法律声明

- 课程详情请咨询
  - 微信公众号：北风教育
  - 官方网址：http://www.ibeifeng.com/

人工智能之推荐系统

推荐系统：推荐算法模型之关联规则算法

主讲人：Gerry

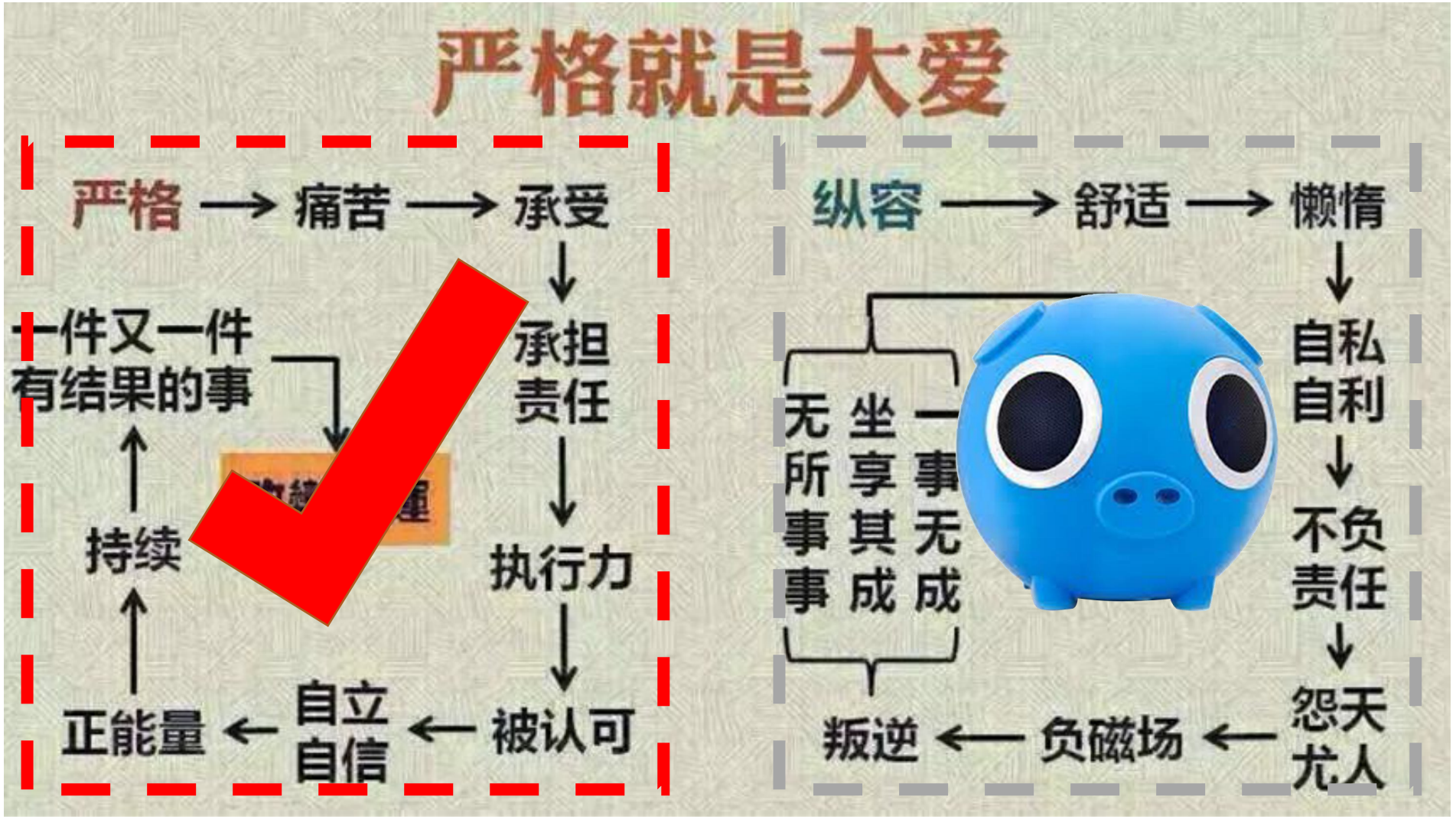
上海育创网络科技有限公司

北风网 IBEIFENG.COM

课程要求

- 课上课下“九字”真言
  - 认真听，善摘录，勤思考
  - **多温故，乐实践**，再发散
- 四不原则
  - 不懒散惰性，不迟到早退
  - 不请假旷课，不拖延作业
- 一点注意事项
  - 违反“四不原则”，不包就业和推荐就业

严格是大爱


严格就是大爱
严格
痛苦
承受
承担责任
执行力
被认可
自立自信
正能量
持续
一件又一件有结果的事
纵容
舒适
懒惰
自私自利
不负责任
怨天尤人
负磁场
叛逆
一事无成
坐享其成
无所事事

寄语

做别人不愿做的事，

做别人不敢做的事，

做别人做不到的事。

课程内容

- 基于关联规则的推荐
- Apriori关联规则算法
- FP Tree关联规则算法
- PrefixSpan关联规则算法

关联规则挖掘推荐算法

- 关联规则挖掘是一种在大规模交易中识别类似规则关系模式的通用技术，可以应用到推荐系统中。
- 交易T是所有有效产品集合P={p_1,p_2,...,p_n}的子集，表示被一起购买的产品集合，关联规则X=>Y表示只要交易T中包含了X里面的元素，那么认为Y里面的元素也有可能被T包含。常见的规则挖掘算法是Apriori算法，关联规则的衡量指标是：支持度(support)和可信度(confidence)。
- 将关联规则应用到推荐系统的主要问题就是需要将评分转换为交易，一般情况把所有的向前(正向)的评分集合<可以是做过去均值化操作后的评分矩阵>或者用户的购买行为可以看做一次交易。

Apriori算法概述

- Apriori算法是常用的用于挖掘出数据关联规则的算法，它用来找出数据值中**频繁出现**的数据集合，这些找出的集合有助于我们的业务决策，同时我们也可以认为这些**频繁出现**的数据集合中的数据项存在一定的关联性，简而言之，可以认为这些数据项之间存在某种“相似性”。
- 比如在电商的网购数据中，如果发现某一些商品经常一起被购买，那么我们可以认为这些商品之间存在某种“相似性”，从而我们可以优化网站中这些商品的排列位置、优化商品的仓库位置或者将这些“相似”的物品推荐给正在浏览对应物品的客户，从而可以达到增加经济效益、节约成本的目的。

Apriori算法概述

Apriori算法相关概念

- 交易集：包含所有数据的一个数据集合，数据集合中的每条数据都是一笔交易
- 项：交易集中的每个商品被成为一个项
- 模式/项集(ItemSet)：项组合被成为模式/项集
- 支持度(Support)：一个项集在在整个交易集中出现的次数/出现的频度，比如：Support({A,C})=2表示A和C同时出现的次数是2次
- 最小支持度：交易次数达到最小支持度的情况下，该项集才会被计算
- 频繁项集：如果项集的支持度大于等于最小支持度，那么该项集被成为频繁项集

Apriori算法相关概念

- 置信度(Confidence)：关联规则左件和右件同时出现的频繁程度，该值越大，表示同时出现的几率越大；
- 关联规则：LHS ➔ RHS(confidence) -----> 如果客户购买了左件(LHS)，也可能购买右件(RHS)，购买的置信度为confidence

Apriori算法相关概念

$$Support_{rule}(X,Y)=\frac{X\cap Y\text{的数据量}}{\text{总的数据量}}$$

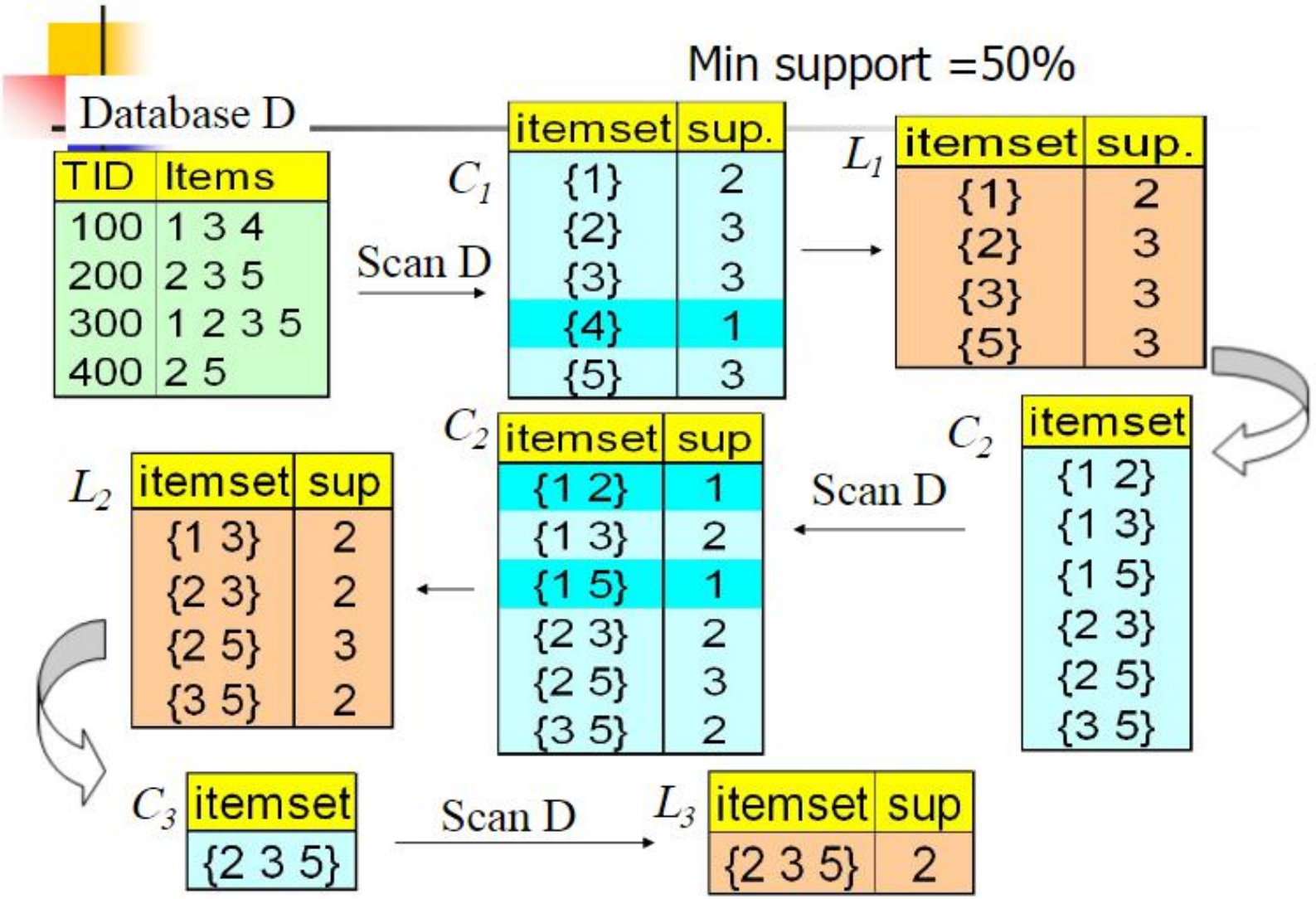
$$Confidence_{rule}(X,Y)=\frac{X\cap Y\text{的数据量}}{\text{含X的数据量}}$$

