关联规则挖掘

七月算法　寒老师

2016年7月3日

1、关联规则简介

- 数据挖掘是一项从大量的记录数据中提取有价值的、人们感兴趣的知识，这些知识是隐含的、事先未知的有用信息，提取的知识一般可表示为概念(Concepts)、规则(Rules)、规律(Regular ides)、模式(Patterns)等形式。

- 关联规则是当前数据挖掘研究的主要方法之一，它反映一个事物与其他事物之间的相互依存性和关联性。如果两个或者多个事物之间存在一定的关联关系，那么，其中一个事物就能够通过其他事物预测到。

- 典型的关联规则发现问题是对超市中的货篮数据（Market Basket）进行分析。通过发现顾客放入货篮中的不同商品之间的关系来分析顾客的购买习惯。

关联规则：发现数据中的规律

- 超市中什么产品会一起购买？(组合推荐)
- 顾客在买了一台PC之后下一步会购买?(搭配推荐)
- 哪种DNA对这种药物敏感?
- 我们如何自动对Web文档进行分类?
- 论文查重?

2、关联规则的基本概念

- 设$I=\{i_1, i_2,\ldots, i_m\}$为所有项目的集合，$D$为事务数据库，事务$T$是一个项目子集（$T\subseteq I$）。每一个事务具有唯一的事务标识$TID$。设$A$是一个由项目构成的集合，称为**项集**。事务$T$包含项集$A$，当且仅当$A\subseteq T$。如果项集$A$中包含$k$个项目，则称其为**k项集**。
- 项集A在事务数据库D中出现的次数占D中总事务的百分比叫做项集的**支持度**。如果项集的支持度超过用户给定的最小支持度阈值，就称该项集是**频繁项集**（或频集）。

关联规则的基本概念

- 关联规则是形如$X \Rightarrow Y$的逻辑蕴含式，其中$X \subset I$，$Y \subset I$，且$X \cap Y = \varnothing$
- 如果事务数据库D中有$s\%$的事务包含$X \cup Y$，则称关联规则$X \Rightarrow Y$的支持度为$s\%$
- 关联规则的信任度为$support\ (X \cup Y) / support\ (X)$

也就是：

$$support\ (X \Rightarrow Y) = P\ (X \cup Y)$$

$$confidence\ (X \Rightarrow Y) = P\ (Y \mid X)$$

强关联规则

- 强关联规则就是支持度和信任度分别满足用户给定阈值的规则。

例：

交易ID	购买的商品
2000	A,B,C
1000	A,C
4000	A,D
5000	B,E,F

设最小支持度为50%, 最小可信度为 50%, 则可得到

- $A \Rightarrow C$ (50%, 66.6%)
- $C \Rightarrow A$ (50%, 100%)

3、关联规则挖掘算法

- Agrawal等人提出的AIS，Apriori和AprioriTid
- Cumulate和Stratify，Houstsma等人提出的SETM
- Park等人提出的DHP
- Savasere等人的PARTITION
- Han等人提出的不生成候选集直接生成频繁模式FPGrowth
- 其中最有效和有影响的算法为Apriori，DHP和PARTITION，FPGrowth。

Apriori算法

- Apriori算法命名源于算法使用了频繁项集性质的先验（Prior）知识。
- Apriori算法将发现关联规则的过程分为两个步骤：
  - 通过迭代，检索出事务数据库中的所有频繁项集，即支持度不低于用户设定的阈值的项集；
  - 利用频繁项集构造出满足用户最小信任度的规则。
- 挖掘或识别出所有频繁项集是该算法的核心，占整个计算量的大部分。

Apriori的性质：

- 2条性质
  - 性质1：频繁项集的所有非空子集必为频繁项集。
  - 性质2：非频繁项集的超集一定是非频繁的。

Apriori的步骤：

- 连接步：为找L_k，通过将L_{k-1}与自身连接产生候选k项集的集合

- 剪枝步：C_k是L_k的超集，也就是说，C_k的成员可以是也可以不是频繁的，但所有的频繁k项集都包含在C_k中。任何非频繁的（k-1）项集都不是频繁k项集的子集。

