簇频繁模式.

证明:由于$|C^{x''}|<minCF$, 根据引理 2, x''不是聚簇频繁模式. 根据引理 1, 可知$|C^{x}|\leqslant|C^{x'}|\leqslant|C^{x''}|$. 因此, 可得$|C^{x'}|$和$|C^{x}|$都要小于 $minCF$. 所以, 根据引理 2, 可得 x'不是聚簇频繁模式且任何超集 $x\supset x'$都不是聚簇频繁模式. **证毕**.

定理 1 和定理 2 给出了在 Naive 算法第 2 阶段实施优化的理论基础, 其中定理 1 能够在结果搜索的过程中, 有效减小搜索空间; 定理 2 可以在确定 n 项聚簇频繁项集结果的时候, 参考 (n-1) 项集的判别结果, 可以减少冗余判别操作. 这 2 个定理能够对第 1 阶段的执行结果执行很好地优化. 具体可以通过如下实例来说明.

考虑表 1 所示 *TDS*, 给定 $minSup$=0.3, $maxNR$=0.1, Eps=2(天), $minPts$=3(天), 计算得到 $minCF$=4.5. 考虑 2 项集{d, f}, 求得 $Sup^{\{d,f\}}=0.3\geqslant minSup$, 之后对 $TS^{\{d,f\}}$进行划分得到 $C^{\{d,f\}}=\{c_1\}=\{[13, 14, 14]\}$, $Noi^{\{d,f\}}=[20, 20]$, $NR^{\{d,f\}}=0.4>maxNR$, 则{d,f}不是聚簇频繁模式, 且$|C^{\{d,f\}}|=3<minCF$. 同理, 可得{d,f}在 *TDS* 中长度为 3 的超集{a, d,f}, {b, d,f}, {c, d,f}, {d, e,f}都不是聚簇频繁模式 (**定理 1**), 且{d,f}长度为 3 的所有超集对应长度为 4 的超集也都不是聚簇频繁模式 (**定理 2**).

3.2.2 数据结构

Naive 算法本质上使用了 Apriori-TID 算法思想生成候选项集, 算法记录了每个 k 项集发生的事务 *ID* 集合, 通过求 2 个 k 项集发生事务的 *ID* 集合交集来确定产生于它们的 (k+1) 项集发生事务的 *ID* 集合. 之后基于发生事务 *ID* 集合, 找该 (k+1) 项集的发生时间集合. 这个过程中, 随着数据规模变大, 每个 k 项集发生事务的 *ID* 集合体量也会变大, 占用较多存储空间, 另外在计算 (k+1) 项集的发生事务 *ID* 集合时, 大量集合的交集运算会更加耗时. 本部分借鉴了文献 [7] 中使用位向量表示时间集合的思路, 提出了 *ICFPM-list* 的数据结构. 可以解决上述问题. 另外为了方便基于项集发生事务 *ID* 集合找该项集的发生时间集合, 我们设计了 *bvTimeMap* 结构.

定义 17.**项集 x 的发生位置向量 $bitvec_x$**. 考虑包含 w 个事务的 *TDS* 和项集 x, $bitvec_x$ 是一个长度为 w, 且用于标识 x 发生事务 *ID* 的向量. $bitvec_x[i](1\leqslant i\leqslant w)$ 表示 $bitvec_x$ 中第 i 个成员, 当 x 在 *ID* 值为 i 的事务中发生, 则 $bitvec_x[i]$ 值为 1, 否则值为 0.

例 17. 考虑表 1 所示 *TDS*, 1 项集{a}对应 $bitvec_{\{a\}}$可以表示为位向量 1010100101001l0.

定义 18.***ICFPM-list***. 考虑包含 w 个事务的 *TDS* 和项集 x, *ICFPM-list* 是一个结构为 (x, $bitvec_x$) 的哈希表, 其中 x 是哈希表的 *key*, $bitvec_x$ 是哈希表的 *value*.x 对应的 *ICFPM-list* 记为 $ICFPM\text{-}list^x$, 使用 *ICFPM-list*(x) 表示 $bitvec_x$.

例 18. 接着考虑例 17, $ICFPM\text{-}list^{\{a\}}$=({$a$}, 101 010 010 100 110), *TDS* 中所有 1 项集对应 *ICFPM-list* 如图 1 所示.

Key	Value
{a}	1 0 1 0 1 0 0 1 0 1 0 0 1 1 0
{b}	0 0 1 1 1 1 1 1 1 0 0 1 0 1 0
{c}	1 0 0 0 0 1 0 0 1 0 1 0 1 0 0
{d}	0 1 0 1 0 0 1 1 1 1 1 0 0 1 1
{e}	0 0 0 0 0 1 1 1 1 1 0 0 1 1 0
{f}	0 0 0 0 1 1 1 1 1 0 0 1 1 1 1

图 1 位向量表示项集及对应发生位置

ICFPM-list 利用位向量存储不同项集的发生位置, 可以减少用于存储每个 k 项集发生事务 *ID* 集合的开销, 同时也便于计算不同项集发生位置的交集, 可以加速获得 (k+1) 项集发生事务 *ID* 集合的过程. 另外为了便于进一步确定项集的发生时间, 定义了 *bvTimeMap*.

定义 19.***bvTimeMap***[7]. 考虑包含 w 个事务的 *TDS*, *bvTimeMap* 是一个长度为 w 的数组. *bvTimeMap*[i]=a 表示 *ID* 为 (i+1) 的事务数据中项集的发生时间为 a.

例 19. 考虑表 1 所示 *TDS*, 由定义 19 不难得到与其相关的 *bvTimeMap*, 具体如图 2 所示.

时间戳	6	7	7	8	12	12	13	13	14	15	17	18	19	20	20
下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

图 2　表 1 对应 *bvTimeMap*

3.3　改进的聚簇频繁模式挖掘算法

3.3.1　算法描述

基于优化策略与数据结构部分的内容对 Naive 算法进行改造, 可以得到改进后的聚簇频繁模式挖掘算法 ICFPM(Improved Cluster Frequent Pattern Mining, ICFPM). 具体如算法 3 所示.

算法 3: ICFPM 算法

输入:*TDS*: 包含 w 个事务的时间有序事务数据集, *minSup*, *Eps*, *minPts*, *maxNR*;
输出: 所有聚簇频繁模式.

1. 得到 *TDS* 中所有 1 项集对应的 *ICFPM-list* 并且计算 *minCF*, *bvTimeMap*;
2. $new\text{-}Candidate_k$=∅, k=1; //$new\text{-}Candidate_k$ 用于以 *ICFPM-list* 结构存储长度为 k 的聚簇频繁模式候选项集;
3. 将 *TDS* 中所有 1 项集放入 *itemsets*;
4. **foreach** x∈*itemsets* **do** //对 *itemsets* 中每个 1 项集 x;
5. TS^x=*map*(*bvTimeMap, ICFPM-list*(x)); //根据 *bvTimeMap* 将 *ICFPM-list*(x) 映射为 TS^x;
6. Sup^x=$|TS^x|/|TDS|$;
7. **if** $Sup^x \geqslant minSup$ **then**
8. $P^x=C^x \cup Noi^x$=DBSCAN(TS^x, *Eps*, *minPts*); //对 TS^x 进行划分;
9. $NR^x=|Noi^x|/|TS^x|$;
10. **if** $NR^x \leqslant maxNR$ **then** 通过 C^x 计算得到 L^x 并且输出 x 及对应 L^x;
11. **if** $|C^x| \geqslant minCF$ **then** 将 $ICFPM\text{-}list^x$ 加入到 $new\text{-}Candidate_k$ 中; //**定理 1**
12. **end if**
13. **end foreach**
14. $new\text{-}Candidates=new\text{-}Candidate_k$;
15. **while** |*new-Candidates*|>1 **do** //|*new-Candidates*|表示包含候选项集总数;
16. $new\text{-}Candidate_{k+1}$=new-Generate(*new-Candidates*, *minSup*, *Eps*, *minPts*, *maxNR*, *minCF*, *bvTimeMap*, |*TDS*|);
17. $new\text{-}Candidates=new\text{-}Candidate_{k+1}$;
18. **end**

