基于向量的数据流滑动窗口中最大频繁项集挖掘 *

徐嘉莉[1], 陈 佳[2], 胡 庆[1], 黄 波[1], 郭红霞[1]
(1. 成都大学 电子信息工程学院, 成都 610106; 2. 电子科技大学 计算机科学与工程学院, 成都 610054)

摘 要: 针对相关算法在挖掘数据流最大频繁项集时所存在的问题,提出了一种基于向量的数据流滑动窗口中最大频繁项集挖掘算法。该算法首先用向量作为概要数据结构,采用定量更新滑动窗口策略解决时间粒度问题;其次通过位运算产生频繁项集,利用矩阵和数组存储辅助信息,深度优先搜索产生最大频繁项集时利用剪枝策略进一步减少挖掘时间;最后用索引链表存储挖掘结果以提高超集检测效率。理论分析和实验结果验证了该算法的有效性。

关键词: 数据流; 最大频繁项集; 滑动窗口; 向量
中图分类号: TP311 **文献标志码**: A **文章编号**: 1001-3695(2012)03-0837-04
doi:10.3969/j.issn.1001-3695.2012.03.009

Algorithm based on vector for mining maximal frequent itemsets in sliding window over data streams

XU Jia-li[1], CHEN Jia[2], HU Qing[1], HUANG Bo[1], GUO Hong-xia[1]
(1. *School of Electronic & Information Engineering, Chengdu University, Chengdu* 610106, *China*; 2. *School of Computer Science & Engineering, University of Electronic Science & Technology of China, Chengdu* 610054, *China*)

Abstract: This paper proposed an algorithm based on vector for mining maximal frequent itemsets in sliding window over data streams (MFISW) aimed at the mining problems of maximal frequent itemsets over data streams. Firstly, the algorithm used vector to express items in data streams and solved the problem of time granularity through quantitative updating strategies. Secondly, it stored the ancillary information using a matrice and a array in creating the frequent sets through the bit operation, and improved the mining efficiency again using pruning technology during creating the maximal frequent sets. Finally, it improved the detecting efficiency by using a index list to store mining results. Theoretical analysis and experimental results show the algorithm is efficient.

Key words: data stream; maximal frequent itemsets; sliding window; vector

0 引言

数据流是一个随时间连续不断到来的数据序列,其连续性、高速性、无限性、时变性和不可再现性等特点[1]使得数据流挖掘算法应该满足如下条件[2]:单次线性扫描;流算法是在线算法,应具有低的时间复杂度;流算法是主存算法,应具有低的空间复杂度;能适应动态变化的数据与流速。

在数据流中,频繁项集挖掘算法按挖掘方式分为启发式挖掘和批处理挖掘[3]。启发式挖掘能够随流数据的到达直接进行处理,在一定程度上能实时反映频繁项集的变化,但对于高速到达的数据流,其处理速度仍然有限,且时间粒度较细;批处理挖掘的处理速度较快,但需要积攒足够数据,无法满足实时性要求,且时间粒度较粗。

按照挖掘结果的不同,频繁项集挖掘算法又分为频繁闭项集挖掘、完全频繁项集挖掘和最大频繁项集挖掘。由于最大频繁项集数目相对较少且已隐含所有的频繁项集,其挖掘算法具有较好的时间、空间效率。因此,数据流中最大频繁项集的挖掘受到业内更多关注,并取得了一定的成果,主要包括 estDec + [4]、DSM-MFI[5]和 INSTANT[6]等算法。上述算法能在数据流流过的同时进行一次扫描,但仍然存在明显的不足。estDec + 算法使用的监测树是一种松散的数据结构,不能有效利用模式树的存储空间、空间复杂度大,在处理一个长度为 l 的事务时需要产生并测试 2^l-1 个中间候选模式,时间复杂度大;DSM-MFI 算法利用概述频繁项集森林结构存储所有的事务数据信息,随着频繁项集数目的增大,维护该结构所需的时间和空间开销也急剧增大。另外,estDec + 、DSM-MFI 和 INSTANT 均属于启发式挖掘,具有较好的实时性,但对于高速到达的数据流,其处理速度仍然有限,时间粒度较细。

