上次课回顾

- 强关联规则：Strong Association Rule，满足最小支持度和最小信任度的关联规则称为强关联规则。

- 关联规则挖掘问题可以划分成两个子问题：
  - 1. 发现频繁项目集:通过用户给定Minsupport，寻找所有频繁项目集或者最大频繁项目集。
  - 2、生成关联规则:通过用户给定Minconfidence，在频繁项目集中，寻找关联规则。

关联规则挖掘基本过程

- 关联规则挖掘的目标：强关联规则，满足最小支持度和最小信任度的关联规则，Strong Association Rule。

- 关联规则挖掘问题可以划分成两个子问题：
  - 1、发现所有的频繁项集

Apriori 算法是通过项目集元素数目的不断增长来逐步完成频繁项目集发现的。首先产生 1-频繁项目集 L_1，然后是 2-频繁项目集 L_2，直到不能再扩展频繁项目集的元素数目而算法停止。在第 k 次循环中，过程先产生 k-候选项目集的集合 C_k，然后通过扫描数据库生成支持度并测试产生 k-频繁项目集 L_k。

  - 2、从频繁项集中发现强关联规则

在频繁项目集中，寻找强关联规则。

- 开始数据库里有4条交易，min_confidence=80%作为可信度阈值，生成强关联规则

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

Itemset	sup
{B, C, E}	2

最大频繁项集（支持度）	子集（支持度）	可信度	规则	是否强规则
B、C、E（50%）	B、C（50%）	100%	BC→E	是
B、C、E（50%）	B、E（75%）	67%	BE→C	否
B、C、E（50%）	C、E（50%）	100%	CE→B	是
B、C、E（50%）	B（75%）	67%	B→CE	否
B、C、E（50%）	C（75%）	67%	C→BE	否
B、C、E（50%）	E（75%）	67%	E→BD	否

关联规则的生成问题

- 在得到了所有频繁项目集后，可以按照下面的步骤生成关联规则（书76面）：
  - 对于每一个频繁项目集*l*，生成其所有的非空子集；
  - 对于*l* 的每一个非空子集x，计算Conference（x），如果Confidence（x）≥minconfidence，那么“x⇨（*l*-x）”成立。

- 算法3-4 从给定的频繁项目集中生成强关联规则

Rule-generate（L，minconf）
（1） FOR each frequent itemset l_k in L
（2） *genrules*（l_k，l_k）；

这个算法的核心是genrules递归过程，它实现一个频繁项目集中所有强关联规则的生成。

令最小支持度50%，最小置信度80%

Database TDB

Tid	Items
1	A, C, D
2	B, C, E
3	A, B, C, E
4	B, E

1st scan

C_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

C_2

2nd scan

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

C_3

Itemset
{B, C, E}
{A,B,C}
{A,B,E}
{A,C,E}

3rd scan

L_3

Itemset	sup
{B, C, E}	2

在频繁项目集中，寻找强关联规则。

- min_confidence=80%

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

频繁项集（支持度）	子集（支持度）	可信度	规则	是否强规则
A、C	C		C→A	
	A		A→C	
B、C	C		C→B	
	B		B→C	
B、E	E		E→B	
	B		B→E	
C、E	E		E→B	
	C		C→E	

关联规则生产算法的优化

- 定义3-3 设项目集X，X_1是X的一个子集，如果规则 X⇨（l — X）不是强规则，那么X_1 ⇨（l — X_1）成立一定不是强规则。

  - 例如 AB⇨CD 不是强规则
    A ⇨* 不是强规则
    B ⇨* 不是强规则

在频繁项目集中，寻找强关联规则。

- 开始数据库里有4条交易，min_confidence=80%作为可信度阈值，生成强关联规则

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

Itemset	sup
{B, C, E}	2

最大频繁项集（支持度）	子集（支持度）	可信度	规则	是否强规则
B、C、E（50%）	B、C（50%）	100%	BC→E	是
B、C、E（50%）	B、E（75%）	67%	BE→C	否
B、C、E（50%）	C、E（50%）	100%	CE→B	是
B、C、E（50%）	B（75%）	67%	B→CE	否
B、C、E（50%）	C（75%）	67%	C→BE	否
B、C、E（50%）	E（75%）	67%	E→BD	否

在频繁项目集中，寻找强关联规则。

- min_confidence=80%

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

频繁项集（支持度）	子集（支持度）	可信度	规则	是否强规则
A、C	C		C→A	
	A		A→C	
B、C	C		C→B	否
	B		B→C	否
B、E	E		E→B	
	B		B→E	
C、E	E		E→B	
	C		C→E	

例子 3-2　对表 3-1，Apriori 算法生成的最大频繁项目集为{ABCD，BCE}（不失一般性，这里仅讨论最大频繁项目集），下面跟踪 Rule-generate 的执行过程（设 minconfidence＝60%）。表 3-2 给出了生成过程。

表 3-2　关联规则生成过程示意

序号	l_k	x_{m-1}	confidence	support	规则（是否是强规则）
1	ABCD	ABC	67%	40%	ABC⇨D（是）
2	ABCD	AB	67%	40%	AB⇨CD（是）
3	ABCD	A	67%	40%	A⇨BCD（是）
4	ABCD	B	40%	40%	B⇨ACD（否）
5	ABCD	AC	67%	40%	AC⇨BD（是）
6	ABCD	C	50%	40%	C⇨ABD（否）
7	ABCD	BC	50%	40%	BC⇨AD（否）
8	ABCD	ABD	100%	40%	ABD⇨C（是）
9	ABCD	AD	100%	40%	AD⇨BC（是）
10	ABCD	D	67%	40%	D⇨ABC（是）
11	ABCD	BD	67%	40%	BD⇨AC（是）
12	ABCD	ACD	100%	40%	ACD⇨B（是）
13	ABCD	CD	100%	40%	CD⇨AB（是）
14	ABCD	BCD	100%	40%	BCD⇨A（是）
15	BCE	BC	50%	40%	BC⇨E（否）
16	BCE	B	40%	40%	B⇨CE（否）
17	BCE	C	50%	40%	C⇨BE（否）
18	BCE	BE	67%	40%	BE⇨C（是）
19	BCE	E	67%	40%	E⇨BC（是）
20	BCE	CE	100%	40%	CE⇨B（是）

关联规则生产算法的优化

- 定义3-4 设项目集X，X_1是X的一个子集，如果规则 Y⇨X是强规则，那么Y ⇨X_1一定是强规则。

  - 例如 AB⇨CD 是强规则
    AB ⇨C 是强规则
    AB ⇨D 是强规则

只要从所有最大频繁项目集出发，去测试可能的关联规则即可，因为其他频繁项目集生产的规则的后件，一定包含在对应的最大频繁项目集生产的关联规则后见中。

Apriori算法的性能瓶颈

- Apriori作为经典的频繁项目集生成算法，在数据挖掘中具有里程碑的作用。

- Apriori算法有两个致命的性能瓶颈：
  - 1. 多次扫描事务数据库，需要很大的I/O负载
    - 对每次k循环，侯选集C_k中的每个元素都必须通过扫描数据库一次来验证其是否加入L_k。假如有一个最大频繁项目集包含10个项的话，那么就至少需要扫描事务数据库10遍。
  - 2. 可能产生庞大的侯选集
    - 由L_{k-1}产生k-侯选集C_k是指数增长的，例如10^4个1-频繁项目集就有可能产生接近10^7个元素的2-侯选集。如此大的侯选集对时间和主存空间都是一种挑战。

第三章 关联规则挖掘理论和算法

内容提要

- 基本概念
- 经典的频繁项目集生成算法分析： Apriori算法
- Apriori算法的性能瓶颈问题
- Apriori的改进算法
- 对项目集格空间理论的发展
- 基于项目序列集操作的关联规则挖掘算法
- 改善关联规则挖掘质量问题
- 约束数据挖掘问题
- 关联规则挖掘中的一些更深入的问题
- 数量关联规则挖掘方法

提高Apriori算法效率的技术

- 一些算法虽然仍然遵循Apriori 属性，但是由于引入了相关技术，在一定程度上改善了Apriori算法适应性和效率。
- 主要的改进方法有：
  - 基于数据分割（Partition）的方法：在一个划分中的支持度小于最小支持度的k-项集不可能是全局频繁的
  - 基于散列（Hash）的方法： 在一个hash桶内支持度小于最小支持度的k-项集不可能是全局频繁的
  - 基于采样（Sampling）的方法：通过采样技术，评估被采样的子集中，并依次来估计k-项集的全局频度
  - 其他：动态删除没有用的事务，不包含任何L_k的事务对未来的扫描结果不会产生影响，因而可以删除