- 比如：在购物数据中，啤酒对应尿布的置信度为70%，支持度为30%。则意味着在所有购物数据中，总共有30%的用户既买啤酒又买尿布；同时在买啤酒的用户中有70%的用户购买尿布。

Apriori算法原理

- Apriori算法本质的作用是找出购物数据集中的**最频繁的K项集**；Apriori算法采用了迭代的方法，先搜索出候选1项集及对应的支持度，剪枝去掉低于最小支持度的1项集，得到频繁1项集。然后对剩下的频繁1项集进行连接，得到候选的频繁2项集，筛选去掉低于最小支持度的候选频繁2项集，得到频繁2项集，以此类推，迭代下去，直到无法找到频繁k+1项集为止，对应的频繁k项集的集合即为算法的输出结果。

Apriori算法原理



Apriori算法原理

- 输入：数据集合D，支持度阈值α；输出：最大的频繁K项集
  - 1. 扫描整个数据集，得到所有出现过的1项集，得到候选频繁1项集。
  - 2. 令k = 1
  - 3. 挖掘频繁k项集
    - 扫描数据计算候选频繁k项集的支持度
    - 去除候选频繁k项集中支持度低于阈值的数据集，得到频繁k项集。如果得到的频繁k项集为空，则直接返回频繁k-1项集的集合作为算法结果，算法结束。如果得到的频繁k项集只有一项，则直接返回频繁k项集的集合作为算法结果，算法结束。
    - 基于频繁k项集和频繁1项集，连接生成候选频繁k+1项集。
  - 4. k=k+1，转入步骤3。

Apriori算法总结

- Apriori算法是一种非常经典的频繁项集的挖掘算法，很多算法都是基于Apriori算法的一种扩展，比如：FP-Tree、GSP、CBA等等。理解掌握Apriori算法原理，对于对数据挖掘相关算法的学习具有非常好的作用。不过，现在一般很少直接使用Aprior算法来进行数据挖掘了，原因是：Apriori算法的数据挖掘效率比较低。

FP Tree算法概述

- Apriori算法作为挖掘频繁项集的算法，需要多次扫描数据，I/O瓶颈比较高，为了解决这个问题，提出了FP-Tree算法，也称为FP Growth算法；在FP Tree算法中，不管存在多少数据量，只需要扫描两次数据集，因此提高了算法的运行效率。
- FP Tree算法改进了Apriori算法的I/O瓶颈，类似BIRCH聚类，利用树结构来提高算法的执行效率，是一种利用空间换时间的一种算法效率提升方式。
- 备注：FP Tree是我们在生产环境中常用的一种数据挖掘频繁项集的算法。

FP Tree算法原理

- 为了减少I/O次数，FP Tree算法引入了一些数据结构来临时存储数据，主要包含三个部分：
  - 1. 项头表：记录所有的1项频繁集以及出现的次数，按照次数降序排列。
  - 2. FP Tree：将原始数据集映射到内存中的一棵FP树。
  - 3. 节点链表：也就是原始数据记录，项头表中的任意一个项都是一个节点链表的头。
- FP Tree算法可以分为一下两个过程：
  - 1. 项头表和FP Tree的构建
  - 2. FP Tree的挖掘

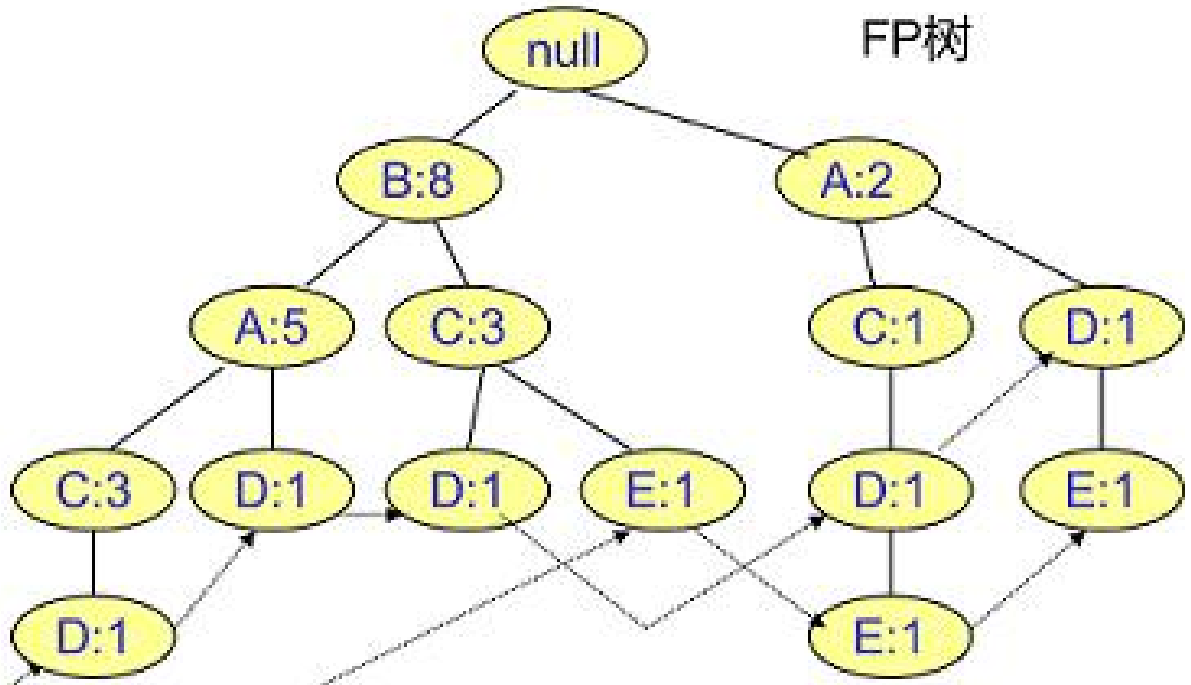
FP Tree算法原理

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

原始数据

项头表

Item	Pointer
B	8
A	7
C	7
D	5
E	3



FP-Tree算法原理之项头表构建

- 扫描所有数据，得到所有一项集的支持度，然后删除支持度低于阈值的项，得到频繁一项集，将所有频繁一项集按照支持度降序排列，放入项头表中。

数据

A B C E F O
A C G
E I
A C D E G
A C E G L
E J
A B C E F P
A C D
A C E G M
A C E G N

项头表.
支持度大于20%

A:8	
C:8	
E:8	
G:5	
B:2	
D:2	
F:2	

FP-Tree算法原理之FP Tree构建

- FP Tree树的构建是FP-Tree算法的关键点，主要分为两个过程：
  - 1. 扫描数据，对于每条数据删除非频繁的1项集，并按照支持度降序排列，得到排序后的数据集(称为节点链表)。
  - 2. 基于排序好的数据集构建FP Tree。初始状态FP树是空的，建立FP树时我们一条条的读入排序后的数据集，插入FP树，插入时按照排序后的顺序，插入FP树中，排序靠前的节点是祖先节点，而靠后的是子孙节点。如果有共用的祖先，则对应的公用祖先节点计数加1。插入后，如果有新节点出现，则项头表对应的节点会通过节点链表链接上新节点。直到所有的数据都插入到FP树后，FP树的建立完成。

FP-Tree算法原理之FP Tree构建

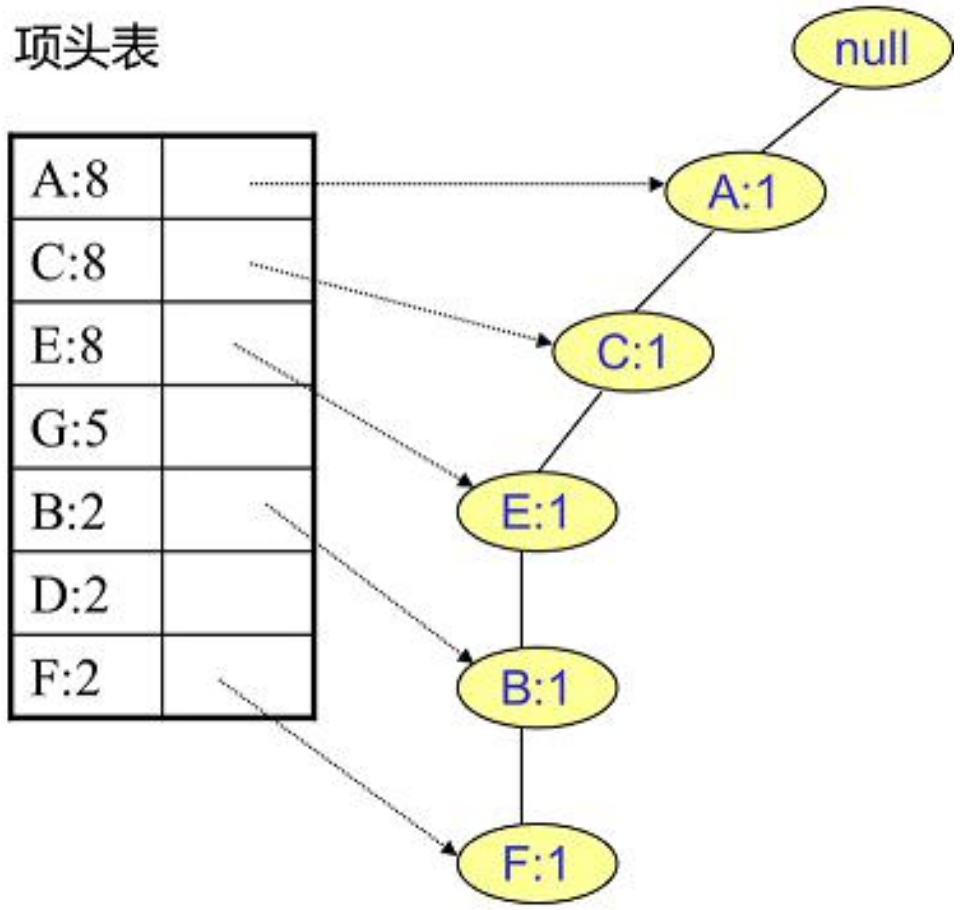
- 获取排序数据：

数据	项头表. 支持度大于20%	排序后的数据集
A B C E F O	A:8	A C E B F
A C G	C:8	A C G
E I	E:8	E
A C D E G	G:5	A C E G D
A C E G L	B:2	A C E G
E J	D:2	E
A B C E F P	F:2	A C E B F
A C D		A C D
A C E G M		A C E G
A C E G N		A C E G