ICFPM 算法首先基于 *TDS* 得到所有 1 项集的 *ICFPM-list*, 并计算得到 *minCF* 和 *bvTimeMap*. 初始化 $new\text{-}Candidate_k$ 为空, k=1, $new\text{-}Candidate_k$ 与 *ICFPM-list* 结构相同, 用来存储所有长度为 1 的聚簇频繁模式候选项集对应的 *ICFPM-list*, 并将 *TDS* 中包含的所有 1 项集放入到 *itemsets* 中 (行 1~3). 遍历 *TDS* 中的每个 1 项集 x, 将其 *ICFPM-list*(x) 通过 *bvTimeMap* 映射得到 TS^x, 并计算得到 Sup^x(行 5~6). 接下来, 如果 $Sup^x \geqslant minSup$, 那么根据定义

15, x 是一个频繁项集, 对 TS^x 根据 *Eps* 和 *minPts* 使用 DBSCAN 进行划分并得到 P^x, 并计算 NR^x. 如果 $NR^x \leqslant maxNR$, 那么 x 是一个聚簇频繁模式, 通过 C^x 得到 L^x 并输出 x 和 L^x(行 7~10). 接下来, 如果 $|C^x| \geqslant minCF$, 根据定理 1, x 是一个聚簇频繁模式候选项集, 将 $ICFPM\text{-}list^x$ 加入 $new\text{-}Candidate_k$ 中 (行 11). 最后, 将 $new\text{-}Candidate_k$ 复制到 *new-Candidates* 中 (行 14), 算法通过 while 循环重复调用 new-Generate 生成更大项集. 每次调用时, new-Generate 都会组合 k 项集 ($k \geqslant 1$) 来生成 (k+1) 项集, new-Generate 结束后会将满足频繁的项集存储到 *new-Candidates* 中, while 结束条件是$|new\text{-}Candidates| \leqslant 1$(行 15~18).

算法 4 给出了 new-Generate 算法的具体执行情况, 具体如下所示.

算法 4: new-Generate 算法

输入:*new-Candidates*, *minSup*, *Eps*, *minPts*, *maxNR*, *minCF*, *bvTimeMap*, |*TDS*|;
输出: 长度为 (k+1) 的聚簇频繁模式.

1. $new\text{-}Candidate_{k+1}=\varnothing$; //用于以 *ICFPM-list* 结构存储所有长度为 (k+1) 聚簇频繁模式候选项集;
2. 将 *new-Candidates* 中包含的所有 k 项集放入 *itemsets* 中;
3. **foreach** $Px \in itemsets$ **do**
4. **foreach** $Py \in itemsets$ 如果满足 $y > x$ 且 Px, Py 有相同长度为 (k-1) 的前缀 P **do**
5. $bitvec_{Pxy}=new\text{-}Candidates(Px) \cap new\text{-}Candidates(Py)$; //*new-Candidates*(*Px*) 和 *new-Candidates* (*Py*) 表示 *Px*, *Py* 对应的位向量;
6. $TS^{Pxy}=map(bvTimeMap, bitvec_{Pxy})$;
7. **if** $SPxy \subset Pxy$ 且$|SPxy|=|Pxy|$-1, $SPxy \in itemsets$ **then** //**定理 2**
8. $Sup^{Pxy}=|TS^{Pxy}|/|TDS|$;
9. **if** $Sup^{Pxy} \geqslant minSup$ **then**
10. $P^{Pxy}=C^{Pxy} \cup Noi^{Pxy}$=DBSCAN($TS^{Pxy}$, *Eps*, *minPts*);
11. $NR^{Pxy}=|Noi^{Pxy}|/|TS^{Pxy}|$;
12. **if** $NR^{Pxy} \leqslant maxNR$ **then** 通过 C^{Pxy} 计算得到 L^{Pxy} 并且输出 *Pxy* 及对应 L^{Pxy};
13. **if** $|C^{Pxy}| \geqslant minCF$ **then** 将 *Pxy*, $bitvec_{Pxy}$ 以 $ICFPM\text{-}list^{Pxy}$ 形式加入 $new\text{-}Candidate_{k+1}$; //**定理 1**
14. **end if**
15. **end if**
16. **end foreach**
17. **end foreach**
18. **return** $new\text{-}Candidate_{k+1}$

new-Generate 以包含一组 k 项集的 *ICFPM-list*(记为 *new-Candidates*), *minSup*, *Eps*, *minPts*, *maxNR*, *minCF*, *bvTimeMap*, |*TDS*|作为输入, 输出长度为 (k+1) 的聚簇频繁模式. 算法首先初始化 $new\text{-}Candidate_{k+1}$ 为空, $new\text{-}Candidate_{k+1}$ 用于存储长度为 (k+1) 的聚簇频繁模式候选项集对应的 *ICFPM-list*, 并将 *new-Candidates* 中包含的所有项集放入 *itemsets* 中 (行 1~2). 接下来, 执行双重循环组合 k 项集对以生成 (k+1) 项集, 令项集的总顺序为 >. 如果 2 个 k 项集 Px 和 Py 拥有相同长度为 (k-1) 的前缀 P, 且满足 $y > x$, 则将这 2 个项集组合为 (k+1) 项集 Pxy(行 2~3). 接下来, 将 Px 和 Py 对应的 *new-Candidates*(*Px*) 和 *new-Candidates*(*Py*) 先取交集后得到 Pxy 对应位向量 $bitvec_{Pxy}$, 之后对 $bitvec_{Pxy}$ 通过 *bvTimeMap* 映射得到 TS^{Pxy}(行 5~6). 根据定理 2, 检查 Pxy 的所有长度为 k 的子项集是否都是聚簇频繁模式 (在 *itemsets* 中), 如果存在子项集不满足, 则 Pxy 及超集都不是聚簇频繁模式 (行 7). 否则, 与算法 3 类似, 对 Pxy 进行聚簇频繁模式判断 (行 8~12). 根据定理 1, 将满足条件的 Pxy 及对应 $bitvec_{Pxy}$ 先转为 $ICFPM\text{-}list^{Pxy}$, 再将 $ICFPM\text{-}list^{Pxy}$ 存储到 $new\text{-}Candidate_{k+1}$ 中 (行 13). 最后, 返回 $new\text{-}Candidate_{k+1}$, 用来下一次调用 new-Generate 时生成 (k+2) 项集 (行 18).

3.3.2 运行实例

考虑表 1 所示 *TDS*, 给定 *minSup*=0.3, *maxNR*=0.3, *Eps*=2(天), *minPts*=3(天). ICFPM(算法 3) 首先将 *TDS* 转换为如图 1 所示 *ICFPM-list*, 并得到如图 2 所示 *bvTimeMap*, *minCF*=2.1. 将 1 项集{*f*}对应 $ICFPM\text{-}list^{\{f\}}$ 通过 *bvTimeMap* 映射得到 $TS^{\{f\}}$=[12, 13, 13, 14, 14, 18, 19, 20, 20], 计算得到 $Sup^{\{f\}}$=9/15=0.6>*minSup*. 接下来对 $TS^{\{f\}}$ 根据 *Eps* 和 *minPts* 使用 DBSCAN 算法进行划分, 得到 $C^{\{f\}}=\{c_1, c_2\}=\{[12, 13, 13, 14, 14], [18, 19, 20, 20]\}$, $Noi^{\{f\}}$=[]. 计算得到 $NR^{\{f\}}$=0<*maxNR*, 所以, {*f*}是一个聚簇频繁模式, 计算得到 $L^{\{f\}}=\{l_1, l_2\}=\{[12, 14], [18, 20]\}$并输出{*f*}和 $L^{\{f\}}$. 由于$|C^{\{f\}}|$>*minCF*, 根据定理 1, {*f*}是一个聚簇频繁模式候选项集, 将 $ICFPM\text{-}list^{\{f\}}$存储到 $new\text{-}Candidate_1$ 中. 同理, 对其他项做聚簇频繁模式判断, 得到 1 项集{*e*}和{*f*}是聚簇频繁模式且都可以作为聚簇频繁模式候选项集;{*b*}和{*d*}不是聚簇频繁模式但可以作为聚簇频繁模式候选项集. 将 $new\text{-}Candidate_1$ 复制到 *new-Candidates* 中, 由于|*new-Candidates*|>1, 所以, 调用 new-Generate(算法 4).