基于数据分割的方法

- **定理3-5** 设数据集D被分割成分块D_1，D_2，…，D_n，全局最小支持数为minsup_count。如果一个数据分块D_i的局部最小支持数$minsup_count_i$（i=1，2，…，n），按着如下方法生成：

  $minsup\_count_i = minsup\_count * ||D_i|| / ||D||$ 则所有的局部频繁项目集涵盖全局频繁项目集。

- **作用：**
  - **1．合理利用主存空间：**数据分割将大数据集分成小的块，为块内数据一次性导入主存提供机会。
  - **2．支持并行挖掘算法：**每个分块的局部频繁项目集是独立生成的，因此提供了开发并行数据挖掘算法的良好机制。

提高Apriori算法效率的技术

- 一些算法虽然仍然遵循Apriori 属性，但是由于引入了相关技术，在一定程度上改善了Apriori算法适应性和效率。
- 主要的改进方法有：
  - 基于数据分割（Partition）的方法：在一个划分中的支持度小于最小支持度的k-项集不可能是全局频繁的
  - 基于散列（Hash）的方法： 在一个hash桶内支持度小于最小支持度的k-项集不可能是全局频繁的
  - 基于采样（Sampling）的方法：通过采样技术，评估被采样的子集中，并依次来估计k-项集的全局频度
  - 其他：动态删除没有用的事务，不包含任何L_k的事务对未来的扫描结果不会产生影响，因而可以删除

基于散列的方法

- 1995，Park等发现寻找频繁项目集的主要计算是在生成2-频繁项目集上。因此，Park等利用了这个性质引入杂凑技术来改进产生2-频繁项目集的方法。
- 例子：桶地址 =（10x+y）mod 7；minsupport_count=3

TID	Items
1	I1，I2，I5
2	I2，I4
3	I2，I3
4	I1，I2，I4
5	I1，I3
6	I2，I3
7	I1，I3
8	I1，I2，I3，I5
9	I1，I2，I3

桶地址	0	1	2	3	4	5	6
桶计数	2	2	4	2	2	4	4
桶内	{I1，I4}	{I1，I5}	{I2，I3}	{I2，I4}	{I2，I5}	{I1，I2}	{I1，I3}
	{I3，I5}	{I1，I5}	{I2，I3}	{I2，I4}	{I2，I5}	{I1，I2}	{I1，I3}
			{I2，I3}			{I1，I2}	{I1，I3}
			{I2，I3}			{I1，I2}	{I1，I3}

L2={{I2，I3}，{I1，I2}，{I1，I3}}

提高Apriori算法效率的技术

- 一些算法虽然仍然遵循Apriori 属性，但是由于引入了相关技术，在一定程度上改善了Apriori算法适应性和效率。
- 主要的改进方法有：
  - 基于数据分割（Partition）的方法：在一个划分中的支持度小于最小支持度的k-项集不可能是全局频繁的
  - 基于散列（Hash）的方法： 在一个hash桶内支持度小于最小支持度的k-项集不可能是全局频繁的
  - 基于采样（Sampling）的方法：通过采样技术，评估被采样的子集中，并依次来估计k-项集的全局频度
  - 其他：动态删除没有用的事务，不包含任何Lk的事务对未来的扫描结果不会产生影响，因而可以删除

第三章 关联规则挖掘理论和算法

内容提要

- 基本概念与解决方法
- 经典的频繁项目集生成算法分析
- Apriori算法的性能瓶颈问题
- Apriori的改进算法
- 对项目集格空间理论的发展: close算法、fp-tree算法
- 基于项目序列集操作的关联规则挖掘算法
- 改善关联规则挖掘质量问题
- 约束数据挖掘问题
- 关联规则挖掘中的一些更深入的问题
- 数量关联规则挖掘方法

闭合项目集挖掘——close算法

- 闭合项目集：一个项目集C，当且仅当对于在C中的任何元素，不可能在C中，存在小于或等于它的支持度的子集。

TID	Item
1	a, b, c
2	a, b, c, d
3	b, c, e
4	a, c, d, e
5	d, c

C={ab，abc}是不是闭合的？

count{ab}=2

count{abc}=2

C={ab，abc}不是闭合项集

C2={bc，abc}是不是闭合的？

count{bc}=3，count{abc}=2

所以， C2={bc，abc}是闭合项集

闭合项目集挖掘——close算法

- 闭合频繁项集：closed frequent Itemset，如果闭项集同时是频繁的，也就是它的支持度大于等于最小支持度阈值，那它就称为闭频繁项集。

交易号（TID）	商品（Items）
1	beer, diaper, nuts
2	beer, biscuit, diaper
3	bread, butter, cheese
4	beer, cheese, diaper, nuts
5	beer, butter, cheese, nuts

令minsup=60%，那么项集{diaper，{diaper，beer}}是闭合频繁项集吗？

闭合项目集挖掘——close算法

- 闭合频繁项集：closed frequent Itemset

交易号（TID）	商品（Items）
1	beer, diaper, nuts
2	beer, biscuit, diaper
3	bread, butter, cheese
4	beer, cheese, diaper, nuts
5	beer, butter, cheese, nuts

support{diaper，{diaper，beer}}=60%,是频繁项集，不是闭合频繁项集。

support{diaper，beer}=60%，是闭合频繁项集，因为不存在一个与它具有相同支持度的超集

闭合项目集挖掘——close算法

- 闭合项目集：一个项目集C，当且仅当对于在C中的任何元素，不可能在C中，存在小于或等于它的支持度的子集。

TID	Item
1	a, b, c
2	a, b, c, d
3	b, c, e
4	a, c, d, e
5	d, c

C1={ab 2，abc 2}不是闭合项集

C2={bc 3，abc 2}是闭合项集

请找出d的闭合项集?

{abcd}∩{acde}∩{dc}={cd}

Count{cd}= 3

闭合项目集挖掘——close算法

- 一个频繁闭合项目集的所有闭合子集一定是频繁的；一个非频繁闭合项目集的所有闭合超集一定是非频繁的。

Close 算法是对 Apriori 的改进算法。在 Close 算法中，也使用了迭代的方法：利用频繁闭合 i-项目集记为 FC_i，生成频繁闭合$(i+1)$-项目集，记为 $FC_{i+1}(i\geqslant 1)$。首先找出候选 1-项目集，记为 FCC_1，通过扫描数据库找到它的闭合以及支持度，得到候选闭合项目集。然后对其中的每个候选闭合项进行修剪，如果支持度不小于最小支持度，则加入到频繁闭合项目集 FC_1 中。再将它与自身连接，以得到候选频繁闭合 2-项目集 FCC_2，再经过修剪得出 FC_2，再用 FC_2 推出 FC_3，如此继续下去，直到有某个值 r 使得候选频繁闭合 r-项目集 FCC_r 为空，这时算法结束。

在 Close 算法中调用了三个关键函数：Gen_Closure(FCC_i)，Gen_Generator(FC_i)和 Deriving frequent itemsets，为了正确理解 Close，需要对它们进行描述。

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

1-项集：{A}、{B}、{C}、{D}、{E}

2-项集：{AB}、{AC}、{AD}、{AE}、{BC}、{BD}、{BE}、{CD}、{CE}、{DE}

3-项集：……

4-项集：……

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{A}		
{B}		
{C}		
{D}		
{E}		

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{A}		
{B}		
{C}		
{D}		
{E}	{A B E}	2

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{A}		
{B}		
{C}		
{D}	{B D}	2
{E}	{A B E}	2

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{A}		
{B}		
{C}	{C}	6
{D}	{B D}	2
{E}	{A B E}	2