FP-Tree算法原理之FP Tree构建

- 插入第一条数据：

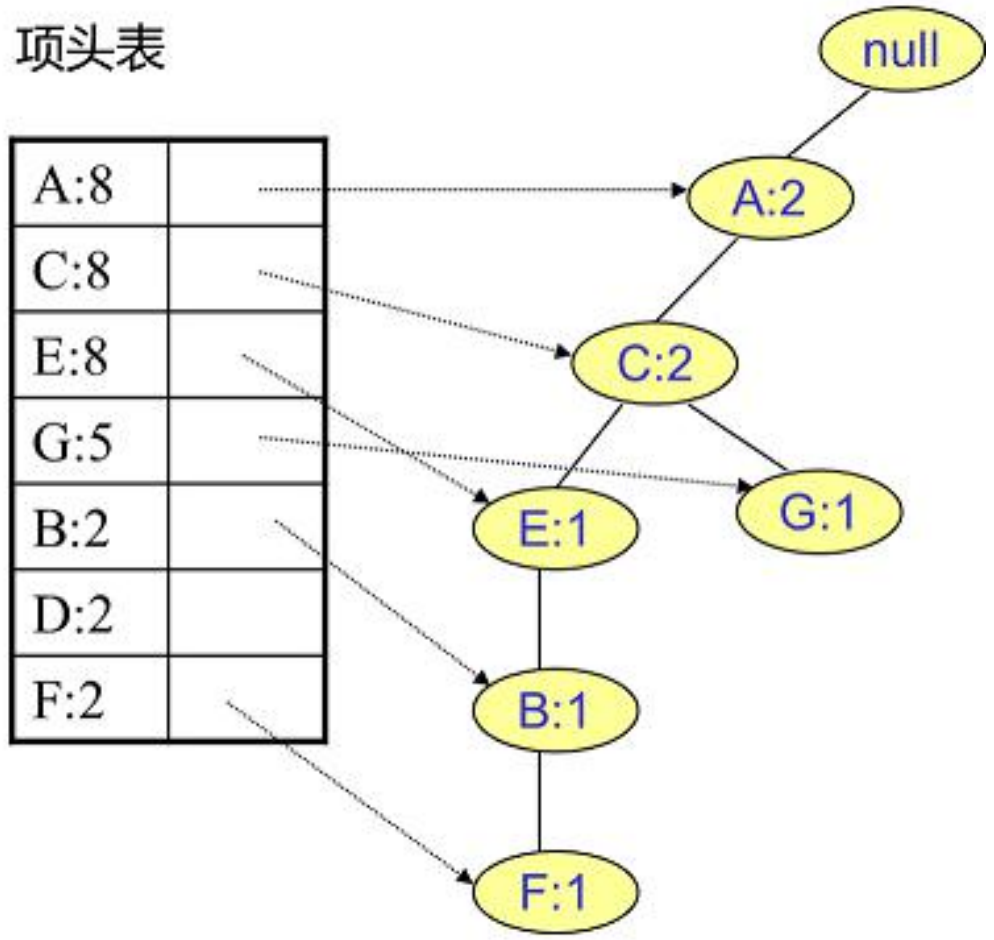
A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree算法原理之FP Tree构建

- 插入第二条数据：

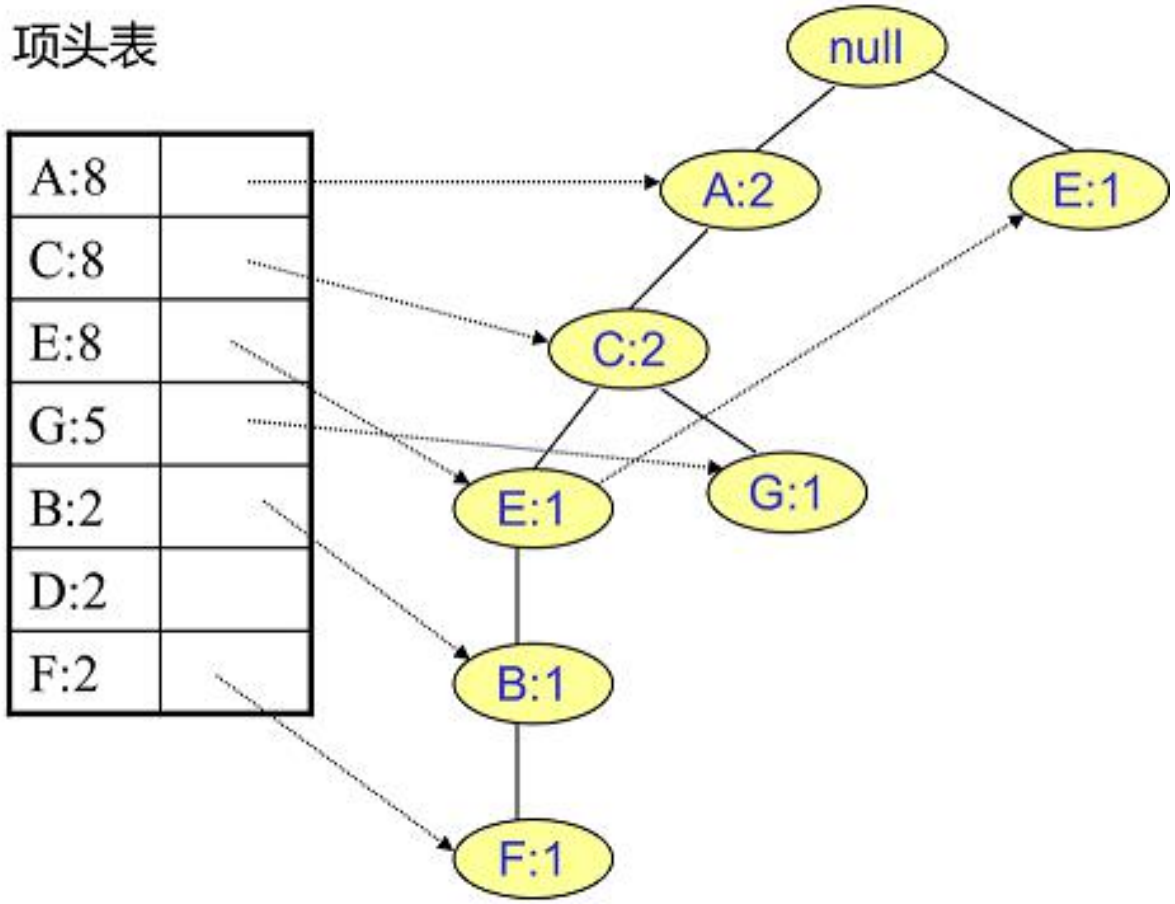
A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree算法原理之FP Tree构建

- 插入第三条数据：

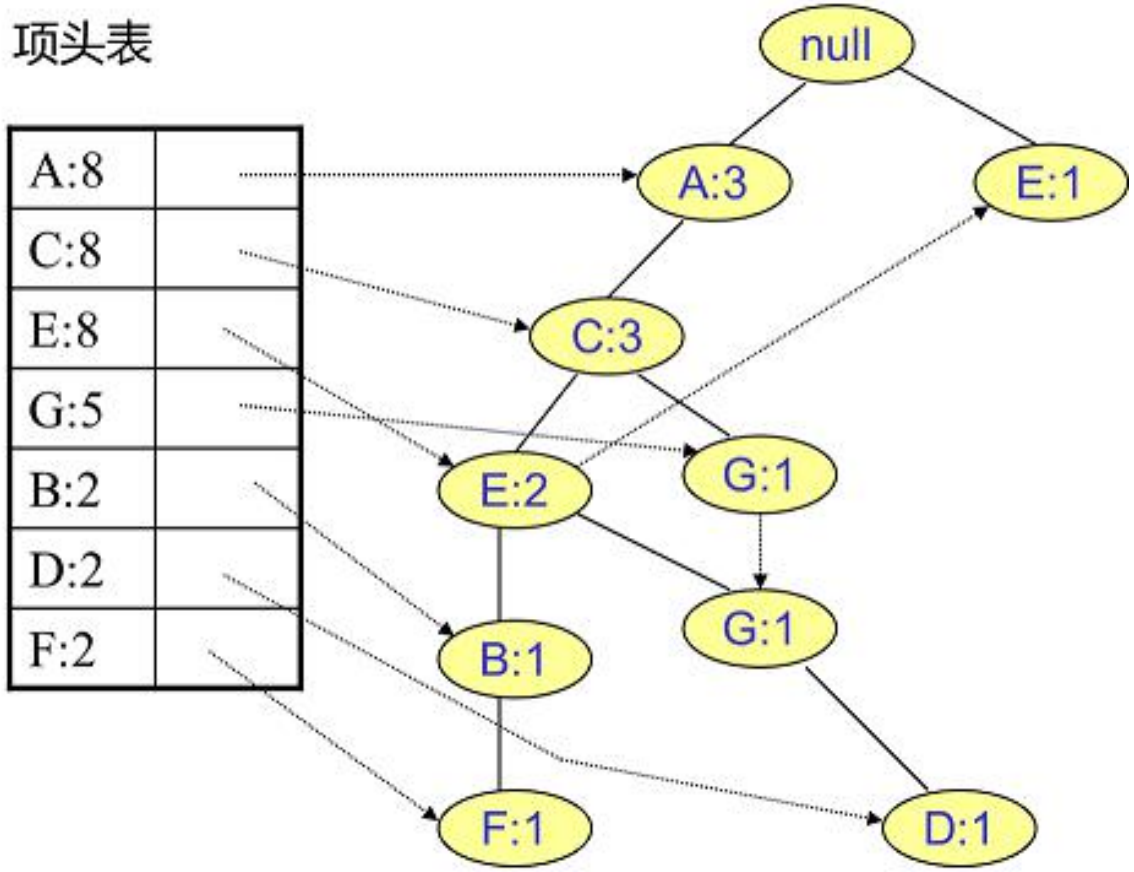
A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree算法原理之FP Tree构建