在实际应用中,人们更多关注近期数据,出现了基于滑动窗口的数据流挖掘算法,如典型的 FP-stream 算法[7],它使用 FP-stream 数据结构,能在有限的存储空间内挖掘数据流中时间敏感的频繁模式,时间效率较好;但由于它采用批处理方式,对流数据的处理及用户对当前数据查询响应的实时性较差,时间粒度较粗;其次,该算法不仅需要维护数据流全局的频繁模

收稿日期: 2011-08-28; **修回日期**: 2011-10-08 **基金项目**: 国家"863"计划资助项目(2007AA01Z443);成都大学校基金资助项目(2010XJZ16)

作者简介:徐嘉莉(1969-),女,四川成都人,副教授,硕士,主要研究方向为数据挖掘、网格计算(lotussunnyx@163.com);陈佳(1980-),女,四川南充人,副教授,博士,主要研究方向为数据挖掘、网格计算;胡庆(1977-),女,四川梁平人,讲师,硕士,主要研究方向为物联网技术;黄波(1974-),男,成都人,讲师,硕士,主要研究方向为电子信息;郭红霞(1976-),女,新疆克拉玛依人,讲师,硕士,主要研究方向为电子信息.

式树,还需要额外的时间和空间维护各个数据分段的频繁模式树,而且需要两次访问数据流元素才能捕获数据流上的频繁模式。

基于上述分析,本文提出了一种基于向量的数据流滑动窗口中最大频繁项集挖掘算法(MFISW)。

1　相关定义与性质

$I=\{I_1,I_2,\cdots,I_m\}$ 是数据流上项的集合,数据流 $DS=\{T_1,\cdots,T_{i-1},T_i,T_{i+1},\cdots\}$ 为一个大小可能无限的事务数据序列,其中 T_i 表示在第 i 时刻到达的事务,对于任意 T_i,都有 $T_i\subseteq I$。

定义 1　全序关系 $<$。根据字典中的字母顺序,若 X 小于 Y,则记为 $X<Y$,如 $A<B<C$。同理,定义两个项集的字典顺序为 $<$,如 $A<ABC<CD$。

假定文中项集按照全序关系排列。

定义 2　将 DS 分成若干个基本窗口,记为 W,其包含的事务个数 $|W|$ 就是基本窗口的大小,一般取 $|W|=1$。滑动窗口 SW 由一系列连续的基本窗口组成,记为 $(W_1,W_2,\cdots,W_n)$,每个 SW 包含的基本窗口的个数就是该 SW 的大小,记为 $|SW|$。

定义 3　给定 SW, $\forall X\subseteq I$,则 X 在 SW 中的支持度为 $\sup(X)=N_x/|SW|$,其中 N_x 为 SW 中包含 X 的事务数。

定义 4　给定 SW 和最小支持度 min_sup, $\forall X\subseteq I$,若 $\sup(X)\geqslant$min_sup,则称 X 为 SW 中的频繁项集。

定义 5　给定 SW 和 min_sup, $\forall X\subseteq I$,若 $\sup(X)\geqslant$min_sup 且 $\forall(x\subset Y\wedge Y\subseteq I)$,均有 $\sup(Y)<$min_sup,则称 X 为 SW 中的最大频繁项集。

性质 1　$\forall X\subseteq I,Y\subseteq I$,且 $X\supset Y,Y\neq\varnothing$,若 X 是频繁项集,则 Y 一定也是频繁项集。

性质 2[8]　$\forall X\subseteq I,Y\subseteq I$,且 $X\subset Y$,若 X 不是频繁项集,则 Y 也不是频繁项集。

推论　$\forall I_{i1}<I_{i2}<I_{i3}<\cdots<I_{ik}<I_{iu}(k\geqslant 2,1\leqslant i_k\leqslant m)$,则 $\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu}\}$ 是频繁 $(k+1)$-项集的必要条件是:

a) $\{I_{i1},I_{iu}\},\{I_{i2},I_{iu}\},\cdots,\{I_{ik},I_{iu}\}$ 都是频繁 2-项集;