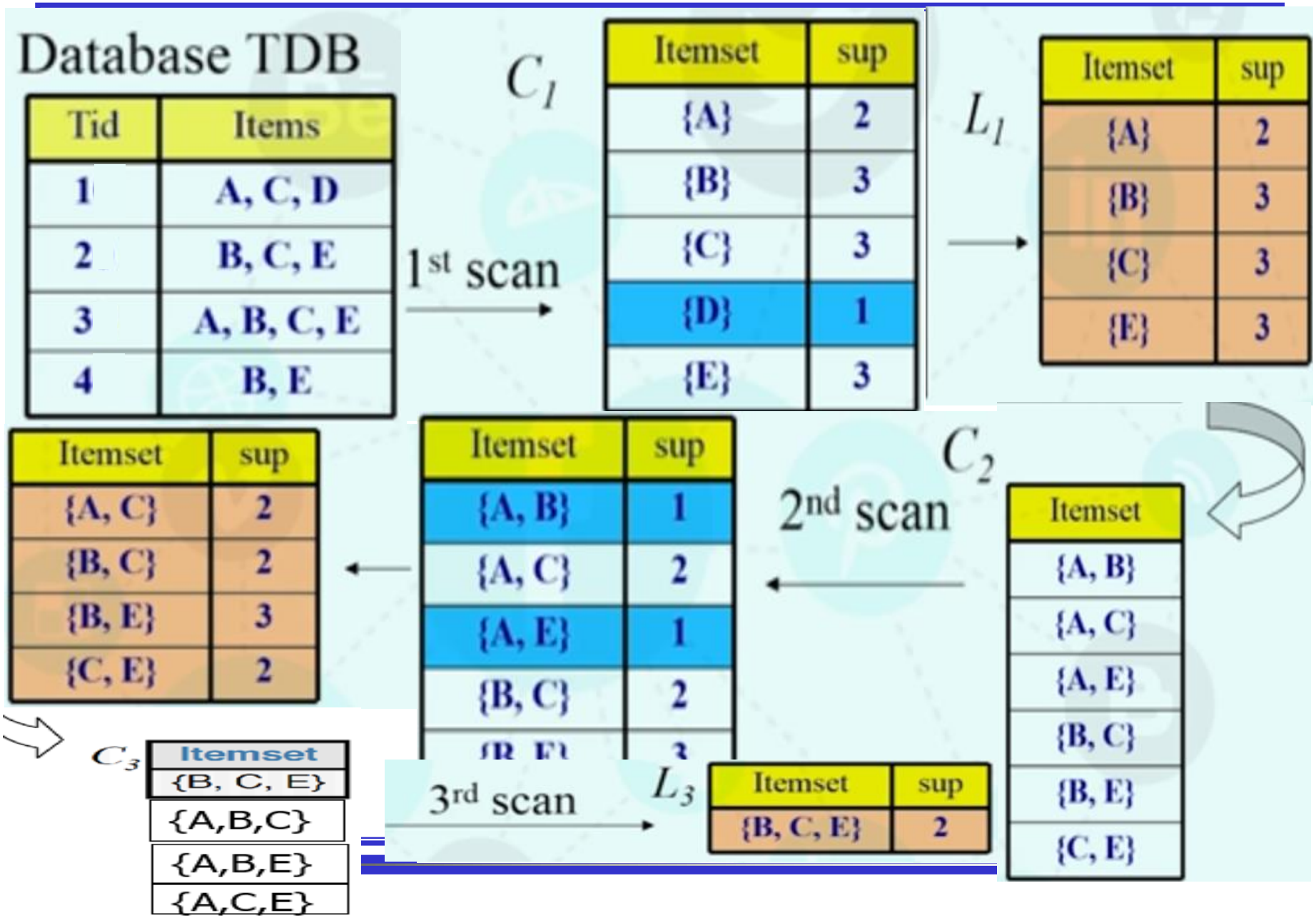
Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{A}	{A}	6
{B}	{B}	7
{C}	{C}	6
{D}	{B D}	2
{E}	{A B E}	2

与apriori算法做比较



Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

- 一元组：{A}、{B}、{C}、~~{D}~~、~~{E}~~

Generator	Closure	Support
{A}	{A}	6
{B}	{B}	7
{C}	{C}	6
{D}	{B D}	2
{E}	{A B E}	2

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

- 一元组：{A}、{B}、{C}、~~{D}~~、~~{E}~~

Generator	Closure	Support
{A}	{A}	6
{B}	{B}	7
{C}	{C}	6
{D}	{B D}	2
{E}	{A B E}	2

- 二元组：{AB}、{AC}、{AD}、{AE}、{BC}、{BD}、{BE}、{CE}、{DE}

  修剪操作

  闭合子集：{AE}、{BE}

  闭合空集：{CD}、{DE}（在表3-1中没有出现过）

二元组：{AB}、{AC}、{AD}、~~{AE}~~、{BC}、{BD}、~~{BE}~~、~~{CD}~~、{CE}、~~{DE}~~

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

二元组：{AB}、{AC}、{AD}、{AE}、{BC}、{BD}、{BE}、{CD}、{CE}、{DE}

Generator	Closure	Support
{AB}		
{AC}		
{AD}		
{BC}		
{BD}		
{CE}		

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{AB}		
{AC}		
{AD}		
{BC}		
{BD}		
{CE}	{ABCE}	1

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{AB}		
{AC}		
{AD}		
{BC}		
{BD}	{BD}	2
{CE}	{ABCE}	1

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{AB}	{AB}	4
{AC}	{AC}	4
{AD}	{ABD}	1
{BC}	{B C}	4
{BD}	{BD}	2
{CE}	{ABCE}	1

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{AB}	{AB}	4
{AC}	{AC}	4
{AD}	{ABD}	1
{BC}	{B C}	4
{BD}	{BD}	2
{CE}	{ABCE}	1

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

TID	Itemset
1	A B E
2	B D
3	B C
4	A B D
5	A C
6	B C
7	A C
8	A B C E
9	A B C

Generator	Closure	Support
{AB}	{A B}	4
{AC}	{A C}	4
{BC}	{B C}	4
{BD}	{B D}	2

Generator	Closure	Support
{ABC}	{A B C}	2

Close算法的例子

- 下面是Close算法作用到表中数据集的执行过程（假如minsup_count=2）：

Generator	Closure	Support
{A}	{A}	6
{B}	{B}	7
{C}	{C}	6
{D}	{B D}	2
{E}	{A B E}	2

所有不重复的闭合加入到FC：{A、B、C、BD、ABE、AB、AC、BC、ABC}

Generator	Closure	Support
{AB}	{A B}	4
{BC}	{A C}	4
{BC}	{B C}	4
{BD}	{B D}	2

Generator	Closure	Support
{ABC}	{A B C}	2

Close算法

算法 3-6 Close 算法

(1) generators in FCC_1 = {1-itemsets}; //候选频繁闭合 1-项目集

(2) FOR(i=1; FCC_i.generators=Φ; i++) DO BEGIN

(3) closures in FCC_i=Φ;

(4) supports in FCC_i=0;

(5) FCC_i=Gen_Closure(FCC_i); //计算 FCC 的闭合

(6) FOR all candidate closed itemsets c∈FCC_i DO BEGIN

(7) IF(c.support>=minsupport) THEN FC_i=FC_i∪{c}; //修剪小于最小支持度的项

(8) END

(9) FCC_{i+1}=Gen_Generator(FC_i); //生成 FCC_{i+1}

(10) END

(11) FC=$\bigcup_i FC_i$(FC_i.closure,FC_i.support); //返回 FC

(12) Deriving frequent itemsets(FC,L);

Close算法

算法 3-7 Gen_Closure 函数

```
(1) FOR all transactions t∈D DO BEGIN
(2)     Go=Subset(FCC_i. generator, t);
(3)     FOR all generators p∈Go DO BEGIN
(4)       IF(p. closure=Φ) THEN p. closure=t;
(5)       ELSE p. closure=p. closure∩t;
(6)       p. support++;
(7)     END
(8) END
(9) Answer=∪{c∈FCC_i | c. closure≠Φ};
```

Close算法

算法 3-8 Gen_Generator 函数

```
(1) FOR all generators p∈FCC_{i+1} DO BEGIN
(2)    Sp=Subset(FC_i.generator,p);          //取得 p 的所有 i-项子集
(3)    FOR all s∈Sp DO BEGIN
(4)       IF(p∈s.closure) THEN               //如果 p 是它的 i 项子集闭合的子集
(5)          delete p from FCC_{i+1}.generator;   //将它删除
(6)    END
(7) END
(8) Answer=⋃{c∈FCC_{i+1}};
```

Close算法

算法 3-9 函数 Deriving frequent itemsets(FC,L)

```
(1) k=0;
(2) FOR all frequent closed itemsets c∈FC DO BEGIN
(3)     L‖c‖ = L‖c‖ ∪{c};                    //按项的个数归类
(4)     IF(k<‖c‖) THEN k=‖c‖;                //记下项目集包含的最多的个数
(5) END
(6) FOR(i=k; i>1; i--) DO BEGIN
(7)   FOR all itemsets c∈Li DO
(8)     FOR all(i-1)-subsets s of c DO       //分解所有(i-1)-项目集
(9)       IF(s!∈Li-1) THEN BEGIN             //不包含在 Li-1中
(10)        s.support=c.support;             //支持度不变
(11)        Li-1=Li-1∪{s};                   //添加到 Li-1中
(12)      END
(13) L=∪Li;                                  //返回所有的 Li
```