Apriori算法

L_1={频繁1项集};
for(k=2;$L_{k-1}\neq\varnothing$;k++) do begin
    C_k=apriori_gen(L_{k-1}); //新的候选频繁项集
    for all transactions t∈D do begin //扫描计数
        C_t=subset(C_k,t); //得到t的子集，它们是候选
        for all candidates c∈C_t do
            c.count++;
        end;
    L_k={c∈C_k|c.count≥minsup}
end;

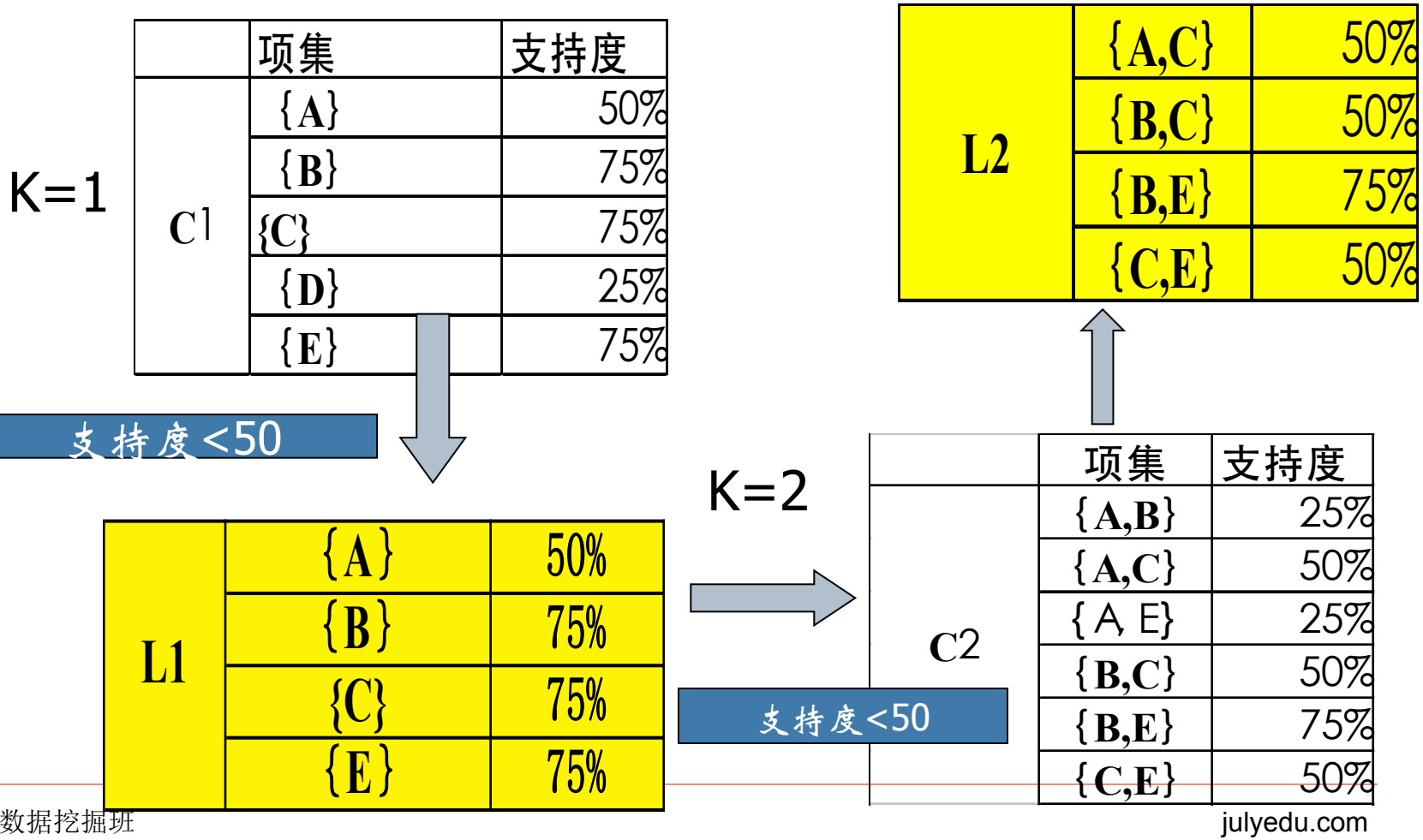
Answer= $\bigcup_k L_k$

Apriori算法实例

现有A、B、C、D、E五种商品的交易记录表，试找出三种商品关联销售情况(k=3)，最小支持度=50%。

交易号	商品代码
100	A、C、D
200	B、C、E
300	A、B、C、E
400	B、E

实例解答


K=1
项集
支持度
C1
{A} 50%
{B} 75%
{C} 75%
{D} 25%
{E} 75%
支持度<50
L1
{A} 50%
{B} 75%
{C} 75%
{E} 75%
K=2
项集
支持度
C2
{A,B} 25%
{A,C} 50%
{A,E} 25%
{B,C} 50%
{B,E} 75%
{C,E} 50%
支持度<50
L2
{A,C} 50%
{B,C} 50%
{B,E} 75%
{C,E} 50%

L2	{A,C}	50%
	{B,C}	50%
	{B,E}	75%
	{C,E}	50%

L3	{B, C, E}	50%

从K2中求可用来计算的的三项集	
{A,C} +{B,C}	{A,B,C}
{A,C} +{B,E}	超过三项
{A,C} +{C,E}	{A,C，E}
{B,C} +{B,E}	{B,C，E}
{B,C} +{C,E}	{B,C，E}
{B,E} +{C,E}	{B,C，E}

C3	{A, B, C}	25%
	{A, C, E}	25%
	支持度<50	
	{B, C, E}	50%

Apriori算法的不足

- C_k中的项集是用来产生频集的候选集.
- 最后的频集L_k必须是C_k的一个子集。
  - C_k中的每个元素需在交易数据库中进行验证来决定其是否加入L_k