b) 至少存在 k 个比 I_{iu} 编号小且能与 I_{iu} 组成频繁 2-项集的项。

证明　因为 $\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu}\}$ 是频繁 $(k+1)$-项集 $(k\geqslant 2)$,由性质 1 得 $\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu}\}$ 的子集 $\{I_{i1},I_{iu}\},\{I_{i2},I_{iu}\},\cdots,\{I_{ik},I_{iu}\}$ 都是频繁 2-项集;又因为 $\{I_{i1},I_{iu}\},\{I_{i2},I_{iu}\},\cdots,\{I_{ik},I_{iu}\}$ 一共有 k 个且都包含 I_{iu},而且 $I_{i1}<I_{i2}<I_{i3}<\cdots<I_{ik}<I_{iu}$,所以比 I_{iu} 编号小且能与 I_{iu} 组成频繁 2-项集的项有 k 个。除 $I_{i1},I_{i2},\cdots,I_{ik},I_{iu}$ 之外,可能还存在比 I_{iu} 编号小且能与 I_{iu} 组成频繁 2-项集的项,所以至少存在 k 个比 I_{iu} 编号小且能与 I_{iu} 组成频繁 2-项集的项。

2　MFISW 算法

2.1　窗口初始化

MFISW 算法为当前 SW 中事务包含的项 $I_j(j=1,2,\cdots,m)$ 建立对应的向量 V_SW_j, V_SW_j 的长度就是 $|SW|$。将流过窗口的事务看成队列,向量 V_SW_j 也看成队列。如果 I_j 在 SW 中第 i 个事务中出现,则 V_SW_j 的第 i 位置 1,否则置 0。V_SW_j 中"1"的数目就是包含项 I_j 的事务数。$\{I_{i1},I_{i2},\cdots,I_{ik}\}$ 的支持数就是 $V_SW_{i1}\wedge V_SW_{i2}\wedge\cdots\wedge V_SW_{ik}$ 中"1"的个数,记为 sup_num$(I_{i1},I_{i2},\cdots,I_{ik})$, $(I_{i1},I_{i2},\cdots,I_{ik})$ 的支持度为 $\sup(I_{i1},I_{i2},\cdots,I_{ik})=$ sup_num$(I_{i1},I_{i2},\cdots,I_{ik})/|SW|$。

当窗口内事务数小于 $|SW|$ 时,就处于窗口初始化阶段[9]。在此阶段,新事务不断移入窗口,而没有事务移出;随着新事务的不断移入,窗口中项对应的向量也在不断更新。为了减少内存消耗,当 $|V_SW_j|(j=1,2,\cdots,m)$ 为 0 时,即当前窗口不存在项 I_j,就删除 V_SW_j。核心算法如下:

算法 1　init_slidewindow

输入: $DS=\{T_1\cdots,T_{i-1},T_i,T_{i+1},\cdots\}$, $I=\{I_1,I_2,\cdots,I_m\}$。
输出: $V_SW_j(j=1,2,\cdots,m)$。

```
init_ slidewindow (SW)
{if (SW is not full) then
  {while each new coming Ti in SW do
    for(j=1;j<=m;j++)
      if (Ij in Ti) then en_queue(V_SWj,1);
      else en_queue(V_SWj,0);
    for(j=1;j<=m;j++)
    if (|V_SWj|==0) then delete V_SWj;
  }}
```

2.2　窗口滑动

初始化阶段结束后进入窗口滑动阶段,开始定量更新滑动窗口。

已知 $|SW|$,临时窗口 TW 的长度为 $|TW|$, SW 更新的长度为 $|TW|$,一般 $|TW|$ 大于启发式方法的更新长度而远小于批处理方法的更新长度。可以根据 DS 流速改变 $|TW|$,当流速比较快时, $|TW|$ 可以稍大些;当流速比较缓慢时, $|TW|$ 可以稍小些。这样就能灵活适应 DS 中流量的多变性,解决批处理方法和启发式方法中时间粒度较粗或较细的问题。

将新到的事务放入 TW,当 TW 满时,则更新 SW,即把 SW 中最先到来的 $|TW|$ 个事务从中丢弃,将 TW 中的 $|TW|$ 个事务放入 SW 末尾。同样,最早的 $|TW|$ 个事务所包含的项从 $V_SW_j(j=1,2,\cdots,m)$ 中移除,新到的 $|TW|$ 个事务所包含的项被插入 V_SW_j 尾部。为了减少内存消耗,当 $|V_SW_j|(j=1,2,\cdots,m)$ 为 0 时,就删除 V_SW_j。