Close算法

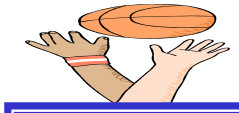
在 Apriori 算法中，C_k 中的每个元素需在事务数据库中进行验证（计算支持度）来决定其是否加入 L_k，这里的验证过程是算法性能的一个瓶颈。这个方法要求多次扫描可能很大的事务数据库，即如果频繁项目集最多包含 c 个项，那么就需要扫描事务数据库 $c+1$ 遍，这需要很大的 I/O 负载。虽然之后的算法进行了优化，在自身的连接过程采用了修剪技术，减少了 C_k 的大小，改进了生成频繁项目集的性能，但是，如果数据库很大，算法的效率还是很低的。因此采用闭合的方法，可以在一定程度上减少数据库扫描的次数。

使用频繁闭合项目集发现频繁集，可以提高发现关联规则的效率。由于实际上既是频繁的又是闭合的项目集的比例比频繁项目集的比例要小得多。而且随着数据库事务数的增加和项的大小的增加，项目集格增长很快。使用闭合项目集格可以通过减少查找空间、减少数据库扫描次数，来改进 Apriori 方法。

第三章 关联规则挖掘理论和算法

内容提要

- 基本概念与解决方法
- 经典的频繁项目集生成算法分析
- Apriori算法的性能瓶颈问题
- Apriori的改进算法
- 对项目集格空间理论的发展: close、fp-tree
- 基于项目序列集操作的关联规则挖掘算法
- 改善关联规则挖掘质量问题
- 约束数据挖掘问题
- 关联规则挖掘中的一些更深入的问题
- 数量关联规则挖掘方法



FP-tree算法的基本原理

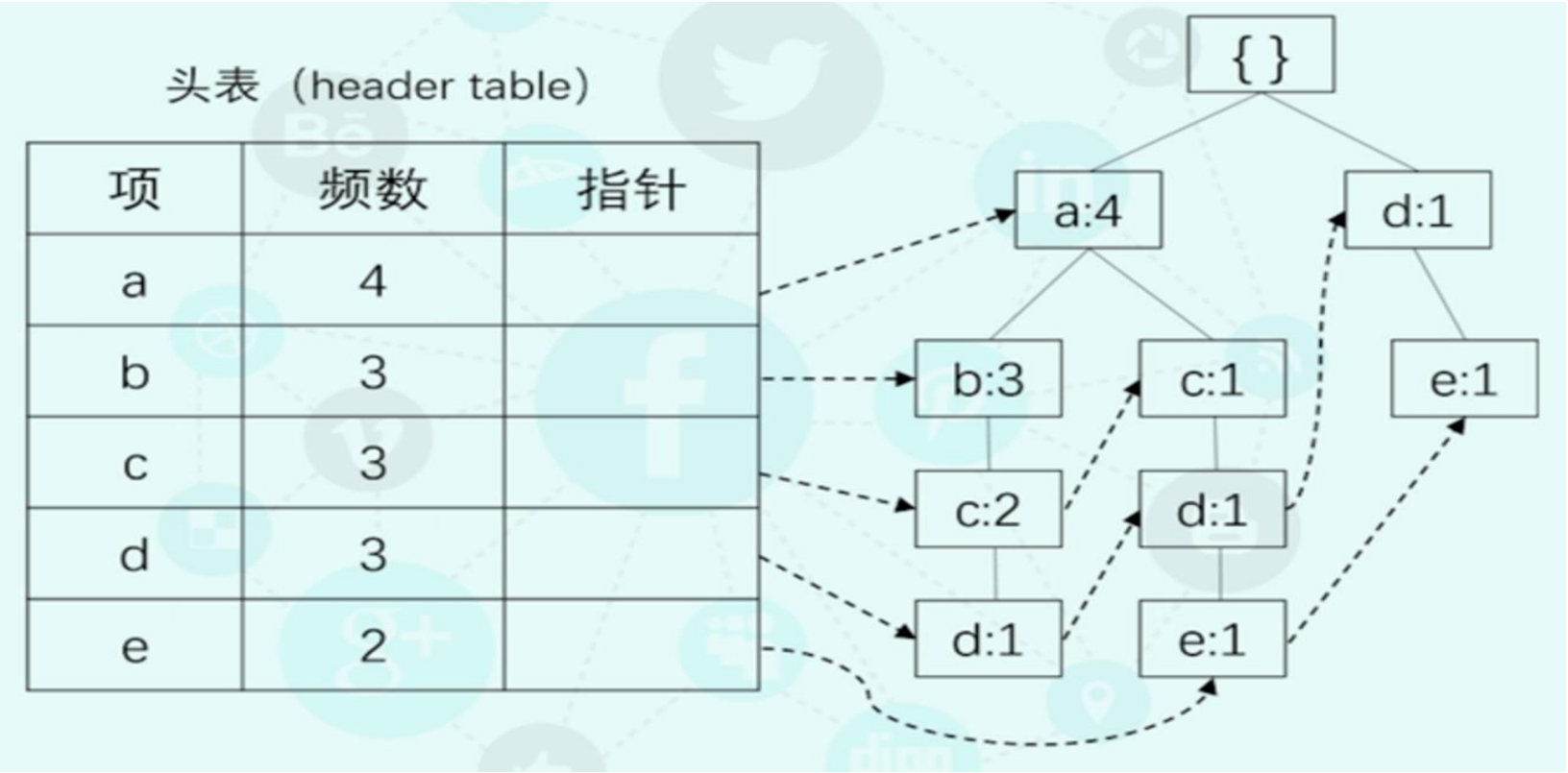
- Frequent Pattern Tree：不使用侯选集，直接压缩数据库成一个频繁模式树，通过频繁模式树可以直接得到频集。进行2次数据库扫描：一次对所有1-项目的频度排序；一次将数据库信息转变成紧缩内存结构。

交易号（TID）	商品（Items）
1	beer, diaper, nuts
2	beer, biscuit, diaper
3	bread, butter, cheese
4	beer, cheese, diaper, nuts
5	beer, butter, cheese, nuts

交易号（TID）	商品（Items）
1	a, b, c
2	a, b
3	d, e
4	a, b, c, d
5	a, c, d, e

FP-tree算法的基本原理

- Frequent Pattern Tree：不使用侯选集，直接压缩数据库成一个频繁模式树，通过频繁模式树可以直接得到频集。进行2次数据库扫描：一次对所有1-项目的频度排序；一次将数据库信息转变成紧缩内存结构。



- Frequent Pattern Tree：不使用侯选集，直接压缩数据库成一个频繁模式树，通过频繁模式树可以直接得到频集。进行2次数据库扫描：一次对所有1-项目的频度排序；一次将数据库信息转变成紧缩内存结构。

3.5.2 FP-tree 算法

2000 年，Han 等提出了一个称为 FP-tree 的算法。这个算法只进行 2 次数据库扫描。它不使用候选集，直接压缩数据库成一个频繁模式树，最后通过这棵树生成关联规则。

事实上，FP-tree 的算法由两个主要步骤完成：第一步是利用事务数据库中的数据构造 FP-tree；第二步是从 FP-tree 中挖掘频繁模式。

FP-tree算法实例-统计频率

ID	Items
1	牛奶，鸡蛋，面包，薯片
2	鸡蛋，爆米花，薯片，啤酒
3	牛奶，面包，啤酒
4	牛奶，鸡蛋，面包，爆米花，薯片，啤酒
5	鸡蛋，面包，薯片
6	鸡蛋，面包，啤酒
7	牛奶，面包，薯片
8	牛奶，鸡蛋，面包，黄油，薯片
9	牛奶，鸡蛋，黄油，薯片

- Step1:先扫描数据库，统计所有商品的出现次数(频数)，然后按照频数递减排序，删除频数小于最小支持度的商品。

设最小支持度数为：minsup=4

统计频数：

- 牛奶6，鸡蛋7，面包7，薯片7，爆米花2，啤酒4，黄油2.