- 插入第四条数据：

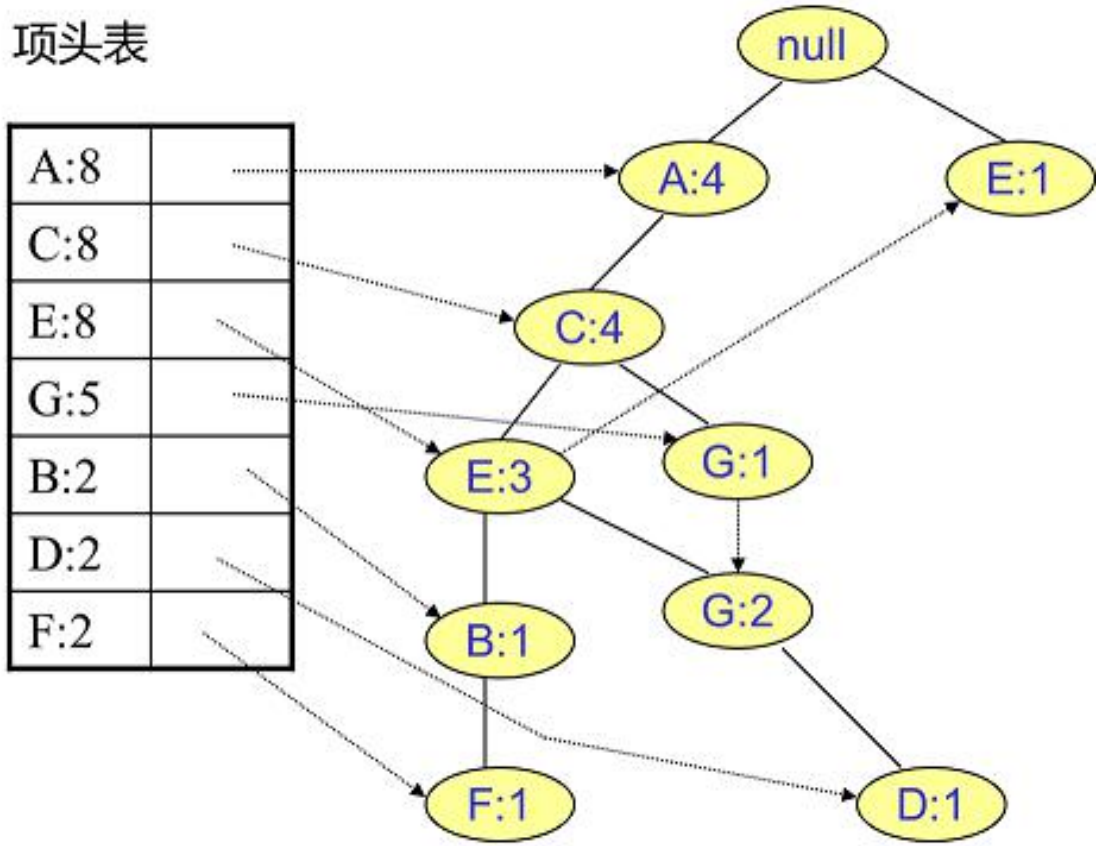
A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree算法原理之FP Tree构建

- 插入第五条数据：

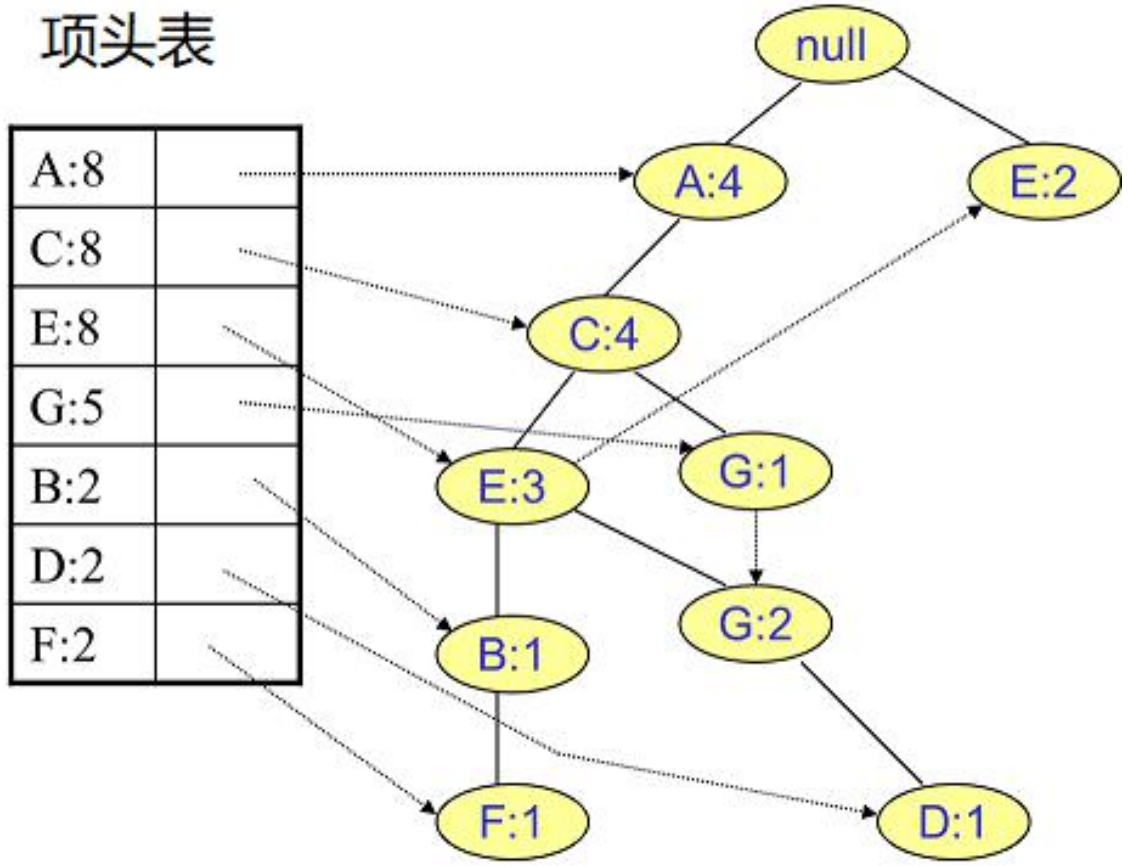
A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree算法原理之FP Tree构建

- 插入第六条数据：

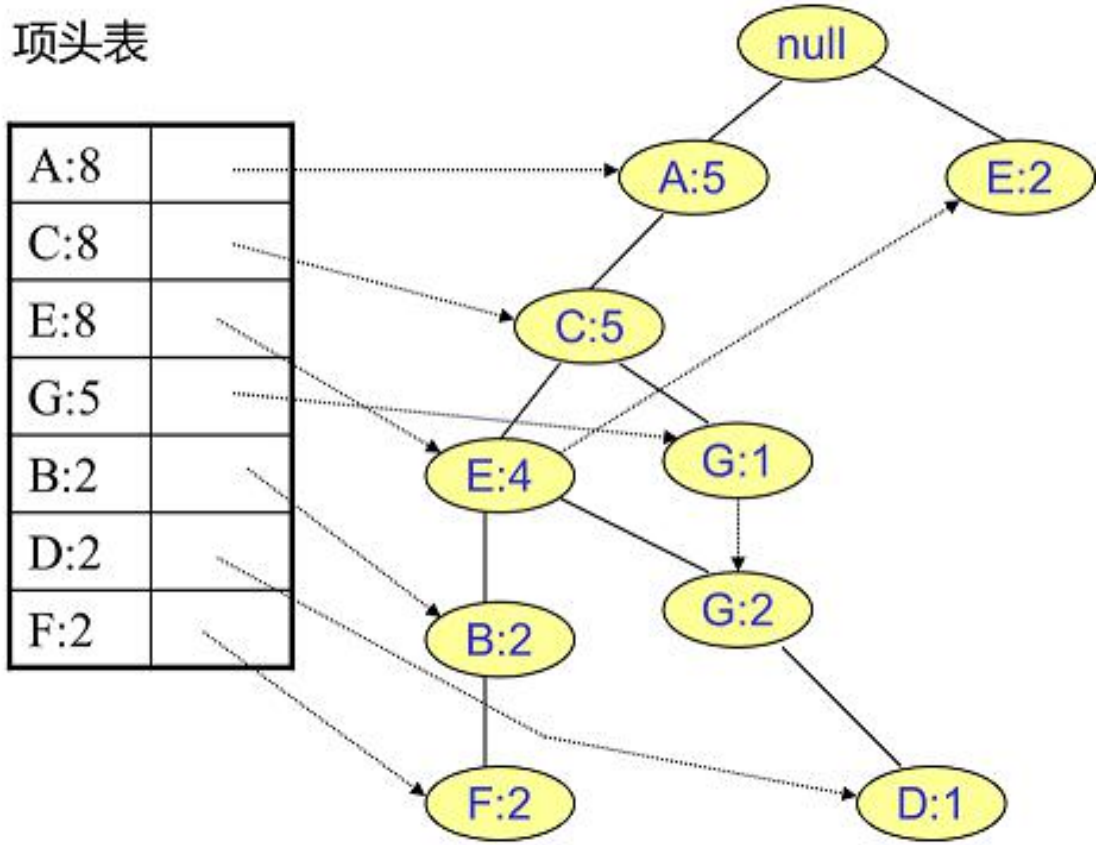
A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree算法原理之FP Tree构建