算法 2　upda_slidewindow

输入: $DS=\{T_1,\cdots,T_{i-1},T_i,T_{i+1},\cdots\}$, $I=\{I_1,I_2,\cdots,I_m\}$, $V_SW_j(j=1,2,\cdots,m)$, $|TW|$。
输出: $V_SW_j(j=1,2,\cdots,m)$。

```
upda_slidewindow(SW)
{if (TW is not full) then init_ slidewindow (TW);
else for(j=1;j<=m;j++)
  {for(p=1;p<=|TW|;p++)
    {dl_queue(V_SWj);
    en_queue(V_SWj,dl_queue(V_TWj));}
    if (|V_SWj|==0) then delete V_SWj;
  }}
```

2.3　最大频繁项集挖掘

当滑动窗口更新完毕或用户发出查询请求时,进入最大频繁项集挖掘阶段。

2.3.1　索引链表

Relim 算法挖掘频繁项集时采用事务链表作为存储结构,获得了比树型结构更高的空间利用率[10]。但是,该事务链表中的项头表采用数组作为存储结构,虽然可以随机存取,但查找元素的时间复杂度为 $O(n)$,查询效率较低。而 hash 表能根据元素关键字的值,通过哈希函数计算得出该元素所存储的位置,查找元素的时间复杂度为 $O(1)$。因此,本节基于 Relim 算法的事务链表,引入 hash 技术,建立索引链表以存放最大频繁项集,记为 MFIs,其组成如下:

a)用 hash 表来存储项头表中的项,表中的每个元素定义为(item,list),其中 item $\in I$,且为频繁 1-项集,在 hash 表中的位置由 hash 函数 f(item)确定;list 指向以 item 为头元素的项集链表。

b)以 item 为头元素的项集链表组成一个最大频繁项集,链表中每个节点定义为(data,next),其中 data $\in I$,next 指向下一个项。

2.3.2　最大频繁项集的挖掘

1)生成矩阵 A 和数组 B

由推论知道,编号最大的项不可能再与比它编号大的项组成频繁项集,因此构造一个上三角零矩阵 $A_{(m-1)\times m}$(m 为项的个数);定义一个长度为 m 的一维整型数组 B,用 $B_{[i]}$ 存放比 I_i 编号小且能与 I_i 构成频繁 2-项集的项的个数, $B_{[i]}$ 初始化为 0。若 $\sup(I_i,I_j)\geqslant$min_sup,则 $A_{[i][j]}=1$,$B_{[j]}=B_{[j]}+1$。其算法核心如下:

算法 3　生成矩阵 A 和数组 B

```
输入:V_SW_i(i=1,…,m),min_sup。
输出:滑动窗口的矩阵 A 和数组 B。
{for i=1 to m-1 do
   {for j=i+1 to m do A[i][j]=0;
   B[i]=0;}
B[m]=0;
for i=1 to m-1 do
   for j=i+1 to m do
   if (sup(I_i)≥min_sup and sup(I_j)≥min_sup and sup(I_i,I_j)≥
min_sup) then { A[i][j]=1; B[j]=B[j]+1;}
}
```

2)产生最大频繁项集

a)产生潜在的最大频繁项集。由算法 3 知道,如果 $B_{[j]}\geqslant 1$,且 $A_{[i,j]}=1$,则$\{I_i,I_j\}$就是频繁 2-项集。根据推论,利用频繁 k-项集的最后一项对该项集进行深度优先扩展,产生潜在的最大频繁项集。

设$\{I_{i1},I_{i2},\cdots,I_{ik}\}$是频繁 k-项集,如果 $A_{[ik,iu]}=1(I_{ik}<I_{iu})$,$B_{[iu]}\geqslant k$,并且 $A_{[i1,iu]}=A_{[i2,iu]}=\cdots=A_{[ik-1,iu]}=1$,那么$\{I_{i1},I_{i2},\cdots,I_{ik}\}$被扩展为$(k+1)$-项集$\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu}\}$;如果 $\sup(I_{i1},I_{i2},\cdots,I_{ik},I_{iu})\geqslant$min_sup,则$\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu}\}$就是频繁$(k+1)$-项集。然后再对$\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu}\}$的最后一项 I_{iu} 进行扩展,直到不能扩展为止。假定$\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu},\cdots,I_{ix}\}$就是不能再扩展的频繁项集,那么它就是潜在的最大频繁项集。在深度优先扩展过程中,利用 $B_{[j]}$ 中存放的是比 I_j 的编号小且能与 I_j 构成频繁 2-项集的项的个数这一特点,实现对$\{I_{i1},I_{i2},\cdots,I_{ik}\}$扩展到$\{I_{i1},I_{i2},\cdots,I_{ik},I_j\}(I_{ik}<I_j)$的剪枝。因为如果 $B_{[j]}<k$,就不用再去检验$\{I_{i1},I_j\}$,…,$\{I_{ik-1},I_j\}$是否是频繁 2-项集,这样在一定程度上提高了算法执行的效率。