降序排序：

- 薯片7,鸡蛋7,面包7,牛奶6,啤酒4（删除小于minsup的商品）

频繁1项集，记为F1

FP-tree算法实例-重新排序

ID	Items
1	牛奶，鸡蛋，面包，薯片
2	鸡蛋，爆米花，薯片，啤酒
3	牛奶，面包，啤酒
4	牛奶，鸡蛋，面包，爆米花，薯片，啤酒
5	鸡蛋，面包，薯片
6	鸡蛋，面包，啤酒
7	牛奶，面包，薯片
8	牛奶，鸡蛋，面包，黄油，薯片
9	牛奶，鸡蛋，黄油，薯片

Step2:对每一条数据记录，按照F1重新排序。

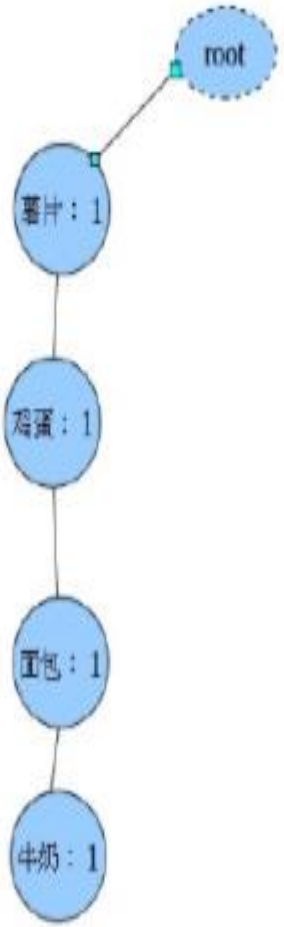
ID	Items
1	薯片,鸡蛋,面包,牛奶
2	薯片,鸡蛋,啤酒
3	面包,牛奶,啤酒
4	薯片,鸡蛋,面包,牛奶,啤酒
5	薯片,鸡蛋,面包
6	鸡蛋,面包,啤酒
7	薯片,面包,牛奶
8	薯片,鸡蛋,面包,牛奶
9	薯片,鸡蛋,牛奶

FP-tree算法实例-建立FP树

ID	Items
1	薯片, 鸡蛋, 面包, 牛奶
2	薯片, 鸡蛋, 啤酒
3	面包, 牛奶, 啤酒
4	薯片, 鸡蛋, 面包, 牛奶, 啤酒
5	薯片, 鸡蛋, 面包
6	鸡蛋, 面包, 啤酒
7	薯片, 面包, 牛奶
8	薯片, 鸡蛋, 面包, 牛奶
9	薯片, 鸡蛋, 牛奶

Step3：把第二步重新排序后的记录，插入到fp-tree中

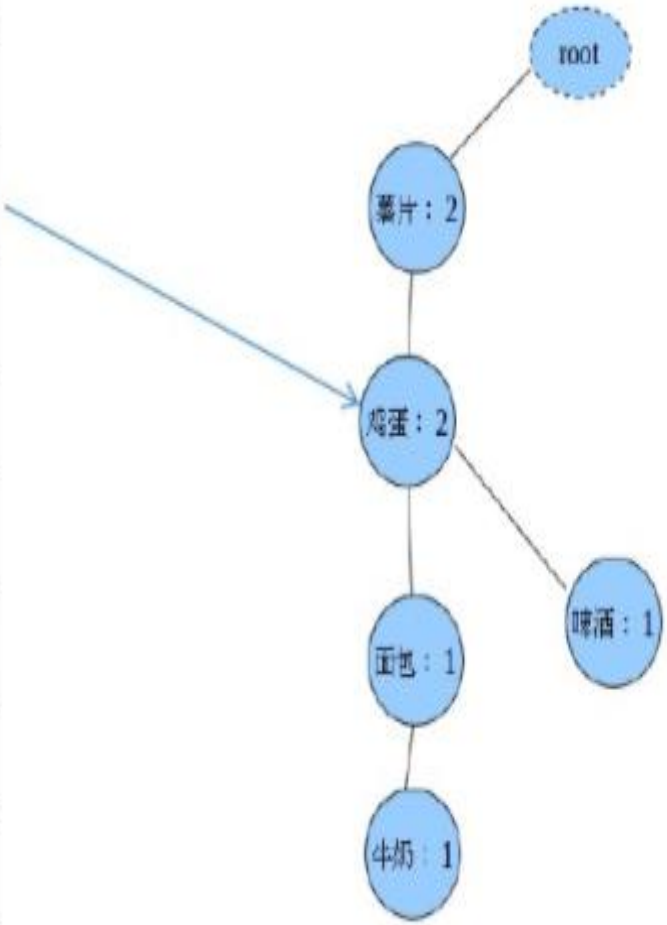
Step3.1：插入第一条（第一步有一个虚的根节点）



FP-tree算法实例-建立FP树

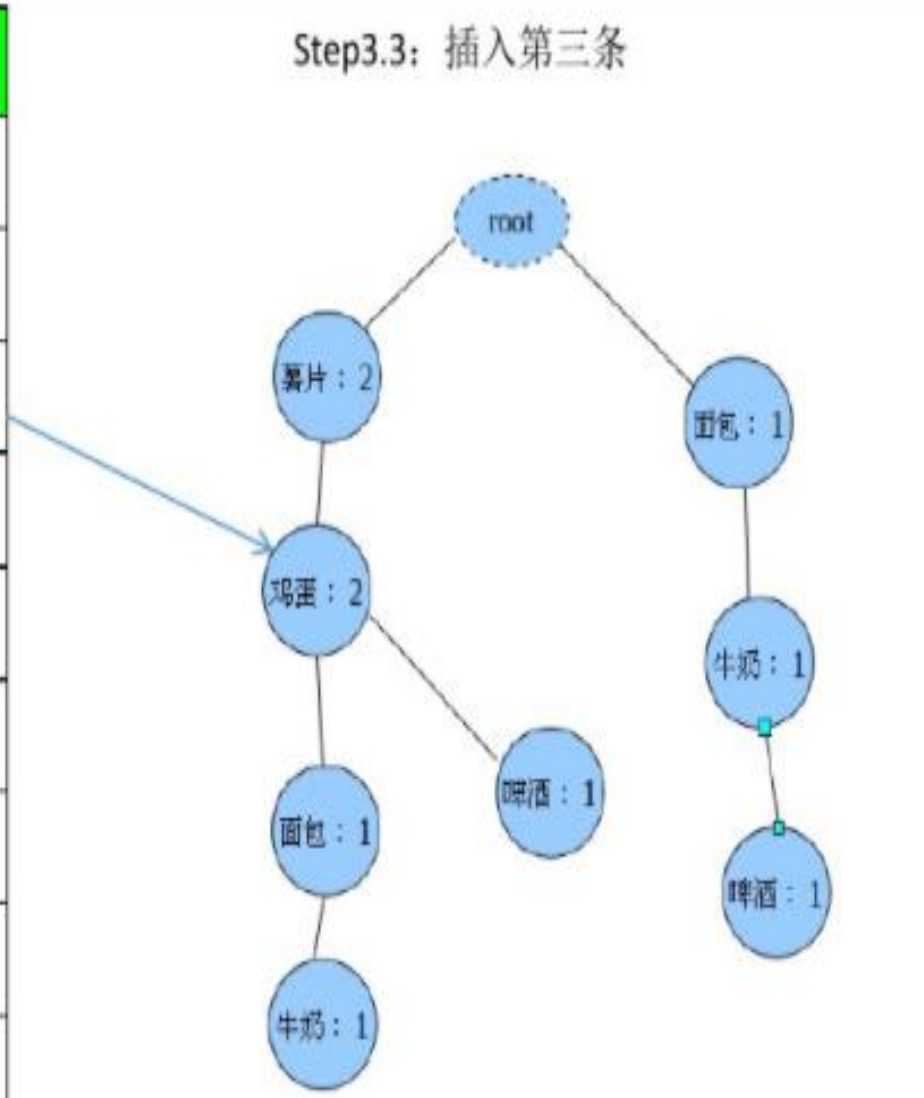
ID	Items
1	薯片, 鸡蛋, 面包, 牛奶
2	薯片, 鸡蛋, 啤酒
3	面包, 牛奶, 啤酒
4	薯片, 鸡蛋, 面包, 牛奶, 啤酒
5	薯片, 鸡蛋, 面包
6	鸡蛋, 面包, 啤酒
7	薯片, 面包, 牛奶
8	薯片, 鸡蛋, 面包, 牛奶
9	薯片, 鸡蛋, 牛奶