接下来进入 new-Generate 后, 首先初始化 $new\text{-}Candidate_2$ 为空, 通过 *new-Candidates* 得到 *itemsets*={*b*, *d*, *e*, *f*}. 由于{*d*}和{*e*}有相同前缀{∅}, 且{*d*}>{*e*}, 所以, 可组合为 2 项集{*d*, *e*}. 将对应 *new-Candidates*({*d*}) 和 *new-Candidates*({*e*}) 取交集得到 $bitvec_{\{d, e\}}$=000000111100010, 映射得到 $TS^{\{d, e\}}$=[13, 14, 14, 15, 20]. 由于{*d*, *e*}的所有长度为 1 的子集{*d*}和{*e*}都在 *itemsets* 中, 根据定理 2, {*d*, *e*}及其超集有可能是聚簇频繁模式, 可以继续向下判断. 接下来, 计算得到 $Sup^{\{d, e\}}$=0.3>*minSup*, 对 $TS^{\{d, e\}}$使用 DBSCAN 算法划分得到 $C^{\{d, e\}}=\{c_1\}=\{[13, 14, 14, 15]\}$, $Noi^{\{d, e\}}$=[20], $NR^{\{d, e\}}$=0.1<*maxNR*. 所以, {*d*, *e*}是一个聚簇频繁模式, 计算 $L^{\{d, e\}}$并输出. 由于 $C^{\{d, e\}}\geqslant$*minCF*, 根据定理 1, {*d*, *e*}是一个聚簇频繁模式候选项集, 将{*d*, *e*}和 $bitvec_{\{d, e\}}$组合为 $ICFPM\text{-}list^{\{d, e\}}$并加入 $new\text{-}Candidate_2$ 中. 同理, 对其他 1 项集进行组合, 对得到 2 项集并做聚簇频繁模式判断. 最后, 得到{*d*, *e*}和{*b*, *f*}是聚簇频繁模式且都可以作为聚簇频繁模式候选项集;{*b*, *e*}和{*e*, *f*}不是聚簇频繁模式但可以作为聚簇频繁模式候选项集加入 $new\text{-}Candidate_2$ 中. 最后, 将 $new\text{-}Candidate_2$ 作为返回值. 根据 new-Generate 得到 $new\text{-}Candidate_2$ 并复制到 *new-Candidates*, 由于|*new-Candidates*|>1, 根据算法 3(行 16), 继续调用 new-Generate 与上述过程相同. 最后, 得到只有{*b*, *e*, *f*}是聚簇频繁模式且可以作为聚簇频繁模式候选项集, 由于|*new-Candidates*|=1≯1, 根据算法 3(行 14), 算法结束.

3.3.3 算法复杂度分析

在第 3.1.2 节中定义了一些变量用于复杂度分析, 这些变量将在本节继续使用.

(1) 时间复杂度

ICFPM 算法首先将包含 *m* 个事务的时间事务数据集 *TDS* 转换为 *ICFPM-list*, 对应的时间复杂度为 O(*m*×*w*), 与 Naive 算法中得到 *R* 的时间复杂度相同. 之后, 对 *TDS* 中每个 1 项集做聚簇频繁模式判断. 对项集 *x*∈*itemsets*, 由于 *x* 对应的 *ICFPM-list*(*x*) 长度为 *m*, 所以, 映射为 TS^x 的时间复杂度为 O(*m*). 由 3.1.2 节可知 Naive 算法使用 *R* 结构做映射时对应的时间复杂度为 O(*r*), 且 *r*⩽*m*, 所以, 使用 *ICFPM-list* 结构做映射时的时间复杂度要多于 Naive 算法的 *R* 结构. 接下来, 求 Sup^x 对应的时间复杂度为 O(1). 如果项集满足频繁, 则可继续向下判断, DBSCAN 算法在最坏情况下的时间复杂度为 O(r^2); 由于计算 L^x 须遍历 1 遍 C^x, 对应时间复杂度为 O(*s*). 综上, 对每个 1 项集做聚簇频繁模式判断的时间复杂度为 O($m+r^2+s$). 之后调用 new-Generate 算法生成 (*k*+1) 项集, 直到无法生成更多项集. 如果 *itemsets* 中包含 *g* 个 *k* 项集, 那么所有 *g* 个 *k* 项集将组合为 (*g*×(*g*-1))/2 对项集以生成 (*k*+1) 项集. 对每一对项集 *Px* 和 *Py*, 将其对应的 *new-Candidates* (*Px*) 和 *new-Candidates* (*Py*) 取交集得到 $bitvec_{Pxy}$, 由于使用向量的位与操作, 对应的时间复杂度为 O(*m*). 由 3.1.2 节可知 Naive-Generate 算法做交集的时间复杂度为 O(r^2), 当 *new-Candidates*(*Px*) 和 *new-Candidates*(*Py*) 长度较长时, 使 O(*m*)⩽O(r^2), 这时, 使用 *ICFPM-list* 结构要好于 *R* 结构. 接下来的判断过程与 1 项集类似, 可得对每个 (*k*+1) 项集做聚簇频繁模式判断的时间复杂度为 O($m+m+r^2+s$).

在最坏情况下, 所有 1 项集及生成的 (*k*+1) 项集都满足频繁, Naive 算法最终会生成 $2^{(n-1)}$ 个项集. 然而, ICFPM 算法在 Naive 算法基础上使用了 2 种优化策略. 首先, 在频繁条件的基础上, 使用定理 1 过滤了更多不可能是聚簇频繁模式的项集, 减少了候选项集数量, 即 *new-Candidates*, 进而减少了搜索空间和时间复杂度; 其次, 定理 2 通过参考 (*n*-1) 项集的判别结果, 减少了 new-Generate 算法中对项集做聚簇频繁模式判断操作, 进而减少了时间复杂度. 综上, 2 种优化策都减少了 Naive 算法时间复杂度. 同时使用 *ICFPM-list* 结构减少了项集对应事务 *ID* 集

合做交集的时间复杂度, 但增加了做映射时的时间复杂度. 所以, ICFPM 算法总的时间复杂度为 $O(m\times w+n\times(m+r^2+s)+(2^{(n-1)}-n-u)\times(m+m+r^2+s))$, 其中, u 表示使用了 2 种优化策略后对比 Naive 算法减少的需要做聚簇频繁模式判断的项集数量. 算法可以根据参数的设置, 减少搜索空间, 进而减少时间复杂度.