b)产生最大频繁项集。检查 MFIs 中是否存在$\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu},\cdots,I_{ix}\}$的超集,若无,则$\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu},\cdots,I_{ix}\}$就是最大频繁项集,将其添加到 MFIs 中。

超集检测的性能很大程度上依赖于最大频繁项集集合的大小,如果该集合较大,采用顺序查找法非常耗时。由于本文用索引链表 MFIs 存储最大频繁项集,可以用索引法检测超集。将$\{I_{i1},I_{i2},\cdots,I_{ik},I_{iu},\cdots,I_{ix}\}$的第一项作为一级索引 index_1,第二项作为二级索引 index_2,则需要检测的子表空间由 hash 表中 index_1$\leqslant I_{i1}$的空间与链表中 index_2$\leqslant I_{i2}$的空间组成,除此空间之外的最大频繁项集不可能是待检测的最大频繁项集的超集,因此能在一定程度上提高检测效率。产生最大频繁项集的算法核心如下:

算法 4　最大频繁 k-项集挖掘算法

```
输入:滑动窗口中矩阵 A、B。
输出:滑动窗口中的最大频繁项集 MFIs。
for( j=2 to m do)
{if (B[j]≥1) then
for i=1 to j-1 do
{ if (A[i][j]=1) then
   {L0=L={I_i,I_j}就是频繁 2-项集;
   k=2;
   if (j<m) then
   {for r=j+1 to m do
      {L=L0; dfs(A,B,j,r,k,L);
      if L0≠L then
      {用索引法检测 MFIs 中是否有 L 的超集;
      if MFIs 中没有 L 的超集 then
      add L to MFIs;
      }}
         if L0==L then
         {用索引法检测 MFIs 中是否有 L0 的超集;
         if MFIs 中没有 L0 的超集 then
         add L0 to MFIs;}}}
}}

dfs(A,B,i_k,p,k,VAR L)
//L={I_i1,I_i2,…,I_ik}是频繁 k-项集
{if (A[ik,p]=1)and (B[p]≥k) and (A[i1,p]=A[i2,p]=A[i3,p]=…=
A[ik-1,p]=1) and (sup(I_i1,I_i2,…,I_ik,I_p)≥min_sup)
then { L=L∪{I_p}; i_k=p; k++;
   if(p<m) then
   for q=p+1 to m do dfs(A,B,i_k,q,k,L);}}
```

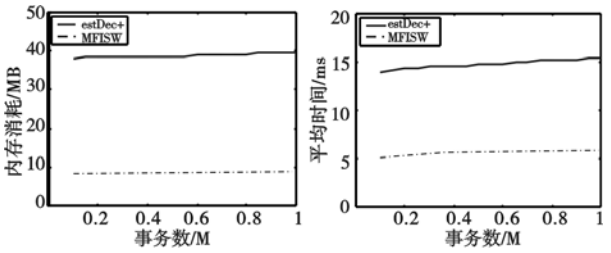
3　算法实现及分析

本文采用 IBM Synthetic data generator[11] 生成的模拟数据,在 Windows XP(Intel Pentium processor 1.73 GHz、内存为 1 GB)平台上、SQL Server 2000 和 Delphi 7.0 的环境下进行仿真。