Step3.2：插入第二条。根结点不管，然后插入薯片，在step3.1的基础上+1，则记为2；同理鸡蛋记为2；啤酒在step3.1的树上是没有的，那么就开一个分支。



FP-tree算法实例-建立FP树

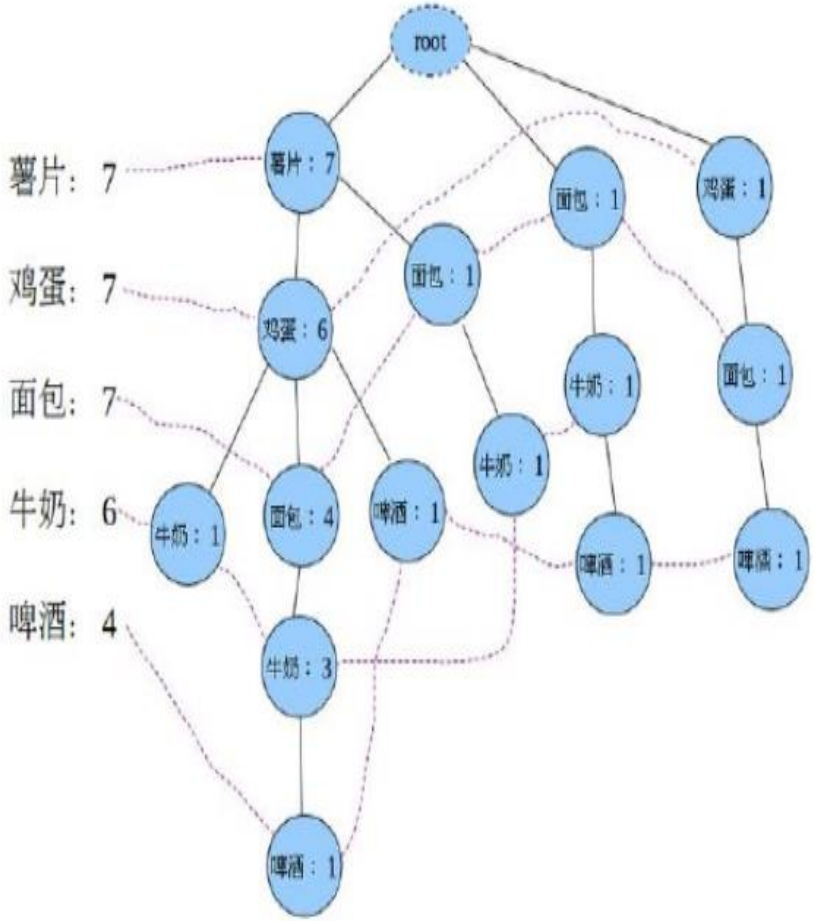
ID	Items
1	薯片, 鸡蛋, 面包, 牛奶
2	薯片, 鸡蛋, 啤酒
3	面包, 牛奶, 啤酒
4	薯片, 鸡蛋, 面包, 牛奶, 啤酒
5	薯片, 鸡蛋, 面包
6	鸡蛋, 面包, 啤酒
7	薯片, 面包, 牛奶
8	薯片, 鸡蛋, 面包, 牛奶
9	薯片, 鸡蛋, 牛奶



FP-tree算法实例-建立FP树

ID	Items
1	薯片, 鸡蛋, 面包, 牛奶
2	薯片, 鸡蛋, 啤酒
3	面包, 牛奶, 啤酒
4	薯片, 鸡蛋, 面包, 牛奶, 啤酒
5	薯片, 鸡蛋, 面包
6	鸡蛋, 面包, 啤酒
7	薯片, 面包, 牛奶
8	薯片, 鸡蛋, 面包, 牛奶
9	薯片, 鸡蛋, 牛奶