(2) 空间复杂度

由于 ICFPM 算法在 Naive 算法基础上, 使用了 *ICFPM-list* 结构, 由于使用位向量表示事务 *ID* 集合, 所以, 所有 1 项集对应 *ICFPM-list* 共占用空间为 $O(m/8)$, 对比 *R* 结构减少了 8 倍空间消耗;*bvTimeMap* 占用空间为 $O(m)$;*itemsets* 占用空间为 $O(n)$. 对项集 $x\in itemsets$, TS^x 占用的空间为 $O(r)$;P^x 占用的空间为 $O(r)$, L^x 只需要存储发生位置区间, 占用的空间为 $O(d)$. 综上, 对每个 1 项集做聚簇频繁模式判断的空间复杂度为 $O(r+r+d)$. 之后, 调用 new-Generate 求 $(k+1)$ 项集. 在 new-Generate 算法生成 $(k+1)$ 项集过程中, 算法只需在内存中保留 *new-Candidates* 和 $new\text{-}Candidate_{k+1}$. 在最坏情况下, 生成的所有 $(k+1)$ 项集都可以作为聚簇频繁模式候选项集, 假设 *new-Candidates* 包含 g 个 k 项集, *new-Candidates* 占用空间为 $O(g\times m/8)$;*itemsets* 占用空间为 $O(g)$; $new\text{-}Candidate_{k+1}$ 中包含 $(g\times(g-1))/2$ 个 $(k+1)$ 项集, 其占用空间为 $O(g^2\times m/8)$. 对比 Naive-Generate 算法中的 *Candidates* 和 $Candidate_{k+1}$ 分别减少了 8 倍. 对每个 $(k+1)$ 项集进行聚簇频繁模式判断与 1 项集过程类似, 空间复杂度都为 $O(r+r+d)$.

在最坏情况下, 所有 1 项集及生成的 $(k+1)$ 项集都满足频繁, Naive 算法最终会生成 $2^{(n-1)}$ 个项集. 然而, ICFPM 算法在 Naive 算法基础上使用了 2 种优化策略, 减少了搜索空间和不必要的判断操作. 同时使用 *ICFPM-list* 结构减少了项集对应事务 *ID* 集合占用空间, 但增加了 *bvTimeMap* 的占用空间. 最终, 可得 ICFPM 算法总的空间复杂度为 $O(m/8+m+n+(2^{(n-1)}-u)\times(r+r+d)+(n-1-v)\times(g\times m/8+g+g^2\times m/8))$. 其中, v 表示 new-Generate 算法减少的迭代次数. 算法可以根据参数的设置, 减少搜索空间, 进而减少空间复杂度.

4 实验评测

本部分实验目的有 2 个:(1) 验证 ICFPM 算法中优化方法的有效性;(2) 对 ICFPM 算法在参数 *minSup*, *Eps*, *minPts*, *maxNR* 不同取值下的性能变化 (时间和空间) 情况进行评测. 由于目前并没有针对时间有序事务数据集上聚簇频繁模式挖掘的研究, 所以我们把 Naive 算法、使用 *ICFPM-list* 结构改造后的 Naive 算法 (记为 Naive-NoPrune 算法)、使用 *ICFPM-list* 结构和优化策略 1 改造后的 Naive 算法 (记为 Naive-T1 算法)、使用 *ICFPM-list* 结构和优化策略 2 改造后的 Naive 算法 (记为 Naive-T2 算法), 作为 ICFPM 算法的实验对比算法.

4.1 实验环境及数据集

本文实验中的算法都使用 Python 实现, 在 Windows 11 22H2 环境下运行. 计算机配置为 Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz. 实验使用 2 个真实数据集: MBA 和 Value-Inc. 其中 MBA 时间范围为 2010/01/12-2011/12/10, 包含 4 186 个不同的商品和 21 663 笔交易. Value-Inc 时间范围为 2018/02/12-2019/02/20, 包含 3 407 个不同的商品和 25 898 笔交易. 数据集可从 Market Basket Analysis (kaggle.com) 和 Sales Analysis for "Value Inc". Case Study Python | Kaggle 获得.

4.2 ICFPM 算法性能评价

4.2.1 ICFPM 算法运行时间分析

首先分析 *minSup* 的取值改变对算法运行时间的影响, 其他参数取值如下:*Eps*=5, *minPts*=10, *maxNR*=0.1, 算法在 *minSup* 变化时运行时间比较如图 3 所示.

由图 3 可以看到, 随着 *minSup* 增加, 所有算法运行时间都逐渐减少. 这是因为 *minSup* 越大, 算法找到的频繁项集越少, 时间复杂度和搜索空间越小, 从而减少了算法运行时间, 与时间复杂度分析结果相同. 同时, 可以在 2 个数据集上观察到在相同 *minSup* 取值下, ICFPM 算法相对于其他 4 种算法消耗时间总是最少, 这是因为 ICFPM 算法使用 2 种剪枝策略可以有效减少搜索空间和时间复杂度, 并且相对于 Naive-T1 算法和 Naive-T2 算法, ICFPM 算法同时减少了候选项集数量和不必要的判断. 还可以观察到, 使用 *ICFPM-list* 的 Naive-NoPrune 算法要好于 Naive 算法, 这是因为使用位向量表示事务 *ID* 集合可以更快的做交集, 与时间复杂度分析结果相同. 当 *minSup* 取

0.003 时, Naive 算法要比 ICFPM 多消耗约 100 倍时间.

分析 *Eps* 的取值变化对算法运行时间的影响, 其他参数取值如下:*minSup*=0.003, *minPts*=10, *maxNR*=0.1, 算法在 *Eps* 变化时运行时间比较如图 4 所示.