- 插入第七条数据：

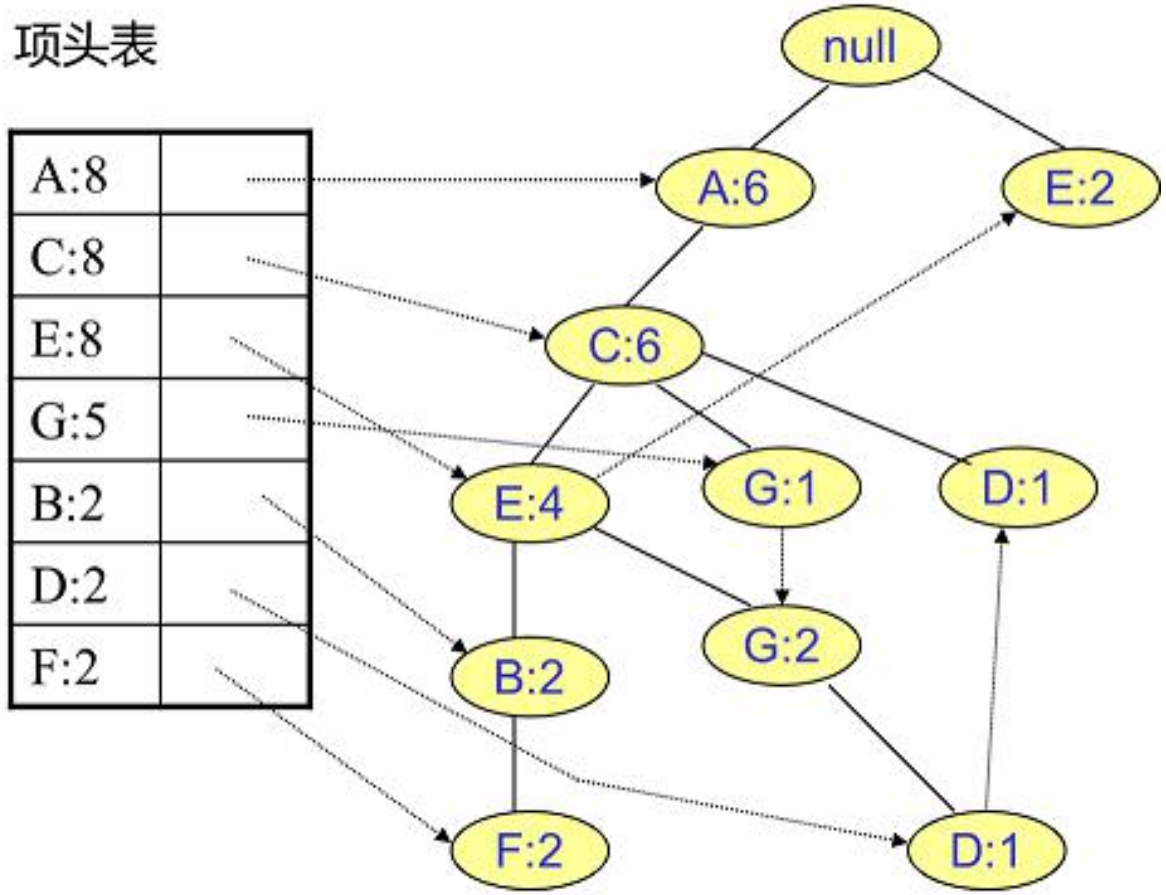
A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree算法原理之FP Tree构建

- 插入第八条数据：

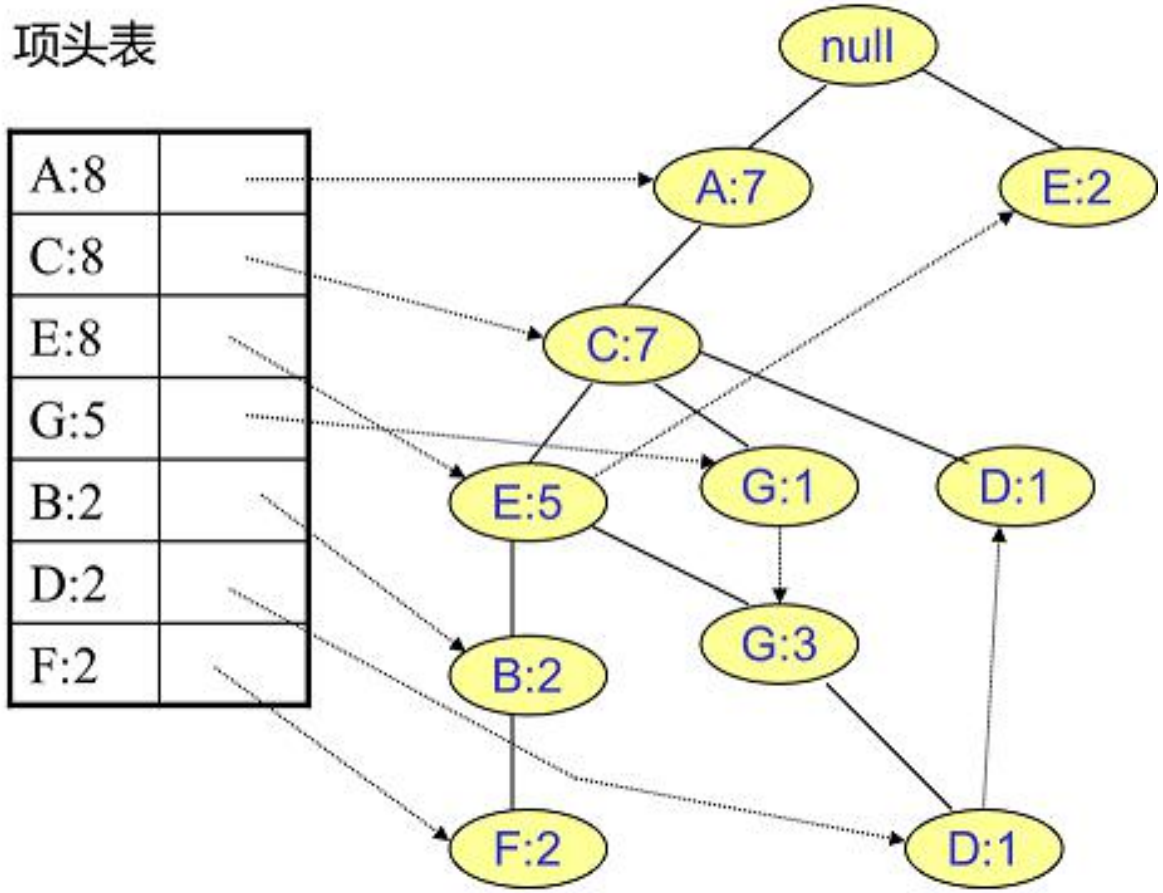
A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree算法原理之FP Tree构建

- 插入第九条数据：

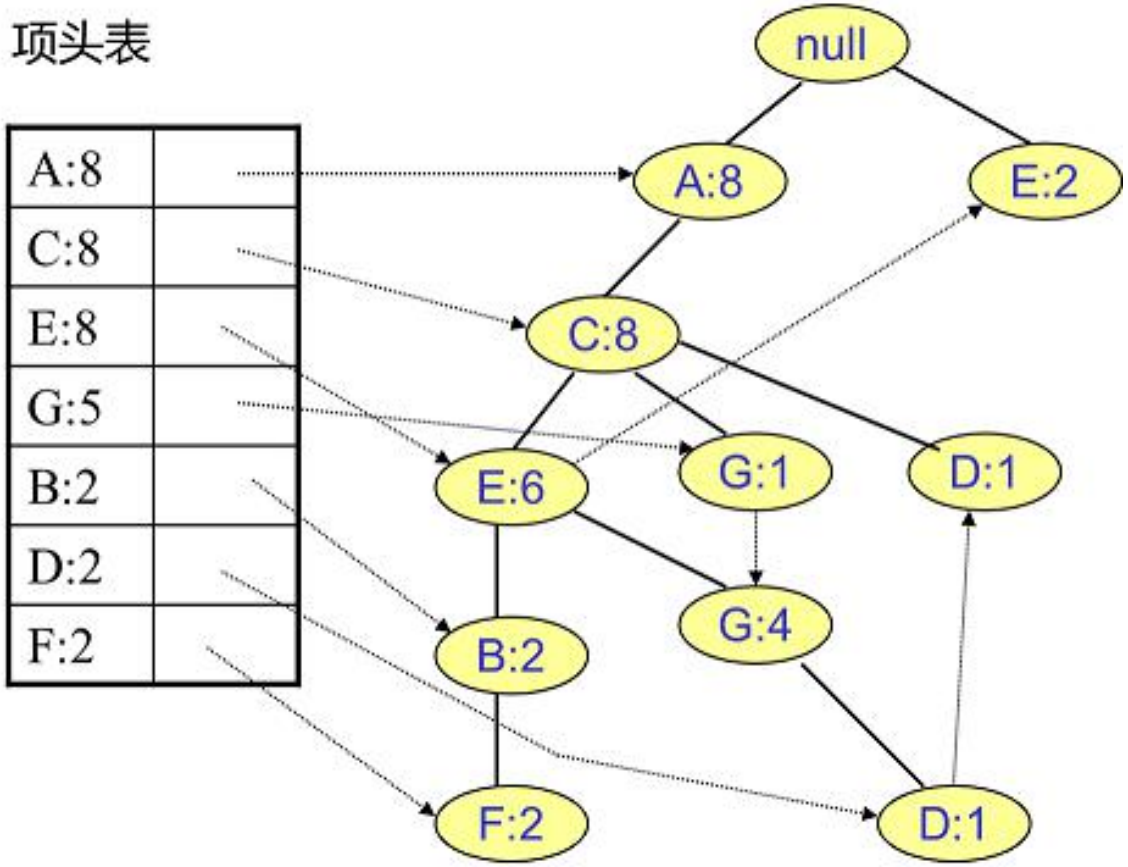
A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree算法原理之FP Tree构建