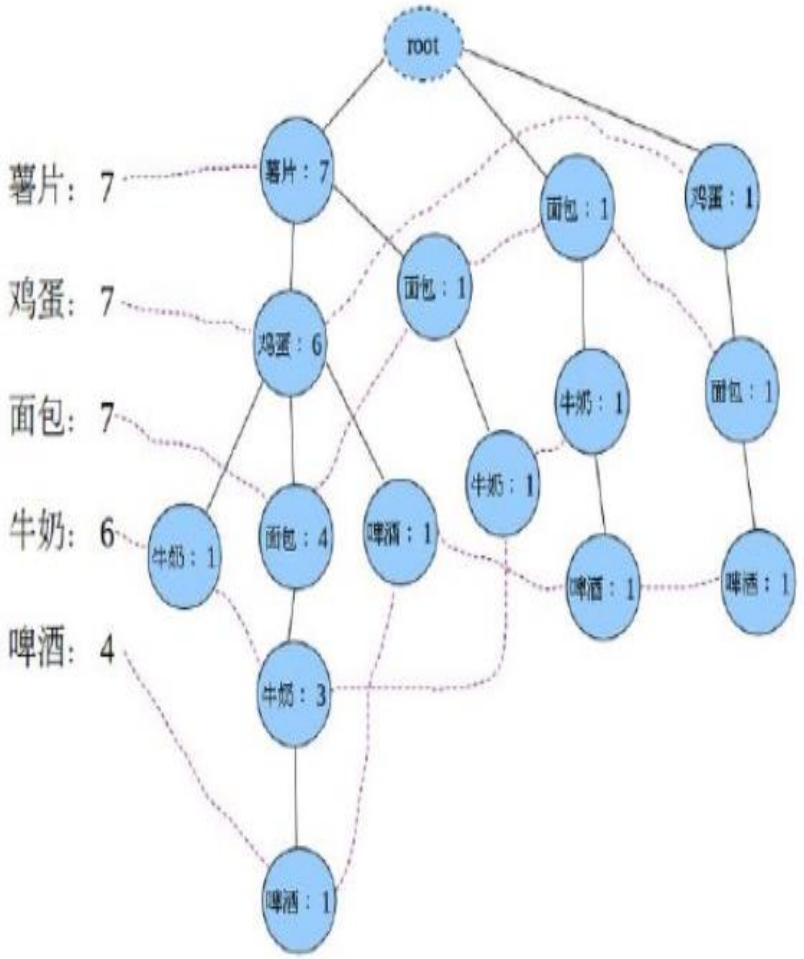
同理，剩余记录依次插入fp-tree中。



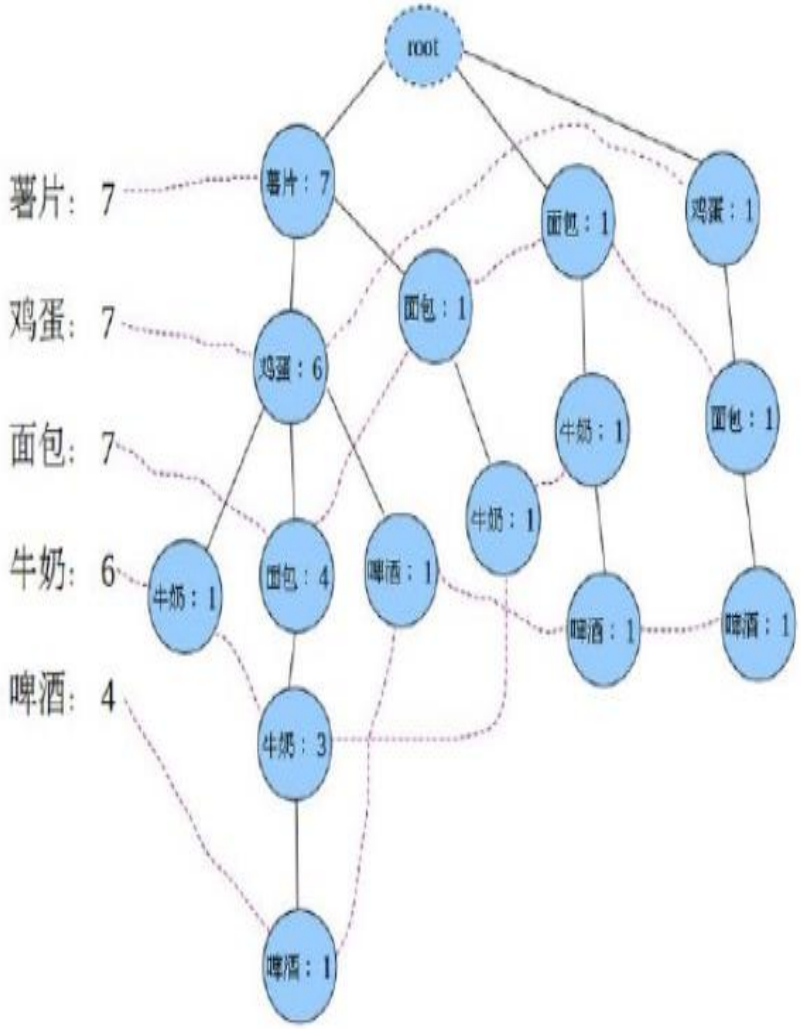
FP-tree算法实例-建立FP树

图中左边的一列叫做头指针表，树中相同名称的节点要链接起来，链表的第一个元素就是头指针表里的元素。

虚线连接起来的表示同一个商品，各个连接的数字加起来就是该商品出现的总次数。



FP-tree算法实例-挖掘频繁项集



•Step4：从FP-Tree中找出频繁项集。

遍历表头项中的每一项（以"牛奶：6"为例），从FP-Tree中找到所有的"牛奶"结点，向上遍历它的祖先结点，得到4条路径，如表所示。

ID	Items
1	薯片：7，鸡蛋：6，牛奶：1
2	薯片：7，鸡蛋：6，面包：4，牛奶：3
3	薯片：7，面包：1，牛奶：1
4	面包：1，牛奶：1

FP-tree算法实例-挖掘频繁项集

•Step4：从FP-Tree中找出频繁项集。

对于每一条路径上的节点，其count都设置为牛奶的count（路径中最末尾的商品数）

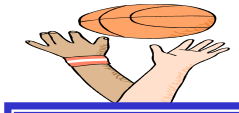
ID	Items
1	薯片：7，鸡蛋：6，牛奶：1
2	薯片：7，鸡蛋：6，面包：4，牛奶：3
3	薯片：7，面包：1，牛奶：1
4	面包：1，牛奶：1

ID	Items
1	薯片：1，鸡蛋：1，牛奶：1
2	薯片：3，鸡蛋：3，面包：3，牛奶：3
3	薯片：1，面包：1，牛奶：1
4	面包：1，牛奶：1

因为每一项末尾都是牛奶，可以把牛奶去掉，得到条件模式基，此时的后缀模式是：牛奶。

ID	Items
1	薯片：1，鸡蛋：1，牛奶：1
2	薯片：3，鸡蛋：3，面包：3，牛奶：3
3	薯片：1，面包：1，牛奶：1
4	面包：1，牛奶：1

ID	Items
1	薯片：1，鸡蛋：1
2	薯片：3，鸡蛋：3，面包：3
3	薯片：1，面包：1
4	面包：1



FP-tree算法的基本原理

- Frequent Pattern Tree：不使用侯选集，直接压缩数据库成一个频繁模式树，通过频繁模式树可以直接得到频集。进行2次数据库扫描：一次对所有1-项目的频度排序；一次将数据库信息转变成紧缩内存结构。
- 基本步骤是：
  - 1、两次扫描数据库，生成频繁模式树FP-Tree：
    - 扫描数据库一次，得到所有1-项目的频度排序表T；
    - 依照T，再扫描数据库，得到FP-Tree。
  - 2、使用FP-Tree，生成频集：
    - 为FP-tree中的每个节点生成条件模式库；
    - 用条件模式库构造对应的条件FP-tree；
    - 递归挖掘条件FP-trees同时增长其包含的频繁集：
      - 如果条件FP-tree只包含一个路径，则直接生成所包含的频繁集。

Fp-tree算法实现

1. 遍历数据集，统计各元素项出现次数，创建头指针表
2. 移除头指针表中不满足最小支持度的元素项
3. 第二次遍历数据集，创建 FP 树。对每个数据集中的项集：
   - 3.1 初始化空 FP 树
   - 3.2 对每个项集进行过滤和重排序
   - 3.3 使用这个项集更新 FP 树，从 FP 树的根节点开始：
     - 3.3.1 如果当前项集的第一个元素项存在于 FP 树当前节点的子节点中，则更新这个子节点的计数值
     - 3.3.2 否则，创建新的子节点，更新头指针表
     - 3.3.3 对当前项集的其余元素项和当前元素项的对应子节点递归 3.3 的过程

1. 构造 FP-tree 方法

FP-tree 被期望是一个存储来自数据库的项目关联及其程度的紧凑树结构，所以在 FP-tree 构造过程中，总是尽量将出现频度高的项目放在靠近根结点。因此，构造 FP-tree 需要两次数据库扫描：首先扫描数据库一次生成 1-频繁集，并把它们按降序排列，放入 L 表中；再扫描数据库一次，对每个数据库的元组，把它对应项目集的关联和频度信息放入到 FP_tree 中。

算法 3-10 FP-tree 构造算法

输入：事务数据库 DB；最小支持度阈值 Minsup。

输出：FP-tree 树，简称 T。

算法 3-12 在 FP-tree 中挖掘频繁模式

输入：构造好的 FP-tree；事务数据库 DB；最小支持度阈值 Minsup。

输出：频繁模式的完全集。

方法：Call FP-growth(FP-tree,null)。

挖掘 FP-Tree 算法的核心是 FP-growth 过程，它是通过递归调用方式实现频繁模式。

算法 3-13 FP-growth(Tree,α)

(1) IF(Tree 只含单个路径 P)THEN FOR 路径 P 中结点的每个组合(记作 β)DO
产生模式 $\beta \cup \alpha$，其支持度 support=β 中结点的最小支持度；

(2) ELSE FOR each a_i 在 FP-tree 的项头表(倒序) DO BEGIN

(2-1) 产生一个模式 $\beta = a_i \cup \alpha$，其支持度 support=a_i.support；

(2-2) 构造 β 的条件模式基，然后构造 β 的条件 FP-tree Tree β；

(2-3) if Tree $\beta \neq \phi$ THEN call FP-growth(Tree β,β)；

2. 从 FP-tree 中挖掘频繁模式方法

用 FP-tree 挖掘频繁集的基本思想是分而治之，大致过程如下：

- 对每个项，生成它的条件模式基（一个“子数据库”，由 FP-tree 中与后缀模式一起出现的前缀路径集组成），然后是它的条件 FP-tree；
- 对每个新生成的条件 FP-tree，重复这个步骤；
- 直到结果 FP-tree 为空，或只含唯一的一个路径（此路径的每个子路径对应的项目集都是频繁集）。

生成频繁模式树FP-Tree

TID	Items
100	{f, a, c, d, g, i, m, p}
200	{a, b, c, f, l, m, o}
300	{b, f, h, j, o}
400	{b, c, k, s, p}
500	{a, f, c, e, l, p, m, n}

$min\_support = 0.5$

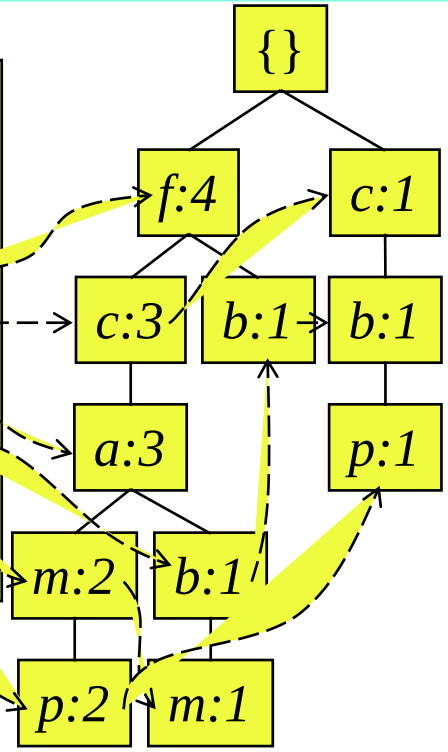
- 至少出现3次

生成频繁模式树FP-Tree

TID	Items	(ordered) frequent items
100	{f, a, c, d, g, i, m, p}	{f, c, a, m, p}
200	{a, b, c, f, l, m, o}	{f, c, a, b, m}
300	{c, f, h, j, o}	{f, b}
400	{d, c, k, s, p}	{c, b, p}
500	{g, f, c, e, l, p, m, n}	{f, c, a, m, p}