图 1 是当 DS 为 T10I4D1000K、$|SW|$为 0.2M($|TW|$为 0.1M)时,estDec+和MFISW 算法的性能比较。如图 1(a)所示, estDec+的内存空间消耗远高于 MFISW,随着已处理的流数据的增多,两种算法的内存空间消耗基本保持不变。这是因为 MFISW 只需要存储 DS 上项对应的向量、上三角矩阵、数组和 MFIs,向量、上三角矩阵和数组所需的空间大小主要由项的个数 m 来决定,MFIs 仅存放当前窗口的最大频繁项集且最大频繁项集数目极少。而 estDec+采用的 CP-tree 是一种较松散的存储结构,需要更多的空间来存储模式信息,因此 estDec+的内存空间消耗远高于 MFISW。另外,在同一个数据流下,MFISW 的向量、数组和矩阵所需空间在 DS 各个窗口上基本保持不变,而 MFIs 仅存放当前窗口的最大频繁项集,因此 MFISW 的内存空间消耗在 DS 各个窗口上变化不大。同样地,由于 estDec+周期性地删除了模式树上的不频繁模式与过期模式,使模式树只维护 DS 上最近的重要模式信息,因此,estDec+的内存空间消耗在 DS 各个窗口上变化也不大。

如图 1(b)所示,estDec+的平均处理时间远高于 MFISW,随着已处理流数据的增多,MFISW 与 estDec+的平均处理时间都相对稳定。这是因为 estDec+采用 Apriori-like 方法处理 DS 上的每一个事务,首先产生事务所包含的全部子模式,然后通过搜索模式树分别测试它们的频繁性,并将其重要模式信息更新到模式树上。而 MFISW 摆脱了 Apriori 框架的束缚,只需扫描一次数据流、不产生庞大的候选项集;在挖掘过程中利用

剪枝策略删除无用节点;采用索引方法缩小超集检测的空间。因此 MFISW 的平均处理时间远小于 estDec + 的平均处理时间。另外,由于 MFISW 在 *DS* 各个 *SW* 上的空间复杂度变化不大,因此,该算法在 *DS* 各个 *SW* 上的平均处理时间变化也不大。同样地,由于 estDec + 的模式树在 *DS* 各个 *SW* 上的空间复杂度变化不大,因此,estDec + 在 *DS* 各个 *SW* 上的平均处理时间变化也不大。



(a) 内存消耗比较　　(b) 处理时间比较

图1　MFISW和estDec+的性能比较

图 2 是当 *DS* 分别为 T10I5D200K、T10I10D200K、T15I10D200K、T15I15D200K 和 T20I20D200K 且 |*SW*| 为 0.2M (|*TW*| 为 0.1M) 时,MFISW 与 eatDec + 处理长事务长模式数据流的能力比较。为了避免各算法的平均处理时间受数据流大小的影响,将数据流大小固定为 200 K。如图 2 所示,随着平均事务长度 *T* 与平均模式长度 *I* 的增加,两种算法的平均处理时间均有所增加,但 estDec + 的增幅比 MFISW 大。这是因为当事务的平均长度增大时,estDec + 需要产生并测试更多的中间候选模式,且当事务的平均模式长度增加时,将有更多的模式信息需要更新到模式树上。因此,当平均事务长度特别是平均模式长度增加时,estDec + 的平均处理时间将大大增加,其扩展性较差。而 MFISW 的时间开销主要取决于深度优先搜索(DFS)的执行次数,DFS 的执行次数是由矩阵元素的个数以及频繁项集的长度决定,即使平均事务长度特别是平均模式长度增加,矩阵元素个数的改变以及频繁项集长度的改变也相对较小,因而 MFISW 平均处理时间的增幅不大,即 MFISW 受平均事务长度及平均模式长度的影响较小,其扩展性较好。

图 3 是当 *DS* 为 T10I4D1000K 时,MFISW、eatDec + 与 FP-stream 面对不同流速的可扩展性比较。如图 3 所示,当 *DS* 流速较小时,滑动窗口 |*SW*| 比较小,MFISW 的 |*TW*| 也比较小,estDec + 的平均处理时间最大,而 MFISW 与 FP-stream 相差不大。但是随着流速增大,|*SW*|(|*TW*|) 增大时,三种方法的平均处理时间均在增加,其中 FP-stream 的增速最快,MFISW 与 estDec + 增速相对缓慢。这是因为 estDec + 属于启发式挖掘,需要为每一个事务产生大量子模式并测试它们的频繁性,而 FP-stream 属于批处理挖掘、MFISW 属于定量更新,因此,当挖掘相对较小的流数据集合时,estDec + 的效率最低,而 MFISW 与 FP-stream 相差不大。当 |*SW*|(|*TW*|) 增大时,三种方法的平均处理时间均有所增加。其中,estDec + 虽然属于启发式挖掘,但它周期性地删除模式树上的不频繁模式及过期模式,降低了模式树的维护代价,因此其算法的平均处理时间增幅相对缓慢;MFISW 只需维护向量、矩阵、数组和 MFIs,并且采用了剪枝策略。另外,根据流速动态调整 |*TW*|,在很大程度上降低了时间消耗,因而其算法的平均处理时间呈缓慢上升趋势;而 FP-stream 虽然属于批处理挖掘,但需要多次移动并合并模式树上节点的倾斜时间窗口,故 FP-stream 平均处理时间的增幅相对要大得多。