- 插入第十条数据：

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G

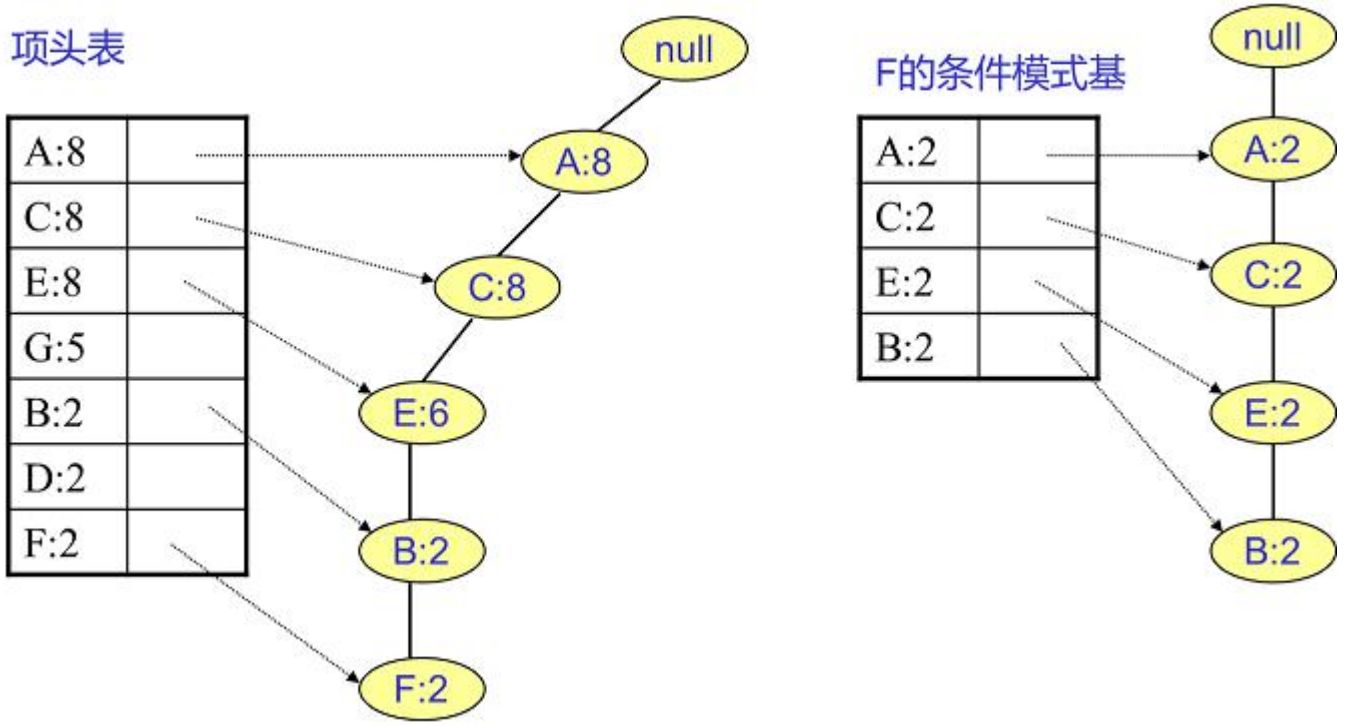


FP-Tree算法原理之FP Tree挖掘

- 当构建好FP树、项头表以及节点链表后，就可以开始进行频繁项集的挖掘了。首先从项头表的底部项依次向上挖掘，对于项头表对应于FP树的每一项，找出对应的条件模式基，所谓条件模式基是以我们要挖掘的节点作为叶子节点所对应的FP子树，得到这个FP子树，我们将子树中每个节点的的计数设置为叶子节点的计数，并删除计数低于支持度的节点。从这个条件模式基，我们就可以递归挖掘得到频繁项集了。

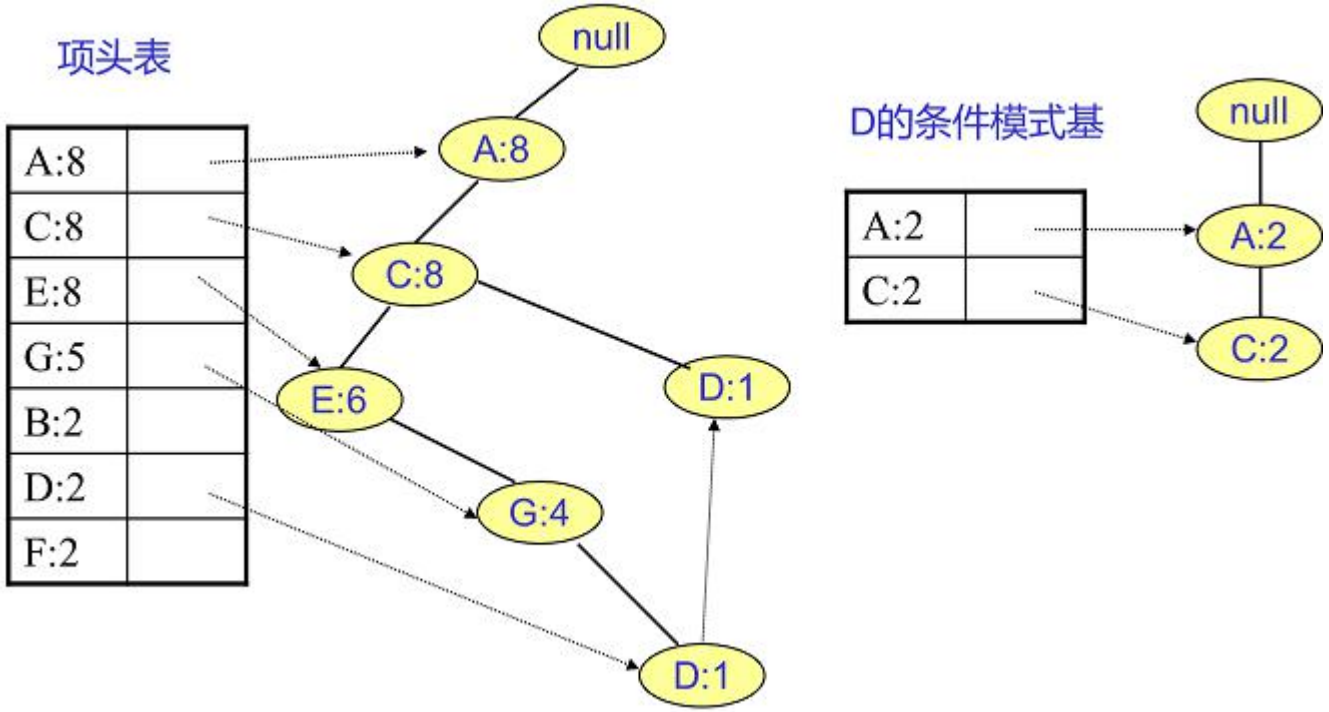
FP-Tree算法原理之FP Tree挖掘

- 寻找F节点的条件模式基
  - 我们很容易得到F的频繁2项集为{A:2,F:2}、{C:2,F:2}、{E:2,F:2}、{B:2,F:2}。递归合并二项集，得到频繁三项集为{A:2,C:2,F:2}、{A:2,E:2,F:2},…。当然一直递归下去，最大的频繁项集为频繁5项集，为{A:2,C:2,E:2,B:2,F:2}



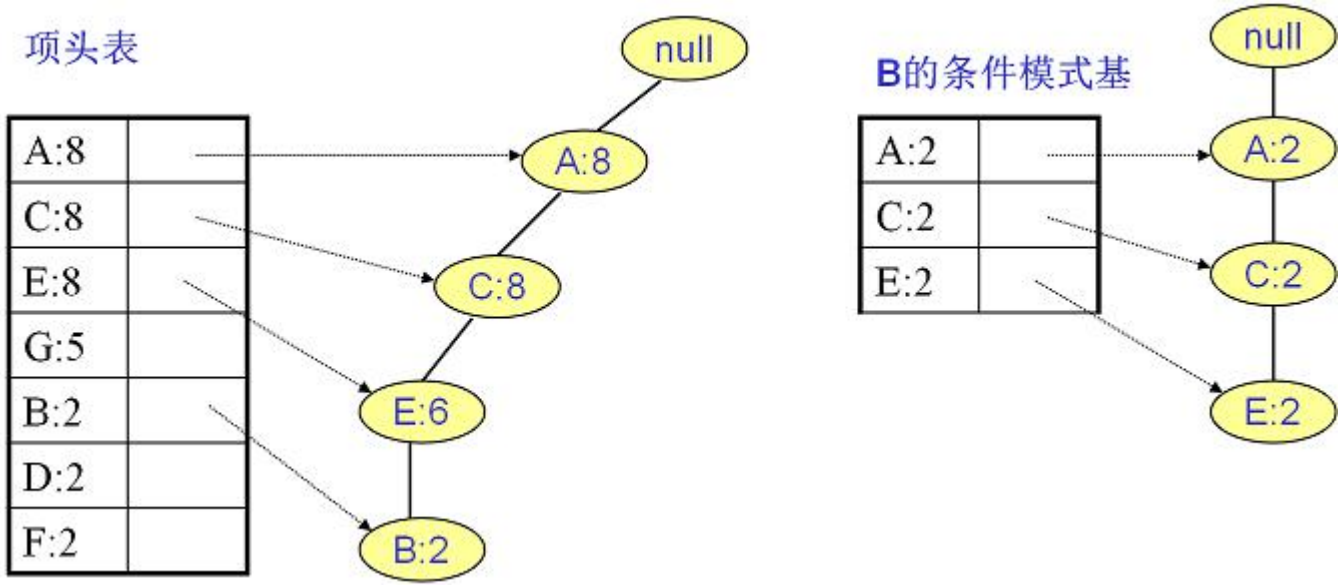
FP-Tree算法原理之FP Tree挖掘

- 寻找D节点的条件模式基
  - D的频繁2项集为{A:2,D:2}、{C:2,D:2}。递归合并二项集，得到频繁三项集为{A:2,C:2,D:2}。D对应的最大的频繁项集为频繁3项集