T

Item	frequency	head
f	4	
c	4	
a	3	
b	3	
m	3	
p	3	



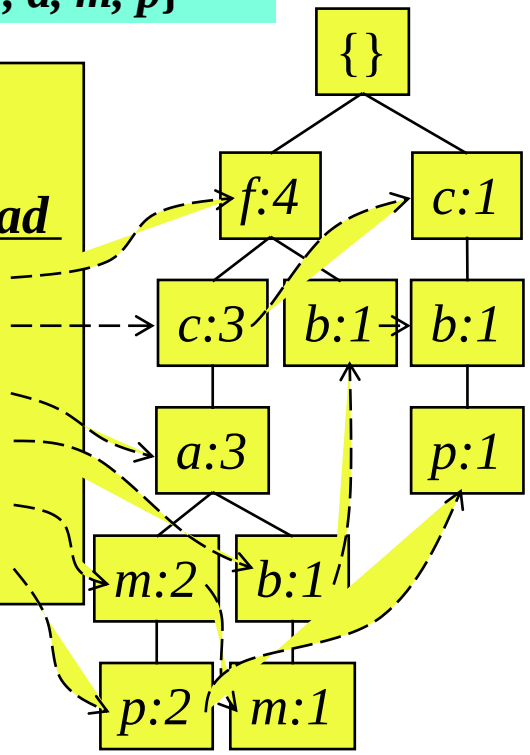
生成频繁模式树FP-Tree

TID	*Original Items*	*(ordered) frequent items*
100	{*f, a, c, d, g, i, m, p*}	{*f, c, a, m, p*}
200	{*a, b, c, f, l, m, o*}	{*f, c, a, b, m*}
300	{*b, f, h, j, o*}	{*f, b*}
400	{*b, c, k, s, p*}	{*c, b, p*}
500	{*a, f, c, e, l, p, m, n*}	{*f, c, a, m, p*}

min_support = 0.5

T

Item	*frequency*	*head*
f	*4*	
c	*4*	
a	*3*	
b	*3*	
m	*3*	
p	*3*	

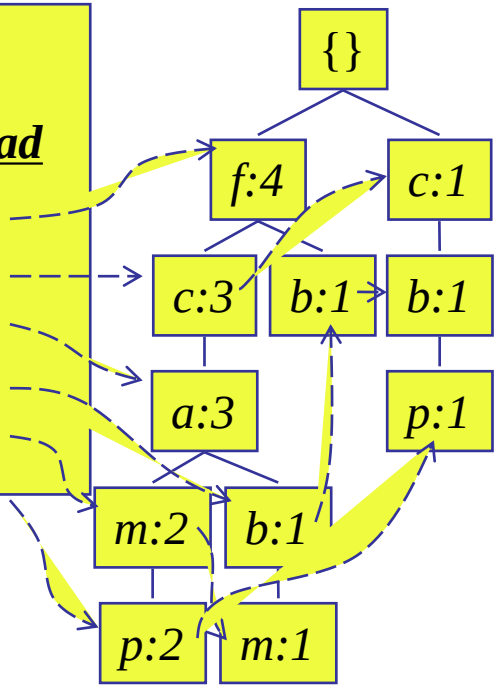


挖掘频集步骤1：生成条件模式库

- 为每个节点，寻找它的所有前缀路径并记录其频度，形成CPB

T

Item	frequency	head
f	4	
c	4	
a	3	
b	3	
m	3	
p	3	

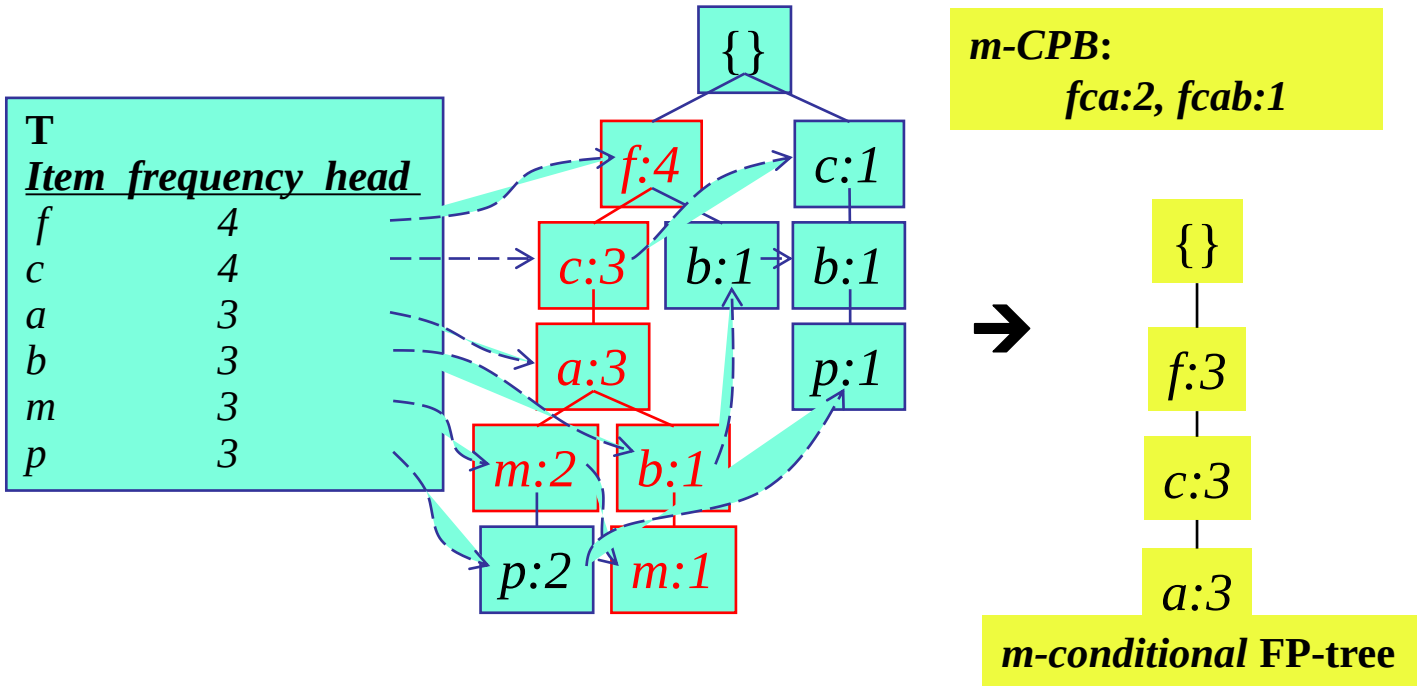


CPB

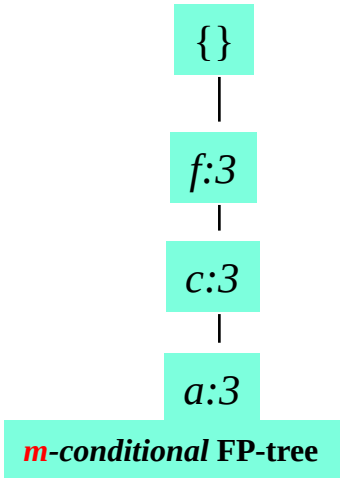
item	***cond. pattern base***
c	***f:3***
a	***fc:3***
b	***fca:1, f:1, c:1***
m	***fca:2, fcab:1***
p	***fcam:2, cb:1***

挖掘频集步骤2：构造C-FP-tree

- 为每一个节点，通过FP-tree构造一个C-FP-tree
- 例如，m节点的C-FP-tree为：



挖掘频集步骤3：递归构造C-FP-tree



***m*-conditional FP-tree**

单路径可以形成频集

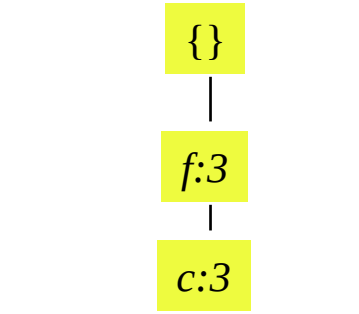
所有频集：

***m*,**

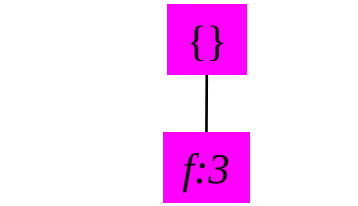
fm, cm, am,

fcm, fam, cam,

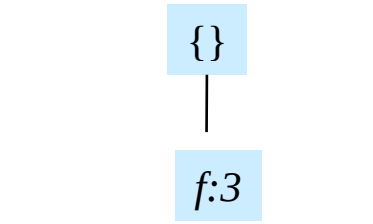
fcam



***am*-conditional FP-tree**



***cm*-conditional FP-tree**



***cam*-conditional FP-tree**