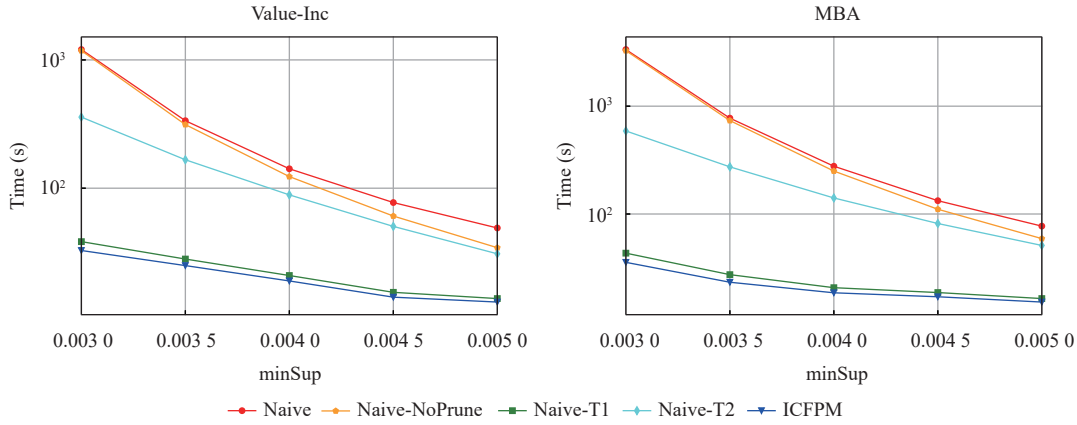

图 3 算法在 *minSup* 变化时运行时间比较

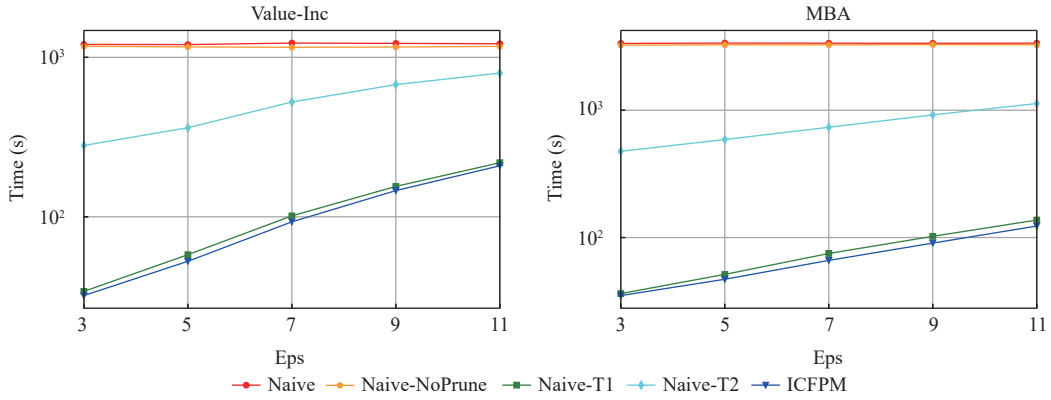

图 4 算法在 *Eps* 变化时运行时间比较

由图 4 可以看到, 在 2 个数据集上随着 *Eps* 取值增加, ICFPM 算法、Naive-T1 算法和 Naive-T2 算法运行时间逐渐增加. 这是因为 *Eps* 越大, 更多时间戳可以聚为一簇, 产生更多候聚簇频繁模式选项集, 时间复杂度和搜索空间增加, 从而增加了算法运行时间, 与时间复杂度分析结果相同. 而 Naive 算法和 Naive-NoPrune 算法的运行时间不随着 *Eps* 增加而改变, 这是因为只通过项集是否满足频繁进行剪枝, 之后需要对所有频繁项集进行聚类等操作, 并没有改变算法搜索空间, 时间复杂度不变, 所以算法运行时间也不会随着 *Eps* 增加而变化. 同时, 可以在 2 个数据集上观察到在相同 *Eps* 取值下, ICFPM 算法相对于其他 4 种算法消耗时间总是最少, 并且 Naive-NoPrune 算法要好于 Naive 算法, 与时间复杂度分析结果相同. 当 *Eps* 取 11 时, Naive 算法要比 ICFPM 多消耗约 100 倍时间.

分析 *minPts* 的取值变化对算法运行时间的影响, 其他参数取值如下: *Eps*=5, *minSup*=0.003, *maxNR*=0.1, 算法在 *minPts* 变化时运行时间比较如图 5 所示.

由图 5 可以看到, 在 2 个数据集上随着 *minPts* 取值增加, ICFPM 算法、Naive-T1 算法和 Naive-T2 算法运行时间逐渐减少. 这是因为 DBSCAN 要求至少 *minPts* 个时间戳才能成簇, *minPts* 越大, 簇的数量越少, 候选项集也越少, 时间复杂度和搜索空间减少, 从而减少了算法运行时间, 与时间复杂度分析结果相同. 而 Naive 算法和 Naive-NoPrune 算法的运行时间不随着 *minPts* 增加而改变, 这是因为只通过项集是否满足频繁进行剪枝, 之后需要对所

有频繁项集进行聚类等操作, 并没有改变算法搜索空间, 时间复杂度不变, 算法运行时间也不会随着 *minPts* 增加而变化. 同时, 可以在 2 个数据集上观察到在相同 *minPts* 取值下, ICFPM 算法相对于其他 4 种算法消耗时间总是最少, 并且 Naive-NoPrune 算法要好于 Naive 算法, 与时间复杂度分析结果相同. 当 *minPts* 取 8 时, Naive 算法要比 ICFPM 多消耗约 100 倍时间.

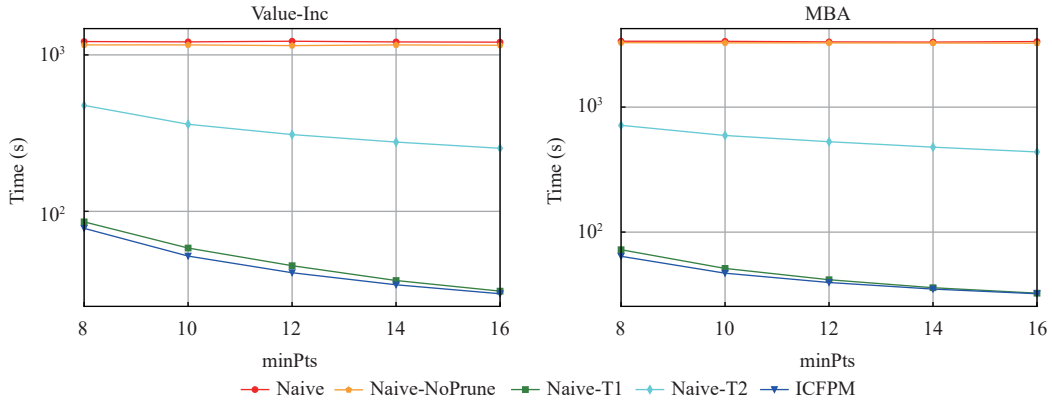

图 5 算法在 *minPts* 变化时运行时间比较

分析 *maxNR* 的取值变化对算法运行时间的影响, 其他参数取值如下:*Eps*=5, *minSup*=0.003, *minPts*=10, 算法在 *maxNR* 变化时运行时间比较如图 6 所示.

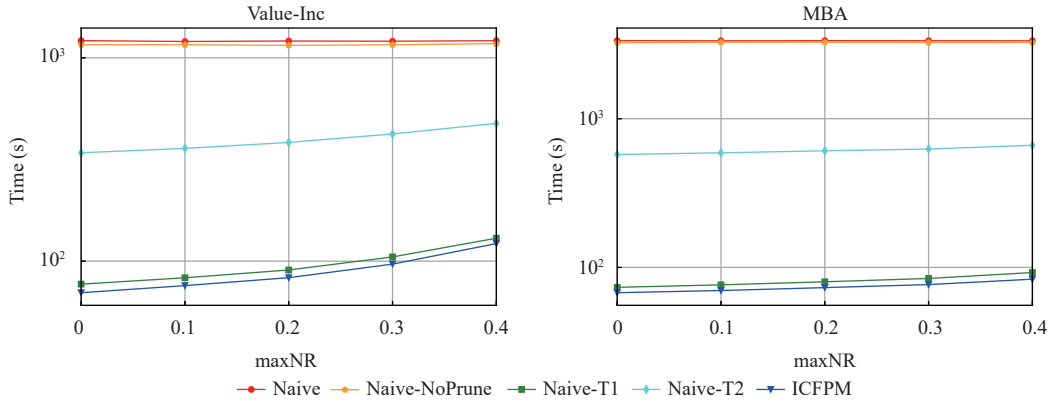

图 6 算法在 *maxNR* 变化时运行时间比较

由图 6 可以看到, 在 2 个数据集上随着 *maxNR* 取值增加, ICFPM 算法、Naive-T1 算法和 Naive-T2 算法运行时间逐渐增加. 这是因为 *maxNR* 越大, 允许存在的噪点越多, 产生更多候选项集, 时间复杂度和搜索空间增加, 从而增加了算法运行时间, 与时间复杂度分析结果相同. 而 Naive 算法和 Naive-NoPrune 算法的运行时间不随着 *minPts* 增加而改变, 这是因为只通过项集是否满足频繁进行剪枝, 之后需要对所有频繁项集进行聚类等操作, 并没有改变算法搜索空间, 时间复杂度不变, 算法运行时间也不会随着 *maxNR* 增加而变化. 同时, 可以在 2 个数据集上观察到在相同 *maxNR* 取值下, ICFPM 算法相对于其他 4 种算法消耗时间总是最少, 并且 Naive-NoPrune 算法要好于 Naive 算法, 与时间复杂度分析结果相同. 当 *maxNR* 取 0 时, Naive 算法要比 ICFPM 多消耗约 100 倍时间.

4.2.2 ICFPM 算法空间开销分析

首先分析 *minSup* 的取值改变对算法空间开销的影响. 其他参数取值如下:*Eps*=5, *minPts*=10, *maxNR*=0.1, 算法在 *minSup* 变化时空间开销比较如图 7 所示.