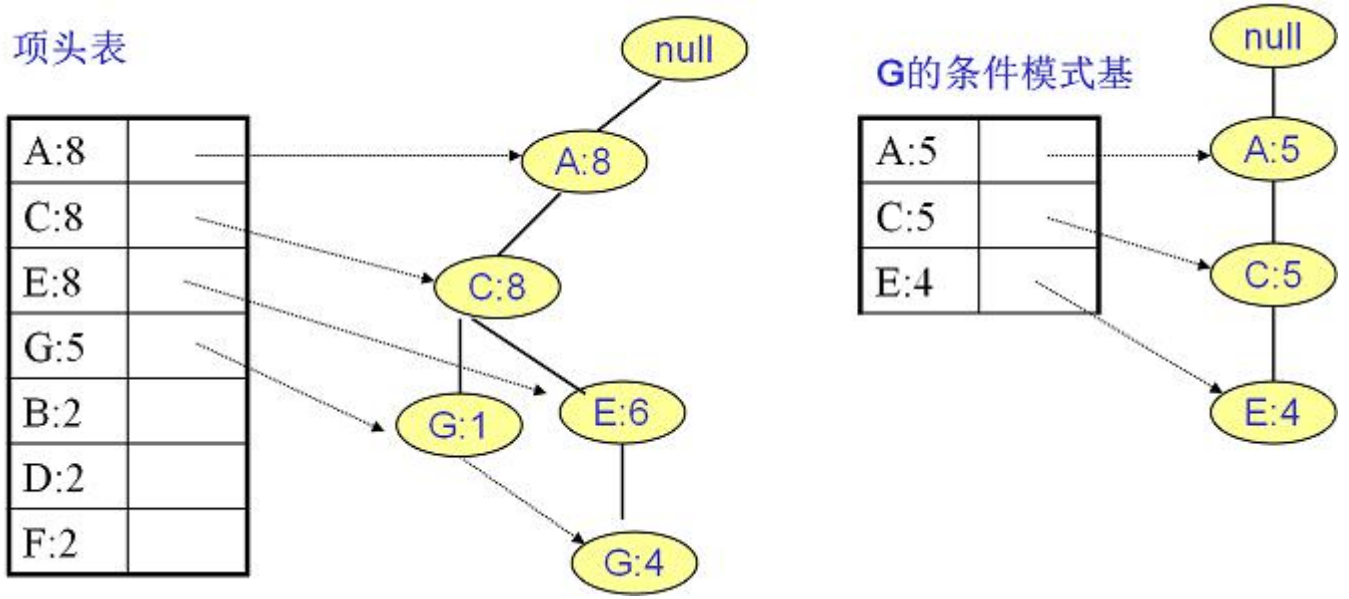
FP-Tree算法原理之FP Tree挖掘

- 寻找B节点的条件模式基
  - B的频繁2项集为{A:2,B:2}、{C:2,B:2}、{E:2,B:2}。递归合并二项集，得到频繁三项集为{A:2,C:2,B:2}、{A:2,E:2,B:2}、{E:2,C:2,B:2}。递归挖掘到B的最大频繁项集为频繁4项集{A:2, C:2, E:2,B:2}。



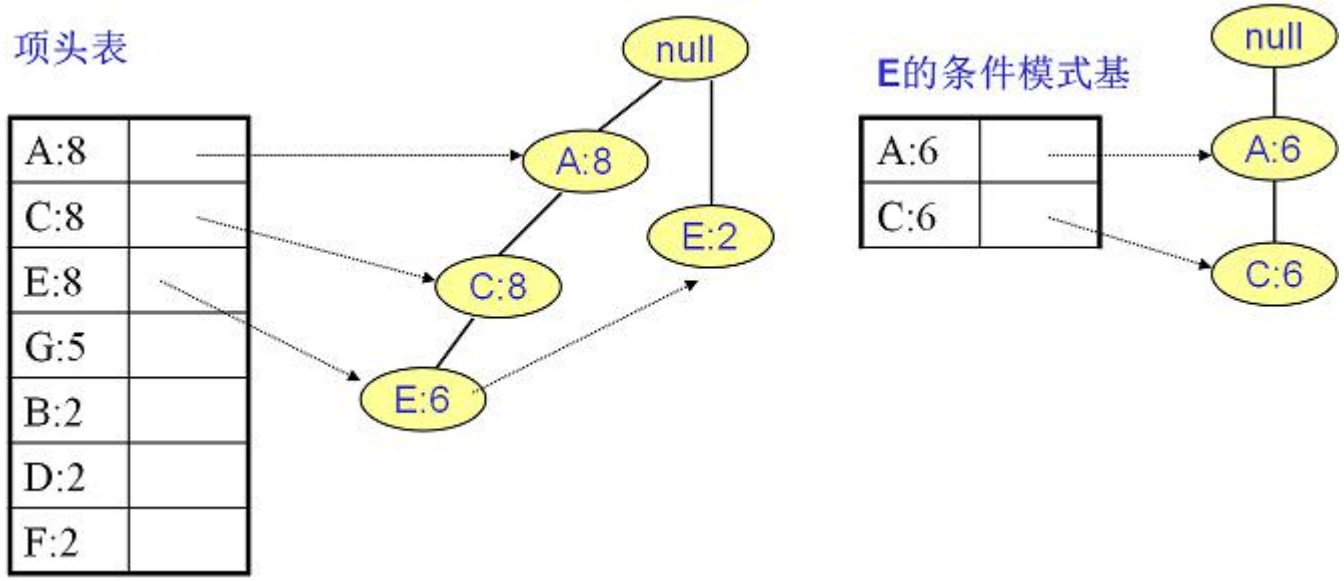
FP-Tree算法原理之FP Tree挖掘

- 寻找G节点的条件模式基
  - G的频繁2项集为{A:5,G:5}、{C:5,G:5}、{E:4,G:4}。递归合并二项集，得到频繁三项集为{A:5,C:5,G:5}、{A:4,E:4,G:4}、{E:4,C:4,G:4}。递归挖掘到G的最大频繁项集为频繁4项集{A:4, C:4, E:4, G:4}。



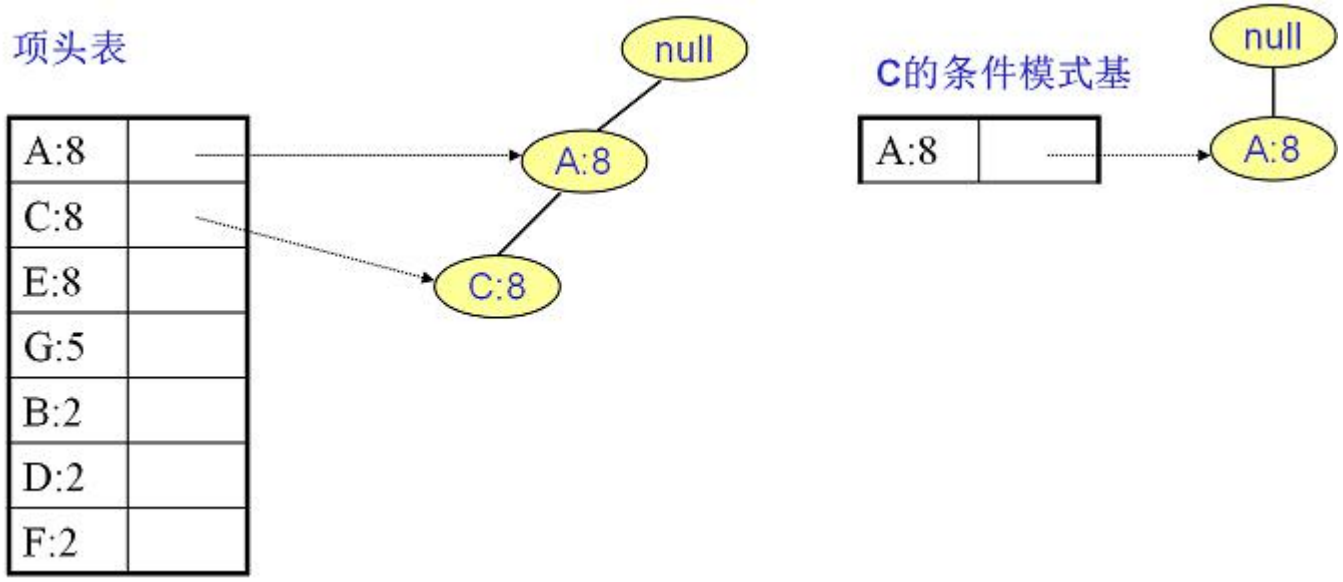
FP-Tree算法原理之FP Tree挖掘

- 寻找E节点的条件模式基
  - E的频繁2项集为{A:6,E:6}、{C:6,E:6}。递归合并二项集。递归挖掘到E的最大频繁项集为频繁3项集{A:6, C:6, E:6}。



FP-Tree算法原理之FP Tree挖掘

- 寻找C节点的条件模式基
  - C的频繁2项集为{A:8,C:8}，即C的最大频繁项集为频繁2项集{A:8,C:8}。



FP-Tree算法原理之FP Tree挖掘

- 频繁2项集：
  - {A:8,C:8}、{A:6,E:6}、{C:6,E:6}、{A:5,G:5}、{C:5,G:5}、{E:4,G:4}、{A:2,B:2}、{C:2,B:2}、{E:2,B:2}、{A:2,D:2}、{C:2,D:2}、{A:2,F:2}、{C:2,F:2}、{E:2,F:2}、{B:2,F:2}
- 频繁3项集：
  - {A:6, C:6, E:6}、{A:5,C:5,G:5}、{A:4,E:4,G:4}、{E:4,C:4,G:4}、{A:2,C:2,B:2}、{A:2,E:2,B:2}、{E:2,C:2,B:2}、{A:2,C:2,D:2}、{A:2,C:2,F:2}、{A:2,E:2,F:2}、{A:2,B:2,F:2}、{C:2,E:2,F:2}、{C:2,B:2,F:2}、{E:2,B:2,F:2}
- 频繁4项集：
  - {A:2,C:2,E:2,F:2}、{A:2,C:2,B:2,F:2}、{C:2,E:2,B:2,F:2}
- 频繁5项集：
  - {A:2,C:2,E:2,B:2,F:2}