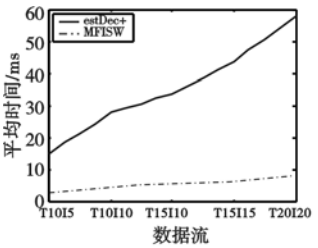


图2　MFISW和estDec+的可扩展性比较

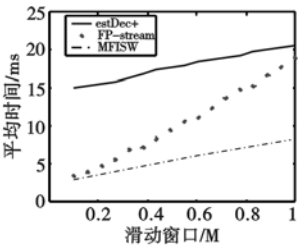


图3　MFISW、estDec+和FP-steram的可扩展性比较

4　结束语

MFISW 算法只需扫描一次数据流,不产生候选项集,用矩阵和数组存放辅助信息减少了空间消耗;挖掘过程中利用剪枝策略进一步减少了时空消耗,用索引法检测超集提高了检测效率,采用定量更新滑动窗口策略解决了批处理方法和启发式方法中时间粒度较粗或较细的问题。与其他同类算法[3~6]相比,MFISW 的效率更高、可扩展性更好。由于 MFISW 是对当前滑动窗口中的数据项进行增量更新,难免存在部分频繁项集重复挖掘的现象,这在一定程度上影响了算法的效率。能否对当前滑动窗口的最大频繁项集进行增量更新以进一步提高算法的效率,这将是下一步要研究的问题。

参考文献:

[1] BABCOCK B, BABU S, DATAR M, *et al*. Models and issues in data stream systems[C]//Proc of the 21st ACM SIGMOD-SIGART Symposium on Principles of Database System. New York: ACM Press, 2002: 1-16.

[2] GAROFALAKIS M, GEHRKE J. Querying and mining data streams: you only get one look a tutorial[C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2002: 635.

[3] 曹国栋. 数据流频繁模式挖掘算法研究与设计[D]. 秦皇岛: 燕山大学, 2007.

[4] LEE D, LEE W. Finding maximal frequent itemsets over online data streams adaptively[C]//Proc of the 5th IEEE International Conference on Data Mining. Washington DC: IEEE Computer Society, 2005: 266-273.

[5] LI Hua-fu, LEE S, SHAN M. Online mining maximal frequent itemsets over data streams[C]//Proc of the 15th International Workshops on Research Issues in Data Engineering: Stream Data Mining and Applications. 2005: 11-18.

[6] MAO Guo-jun, WU Xin-dong, ZHU Xing-quan, *et al*. Mining maximal frequent itemsets from data streams[J]. Journal of Information Science, 2007, 33(3): 251-262.

[7] GIANNELLA C, HAN Jia-wei, PEI Jian, *et al*. Mining frequent patterns in data streams at multiple time granularities[M]//Next Generation Data Mining. Cambridge: MIT Press, 2005: 191-212.

[8] 牛小飞, 石冰, 卢军, 等. 挖掘关联规则的高效 ABM 算法[J]. 计算机工程, 2004, 30(11): 118-120.

[9] 李可, 任家东. 窗口模式下在线数据流中频繁项集的挖掘[J]. 计算机应用研究, 2010, 27(5): 1711-1713.

[10] BORGELT C. Keeping things simple: finding frequent itemsets by recursive elimination[C]//Proc of the 1st International Workshop on Open Source Data Mining. New York: ACM Press, 2005: 66-70.

[11] AGRAWAL R, SRIKANT R. Fast algorithms for mining association rules[C]//Proc of the 20th International Conference on Very Large Databases. San Francisco: Morgan Kaufmann Publishers, 1994: 487-499.