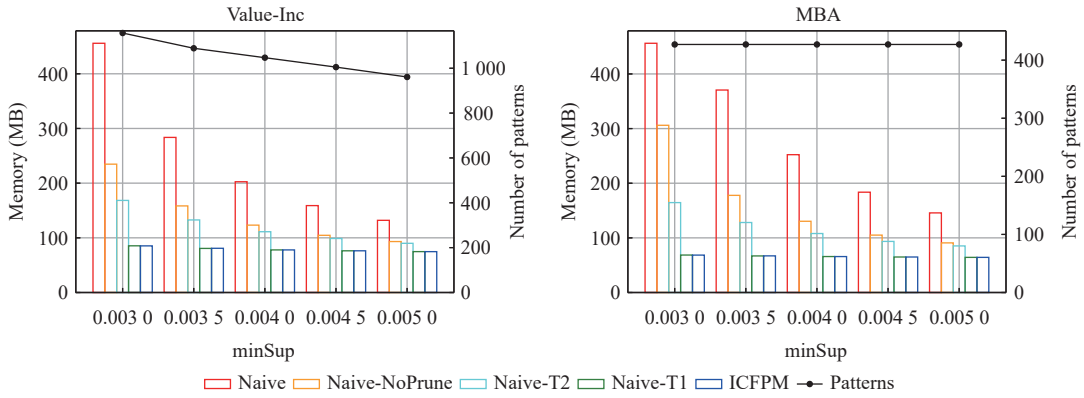

图 7　算法在 *minSup* 变化时空间开销比较

由图 7 可以看到, 在 2 个数据集上随着 *minSup* 增加, 所有算法空间开销和模式数量都逐渐减少. 这是因为 *minSup* 越大, 算法找到的频繁模式越少, 空间复杂度和搜索空间越小, 从而减少了算法空间开销和模式数量, 与空间复杂度分析结果相同. 同时, 可以在 2 个数据集上观察到在相同 *minSup* 取值下, ICFPM 算法相对于其他 4 种算法空间开销总是最少, 这是因为 ICFPM 算法通过 2 种剪枝略可以最大限度的减少搜索空间, 减少了不必要的计算和空间复杂度, 进而减少空间开销. 并且 Naive-NoPrune 算法要好于 Naive 算法, 这是因为使用位向量存储事务 *ID* 集合相比列表存储可以显著减少空间消耗, 与空间复杂度分析结果相同. 当 *minSup* 取 0.003 时, Naive 算法要比 ICFPM 多消耗约 400 MB 空间.

分析 *Eps* 的取值改变对算法空间开销的影响, 其他参数取值如下:*minSup*=0.003, *minPts*=10, *maxNR*=0.1, 算法在 *Eps* 变化时空间开销比较如图 8 所示.

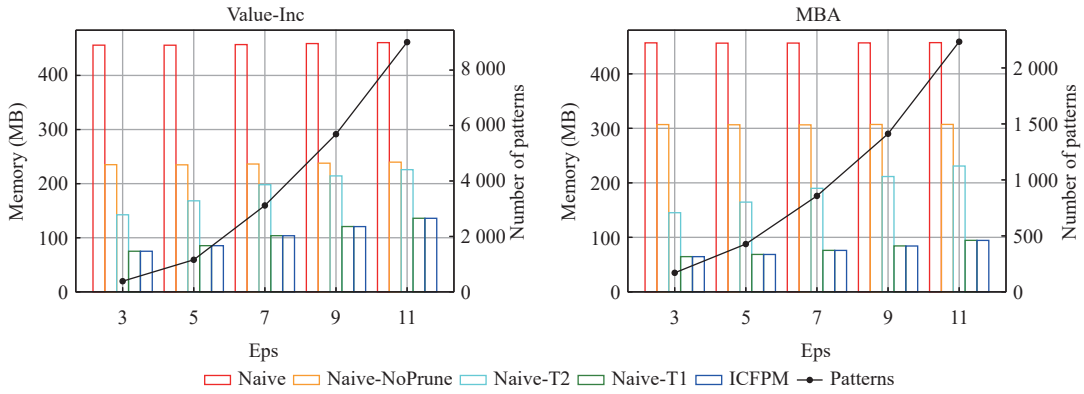

图 8　算法在 *Eps* 变化时空间开销比较

由图 8 可以看到, 在 2 个数据集上随着 *Eps* 增加, ICFPM 算法、Naive-T1 算法和 Naive-T2 算法空间开销和模式数量逐渐增加. 这是因为 *Eps* 越大, 更多时间戳可以聚为一簇, 产生更多候选项集, 增加了搜索空间和空间复杂度, 从而增加了算法空间开销和模式数量, 这与空间复杂度分析结果相同. 而 Naive 算法和 Naive-NoPrune 算法的空间开销不随着 *Eps* 增加而改变, 这是因为只通过频繁阈值剪枝, *Eps* 改变并没有改变算法搜索空间, 所以算法空间开销和空间复杂度也不会随着 *Eps* 增加而变化. 同时, 可以在 2 个数据集上观察到在相同 *Eps* 取值下, ICFPM 算法相对于其他 4 种算法空间开销总是最少, 并且 Naive-NoPrune 算法要好于 Naive 算法, 这与空间复杂度分析结果相同. 当 *Eps* 取 11 时, Naive 算法要比 ICFPM 多消耗约 400 MB 空间.

分析 *minPts* 的取值变化对算法空间开销的影响, 其他参数取值如下: *Eps*=5, *minSup*=0.003, *maxNR*=0.1, 算法

在 *minPts* 变化时空间开销比较如图 9 所示.

由图 9 可以看到, 在 2 个数据集上随着 *minPts* 增加, ICFPM 算法、Naive-T1 算法和 Naive-T2 算法空间开销和模式数量逐渐减少. 这是因为 *minPts* 越大, 簇的数量越少, 候选项集也越少, 减少了搜索空间和空间复杂度, 从而减少了算法空间开销和模式数量, 这与空间复杂度分析结果相同. 而 Naive 算法和 Naive-NoPrune 算法的空间开销不随着 *minPts* 增加而改变, 这是因为只通过频繁阈值剪枝, 并没有改变算法搜索空间, 所以算法空间开销和空间复杂度也不会随着 *minPts* 增加而变化. 同时, 可以在 2 个数据集上观察到在相同 *minPts* 取值下, ICFPM 算法相对于其他 4 种算法空间开销总是最少, 并且 Naive-NoPrune 算法要好于 Naive 算法, 这与空间复杂度分析结果相同. 当 *minPts* 取 3 时, Naive 算法要比 ICFPM 多消耗约 300 MB 空间.

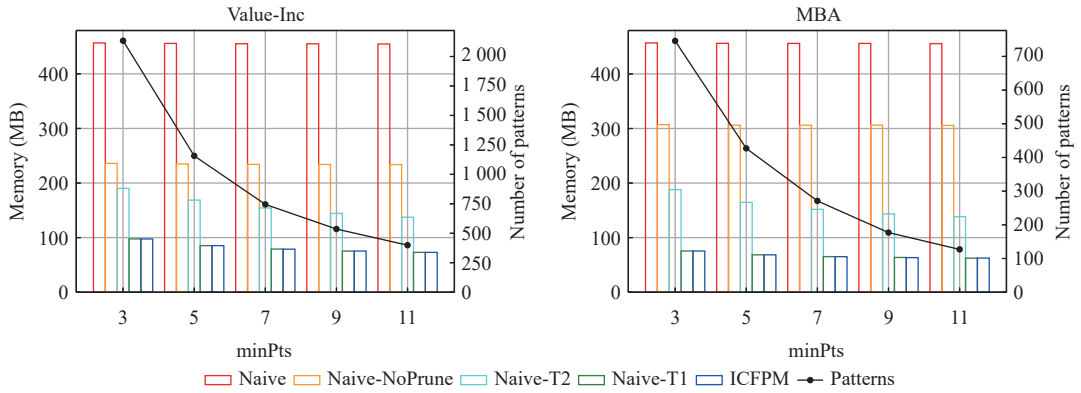

图 9 算法在 *minPts* 变化时空间开销比较

分析 *maxNR* 的取值变化对算法空间开销的影响. 其他参数取值如下: *Eps*=5, *minSup*=0.003, *minPts*=10, 算法在 *maxNR* 变化时空间开销比较如图 10 所示.