FP-Tree算法总结归纳

- FP-Tree算法流程主要包括一下几步：
  - 扫描数据，得到所有的频繁1项集的计数，然后删除支持度低于阈值的项，将1项频繁集放入项头表，并按照支持度降序排列。
  - 读取数据集中的数据，将数据中的非频繁1项集删除，并按照支持度排序排列后将数据插入到FP树中，插入时按照排序后的顺序插入，并计算当前节点的后序子孙节点的数目。直到所有数据均插入到FP树后，FP树构建完成。
  - 从项头表的底部项依次向上找到项头表项对应的条件模式基。从条件模式基递归挖掘得到项头表项项的频繁项集。
  - 如果不限制频繁项集的项数，则返回上一步骤的所有的频繁项集，否则只返回满足项数要求的频繁项集。

PrefixSpan算法概述

- PrefixSpan全称Prefix-Projected Pattern Growth(即前缀投影的模式挖掘)，是用于挖掘频繁序列的数据挖掘算法，和Apriori算法以及FP Tree算法的挖掘目标稍有不同。PrefixSpan算法是生产中常用的一种频繁序列模式挖掘算法。
- 备注：序列中的项集是具有时间上的先后关系的。

项集数据

TID	itemsets
10	a, b, d
20	a, c, d
30	a, d, e
40	b, e, f

序列数据

SID	sequences
10	<a(abc)(ac)d(cf)>
20	<(ad)c(bc)(ae)>
30	<(ef)(ab)(df)cb>
40	<eg(af)cbc>

PrefixSpan算法基本概念

- 子序列：如果某个序列A所有的项集在序列B中都可以找到，则A是B的子序列。

$$A=\{a_1,a_2,...,a_n\} \quad B=\{b_1,b_2,...,b_m\} \quad n\le m \quad 1\le j_1\le j_2\le ....\le j_n\le m$$

$$a_1\subseteq b_{j_1},a_2\subseteq b_{j_2},.....,a_n\subseteq b_{j_n}$$

- 频繁序列：出现频次超过支持度的子序列就叫做频繁序列。
- 前缀序列：即序列前面部分的子序列。

$$A=\{a_1,a_2,...,a_n\} \quad B=\{b_1,b_2,...,b_m\} \quad n\le m$$

$$a_1=b_1,a_2=b_2,...,a_{n-1}=b_{n-1},a_n\subseteq b_n$$

序列<a(abc)(ac)d(cf)>的前缀和后缀例子

前缀	后缀(前缀投影)
<a>	<(abc)(ac)d(cf)>
<aa>	<(_bc)(ac)d(cf)>
<ab>	<(_c)(ac)d(cf)>

- 后缀序列：即序列中位于前缀序列之后的子序列就叫做后缀序列。
- 前缀投影：即投影数据库，即序列数据库S中所有相对于前缀的后缀序列的集合。

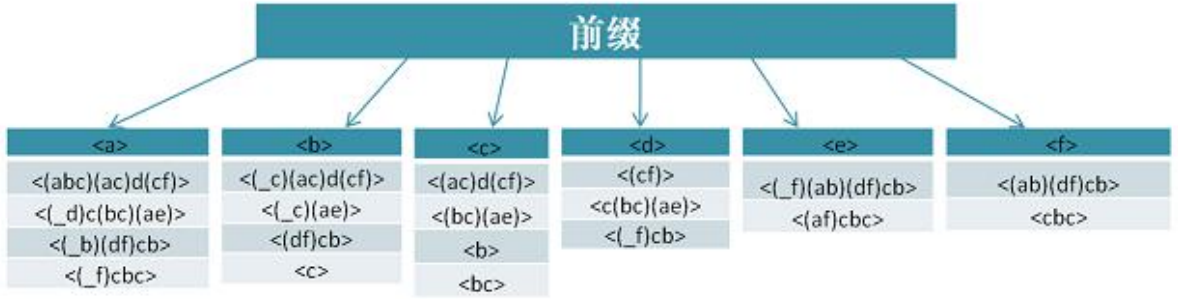
PrefixSpan算法原理

- 类似Apriori算法，先找出所有子序列中长度为1的前缀开始挖掘序列模型(并且删除原始序列中非频繁的长度为1的序列)，搜索对应的投影数据库得到长度为1的前缀对应的频繁序列，然后递归的挖掘长度为2的前缀所对应的频繁序列，。。。。以此类推，一直递归到不能挖掘到更长的前缀挖掘为止。

id	Sequence
10	<a(abc)(ac)d(cf)>
20	<(ad)c(bc)(ae)>
30	<(ef)(ab)(df)cb>
40	<eg(af)cbc>

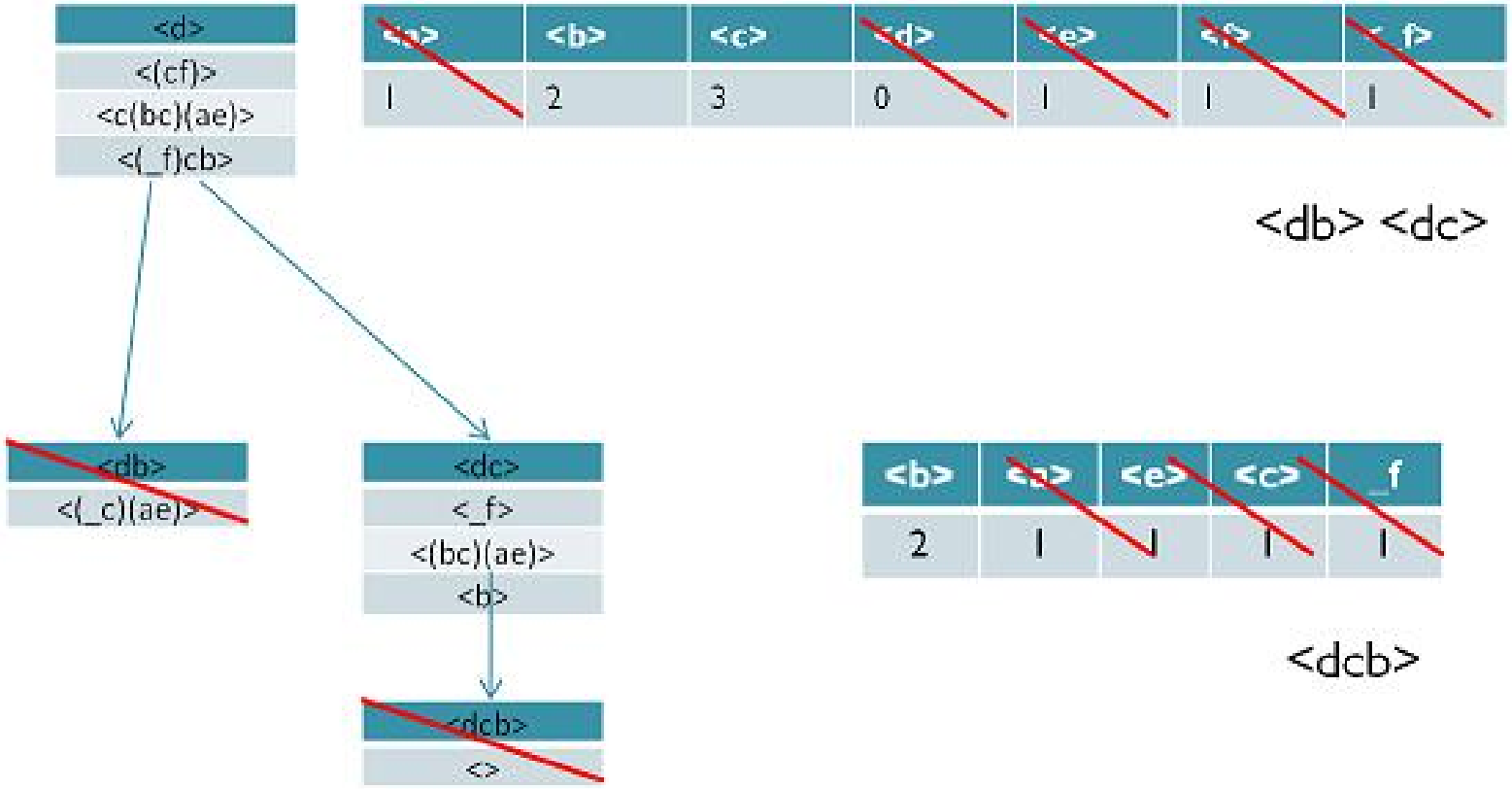
<a><b><c><d><e><f>

<a>	<b>	<c>	<d>	<e>	<f>	<g>
4	4	4	3	3	3	1



PrefixSpan算法原理

- 开始频繁序列挖掘



PrefixSpan算法流程

- 输入：序列数据库S和支持度阈值α
- 输出：所有满足支持度要求的频繁序列集
- 步骤：(备注：所找到的前缀即频繁序列)
  - 1. 找出所有子序列中长度为1的前缀以及对应的投影数据库
  - 2. 对于长度为1的前缀进行计数，将支持度低于阈值α的前缀对应的项从序列数据库S中删除，同时得到所有的频繁1项序列。
  - 3. 对于每个长度为i满足支持度的前缀进行递归挖掘：
    - a. 找出前缀对应的投影数据库，如果投影数据库为空，则递归返回
    - b. 统计对应投影数据库中各项的支持度计数，如果所有项的支持度计数都低于阈值α，则递归返回。
    - c. 将满足支持度计数的各个单项和当前的前缀进行合并，得到若干新的前缀。
    - d. 令i=i+1，前缀为合并单项后的各个前缀，分别递归执行第三步。

协同过滤各种方式总结

- 广义的协同过滤算法主要包括三种算法：
  - 基于用户(UserCF)的协同过滤算法；
  - 基于物品(ItemCF)的协同过滤算法；
  - 基于模型(ModelCF)的协同过滤算法；
    - 使用关联规则的协同过滤；
    - 使用聚类算法的协同过滤；
    - 使用分类算法的协同过滤；
    - 使用回归的协同过滤；
    - 使用矩阵分解/隐语义模型的协同过滤；
    - 使用神经网络的协同过滤；

THANK YOU

上海育创网络科技有限公司