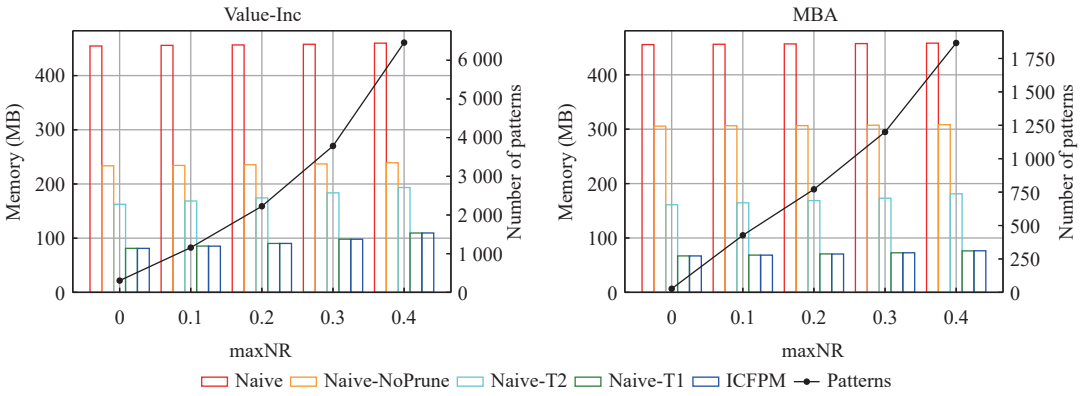

图 10 算法在 *maxNR* 变化时空间开销比较

由图 10 可以看到, 在 2 个数据集上随着 *maxNR* 增加, ICFPM 算法、Naive-T1 算法和 Naive-T2 算法空间开销和模式数量逐渐增加. 这是因为 *maxNR* 越大, 允许存在的噪点越多, 产生更多候选项集, 增加了搜索空间和空间复杂度, 从而增加了算法空间开销和模式数量, 这与空间复杂度分析结果相同. 而 Naive 算法和 Naive-NoPrune 算法的空间开销不随着 *maxNR* 增加而改变, 这是因为只通过频繁阈值剪枝, 并没有改变算法搜索空间, 所以算法空间开销和空间复杂度也不会随着 *minPts* 增加而变化. 同时, 可以在 2 个数据集上观察到在相同 *maxNR* 取值下, ICFPM 算法相对于其他 4 种算法空间开销总是最少, 并且 Naive-NoPrune 算法要好于 Naive 算法, 这与空间复杂度分析结果相同. 当 *maxNR* 取 0 时, Naive 算法要比 ICFPM 多消耗约 400 MB 空间.

5 总 结

本文首次针对时间有序事务数据中聚簇频繁模式的挖掘问题进行了研究. 首先给出了该问题的形式化描述, 接着提出了一种聚簇频繁模式挖掘算法 ICFPM. 该算法在 Naive 算法中引入了 2 种有效的优化策略以及高效的数据结构 *ICFPM-list*, 有效地减少了 Naive 算法的冗余计算. 实验结果表明, ICFPM 算法中使用的优化策略和数据结构都是有效的, 算法在时间和空间开销方面都要好于 Naive 方法, 可以有效解决时间有序事务数据中挖掘聚簇频繁模式的问题.

References:

[1] Agrawal R, Imieliński T, Swami A. Mining association rules between sets of items in large databases. In: Proc. of the 1993 ACM SIGMOD Int'l Conf. on Management of Data. Washington: ACM, 1993. 207–216. [doi: 10.1145/170035.170072]

[2] Agrawal R, Srikant R. Fast algorithms for mining association rules in large databases. In: Proc. of the 20th Int'l Conf. on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers Inc. , 1994. 487–499.

[3] Han JW, Pei J, Yin YW. Mining frequent patterns without candidate generation. ACM SIGMOD Record, 2000, 29(2): 1–12. [doi: 10.1145/335191.335372]

[4] Zaki MJ. Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering, 2000, 12(3): 372–390. [doi: 10.1109/69.846291]

[5] Guo YH, Tong YH, Tang SW, Yang DQ. Inverse frequent itemset mining based on FP-Tree. Journal of Software, 2008, 19(2): 338–350 (in Chinese with English abstract). [doi: 10.3724/SP.J.1001.2008.00338]

[6] Ding JM, Li HB, Deng B, Jia LY, You JG. Fast mining algorithm of frequent itemset based on spark. Journal of Software, 2023, 34(5): 2446–2464 (in Chinese with English abstract). [doi: 10.13328/j.cnki.jos.006404]

[7] Fournier-Viger P, Yang P, Kiran RU, Ventura S, Luna JM. Mining local periodic patterns in a discrete sequence. Information Sciences, 2021, 544: 519–548. [doi: 10.1016/j.ins.2020.09.044]

[8] Fournier-Viger P, Tseng VS. Mining top-K non-redundant association rules. In: 20th Int'l Symp. on Methodologies for Intelligent Systems. Macau, China: Springer, 2012. 31–40. [doi: 10.1007/978-3-642-34624-8_4]

[9] Agouti T. Graph-based modeling using association rule mining to detect influential users in social networks. Expert Systems with Applications, 2022, 202: 117436. [doi: 10.1016/j.eswa.2022.117436]

[10] Yao QY, Yang H, Bao BW, Yu A, Zhang J, Cheriet M. Core and spectrum allocation based on association rules mining in spectrally and spatially elastic optical networks. IEEE Transactions on Communications, 2021, 69(8): 5299–5311. [doi: 10.1109/TCOMM.2021.3082768]

[11] Agrawal R, Srikant R. Mining sequential patterns. In: Proc. of the Eleventh Int. Conf. on Data Engineering. Taipei, China: IEEE, 1995. 3–14. [doi: 10.1109/ICDE.1995.380415]

[12] Han J, Pei JW, Mortazavi-Asl B, Pinto H, Chen QM, Dayal U, Hsu MC. PrefixSpan, : Mining sequential patterns efficiently by prefix-projected pattern growth. In: Proc. 17th Int'l Conf. on Data Engineering. Heidelberg: IEEE, 2001. 215–224. [doi: 10.1109/ICDE.2001.914830]

[13] Huynh HM, Nguyen LTT, Pham NN, Oplatková ZK, Yun U, Vo B. An efficient method for mining sequential patterns with indices. Knowledge-Based Systems, 2022, 239: 107946. [doi: 10.1016/j.knosys.2021.107946]

[14] Zhang W, Yoshida T, Tang XJ, Wang Q. Text clustering using frequent itemsets. Knowledge-Based Systems, 2010, 23(5): 379–388. [doi: 10.1016/j.knosys.2010.01.011]

[15] Zhang LK, Yang GF. Cluster analysis of $PM_{2.5}$ pollution in China using the frequent itemset clustering approach. Environmental Research, 2022, 204: 112009. [doi: 10.1016/j.envres.2021.112009]

[16] Dong GZ, Li JY. Efficient mining of emerging patterns: Discovering trends and differences. In: Proc. of the 5th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. San Diego: ACM, 1999. 43–52. [doi: 10.1145/312129.312191]

[17] Zhang CS, Liu CC, Zhang XL, Almpanidis G. An up-to-date comparison of state-of-the-art classification algorithms. Expert Systems with Applications, 2017, 82: 128–150. [doi: 10.1016/j.eswa.2017.04.003]

[18] Yan YJ, Li ZJ, Chen HW. Efficiently mining of maximal frequent item sets based on FP-Tree. Journal of Software, 2005, 16(2): 215–222 (in Chinese with English abstract). [doi: 10.1360/jos160215]

[19] Li HF, Zhang N. A simple but effective stream maximal frequent itemset mining algorithm. In: 2011 7th Int'l Conf. on Computational Intelligence and Security. Sanya: IEEE, 2011. 1268–1272. [doi: 10.1109/CIS.2011.281]

[20] Wang SP, Wen YY, Zhao H. Mining full weighted maxinal frequent itensets based on sliding window over data stream. Journal of Northeastern University (Natural Science), 2016, 37(7): 931–936 (in Chinese with English abstract). [doi: 10.12068/j.issn.1005-3026.2016.07.005]

[21] Pasquier N, Bastide Y, Taouil R, Lakhal L. Discovering frequent closed itemsets for association rules. In: Proc. of the 7th Int'l Conf. on Database Theory. Jerusalem: Springer, 1999. 398–416. [doi: 10.1007/3-540-49257-7_25]

[22] Pei J, Han J, Mao R. CLOSET: An efficient algorithm for mining frequent closed itemsets. In: ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery. New York: ACM, 2000. 21–30. [doi: 10.1145/360402.360431]

[23] Liu JQ, Ye ZS, Yang XC, Wang XL, Shen LJ, Jiang XN. Efficient strategies for incremental mining of frequent closed itemsets over data streams. Expert Systems with Applications, 2022, 191: 116220. [doi: 10.1016/j.eswa.2021.116220]

[24] Ahmed CF, Tanbeer SK, Jeong BS, Lee YK. Efficient tree structures for high utility pattern mining in incremental databases. IEEE Transactions on Knowledge and Data Engineering, 2009, 21(12): 1708–1721. [doi: 10.1109/TKDE.2009.46]

[25] Zida S, Fournier-Viger P, Lin JCW, Wu CW, Tseng VS. EFIM: A fast and memory efficient algorithm for high-utility itemset mining. Knowledge and Information Systems, 2017, 51(2): 595–625. [doi: 10.1007/s10115-016-0986-0]

[26] Duong QH, Fournier-Viger P, Ramampiaro H, Nørvåg K, Dam TL. Efficient high utility itemset mining using buffered utility-lists. Applied Intelligence, 2018, 48(7): 1859–1877. [doi: 10.1007/s10489-017-1057-2]

[27] Tanbeer SK, Ahmed CF, Jeong BS, Lee YK. Discovering periodic-frequent patterns in transactional databases. In: 13th Pacific-Asia Conf. on Knowledge Discovery and Data Mining. Bangkok: Springer, 2009. 242–253. [doi: 10.1007/978-3-642-01307-2_24]

[28] Kiran RU, Venkatesh JN, Fournier-Viger P, Toyoda M, Reddy PK, Kitsuregawa M. Discovering periodic patterns in non-uniform temporal databases. In: 21st Pacific-Asia Conf. on Knowledge Discovery and Data Mining. Jeju: Springer, 2017. 604–617. [doi: 10.1007/978-3-319-57529-2_47]

[29] Venkatesh J, Kiran RU, Krishna RP, Kitsuregawa M. Discovering periodic-correlated patterns in temporal databases. In: Hameurlain A, Wagner R, Hartmann S, Ma H, eds. Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXVIII: Special Issue on Database-and Expert-Systems Applications. Berlin: Springer, 2018. 146–172. [doi: 10.1007/978-3-662-58384-5_6]

[30] Rage UK, Alampally A, Chennupati S, Toyoda M, Reddy PK, Kitsuregawa M, Reddy M. Finding periodic-frequent patterns in temporal databases using periodic summaries. Data Science and Pattern Recognition, 2019, 3(2): 24–46.

[31] Krzywicki A, Mahidadia A, Bain M. Discovering periodicity in locally repeating patterns. In: 2022 IEEE 9th Int'l Conf. on Data Science and Advanced Analytics. Shenzhen: IEEE, 2022. 1–10. [doi: 10.1109/DSAA54385.2022.10032435]

[32] Kiran RU, Veena P, Ravikumar P, Saideep C, Zettsu K, Shang HC, Toyoda M, Kitsuregawa M, Reddy PK. Efficient discovery of partial periodic patterns in large temporal databases. Electronics, 2022, 11(10): 1523. [doi: 10.3390/electronics11101523]

[33] Upadhya KJ, Paleja A, Geetha M, Rao BD, Chhabra MS. Finding partial periodic and rare periodic patterns in temporal databases. IEEE Access, 2023, 11: 92242–92257. [doi: 10.1109/ACCESS.2023.3308820]

[34] Tanbeer SK, Ahmed CF, Jeong BS, Lee YK. Mining regular patterns in transactional databases. IEICE Transactions on Information and Systems, 2008, E91.D(11): 2568–2577. [doi: 10.1093/ietisy/e91-d.11.2568]

[35] Rashid MM, Karim MR, Jeong BS, Choi HJ. Efficient mining regularly frequent patterns in transactional databases. In: 17th Int'l Conf. on Database Systems for Advanced Applications. Busan: Springer, 2012. 258–271. [doi: 10.1007/978-3-642-29038-1_20]

[36] Amphawan K, Lenca P, Surarerks A. Mining top-k regular-frequent itemsets using database partitioning and support estimation. Expert Systems with Applications, 2012, 39(2): 1924–1936. [doi: 10.1016/j.eswa.2011.08.055]

[37] Kumar GV, Kumari VV. MaRFI: Maximal regular frequent itemset mining using a pair of transaction-ids. International Journal of Computer Science & Engineering Technology, 2013, 4(7): 2229–3345.

[38] Amphawan K, Lenca P. Mining top-k frequent-regular closed patterns. Expert Systems with Applications, 2015, 42(21): 7882–7894. [doi: 10.1016/j.eswa.2015.06.021]

[39] Rehman SU, Khan MA, Nabi HU, Ali S, Alnazzawi N, Khan S. TKIFRPM: A novel approach for topmost-k identical frequent regular patterns mining from incremental datasets. Applied Sciences, 2023, 13(1): 654. [doi: 10.3390/app13010654]

[40] Chen GS, Li ZS. Discovering periodic cluster patterns in event sequence databases. Applied Intelligence, 2022, 52(13): 15387–15404. [doi: 10.1007/s10489-022-03186-z]

[41] Ester M, Kriegel HP, Sander J, Xu XW. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proc. of the Second Int'l Conf. on Knowledge Discovery and Data Mining. Portland: AAAI Press, 1996. 226–231.

附中文参考文献:

[5] 郭宇红, 童云海, 唐世渭, 杨冬青. 基于 FP-Tree 的反向频繁项集挖掘. 软件学报, 2008, 19(2): 338–350. [doi: 10.3724/SP.J.1001.2008.00338]

[6] 丁家满, 李海滨, 邓斌, 贾连印, 游进国. 一种基于 Spark 的频繁项集快速挖掘算法. 软件学报, 2023, 34(5): 2446–2464. [doi: 10.13328/j.cnki.jos.006404]

[18] 颜跃进, 李舟军, 陈火旺. 基于 FP-Tree 有效挖掘最大频繁项集. 软件学报, 2005, 16(2): 215–222. [doi: 10.1360/jos160215]

[20] 王少鹏, 闻英友, 赵宏. 滑动窗口下数据流完全加权最大频繁项集挖掘. 东北大学学报 (自然科学版), 2016, 37(7): 931–936. [doi: 10.12068/j.issn.1005-3026.2016.07.005]

王少鹏(1984－), 男, 博士, 副教授, 主要研究方向为大数据挖掘与分析, 时空大数据处理, AI 和 DB 的融合.

牛超煜(1999－), 男, 硕士生, 主要研究领域为数据挖掘.