- 验证过程是性能瓶颈
  - 交易数据库可能非常大
  - 比如频集最多包含10个项，那么就需要扫描交易数据库10遍
  - 需要很大的I/O负载。

数据与案例

见数据与对应python脚本

Hadoop/Spark实现

- Hadoop的实现过程可以看这里
- 似乎并没有对应的官方spark实现
- 需要实现过程可以参考这里

4、FP-tree算法（不用生成候选集）

- 2000年，Han等提出了一个称为FP-tree的算法。
  FP-tree算法只进行2次数据库扫描。
  - no候选集
  - 直接压缩数据库成一个频繁模式树
  - 通过这棵树生成关联规则。

- FP-tree两个主要步骤

  ①利用事务数据库中的数据构造FP-tree；

  ②从FP-tree中挖掘频繁模式。

步骤1：构造 FP-tree树

具体过程：

1. 扫描数据库一次，得到频繁1-项集
2. 把项按支持度递减排序
3. 再一次扫描数据库，建立FP-tree

FP-tree 结构的好处

- 完备
  - 不会打破交易中的任何模式
  - 包含了频繁模式挖掘所需的全部信息
- 紧密
  - 去除不相关信息—不包含非频繁项
  - 支持度降序排列: 支持度高的项在FP-tree中共享的机会也高
  - 决不会比原数据库大（如果不计算树节点的额外开销)

步骤2：频繁模式的挖掘

具体过程:

根据事务数据库D 和最小支持度min_sup， 调用建树过程建立FP-tree;

if (FP-tree 为简单路径)
    将路径上支持度计数大于等于min_sup 的节点任意组合，得到所需的频繁模式
else
    初始化最大频繁模式集合为空

按照支持频率升序，以每个1 – 频繁项为后缀，调用挖掘算法挖掘最大频繁模式集

根据最大频繁模式集合中最大频繁模式，输出全部的频繁模式.

FP-tree算法的一个例子

事物数据库 ：

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

第一步、构造FP-tree

- 扫描事务数据库得到频繁1-项目集F

I1	I2	I3	I4	I5
6	7	6	2	2

- 定义minsup=20%，即最小支持度为2
- 重新排列F，把项按支持度递减排序：

I2	I1	I3	I4	I5
7	6	6	2	2

重新调整事务数据库

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

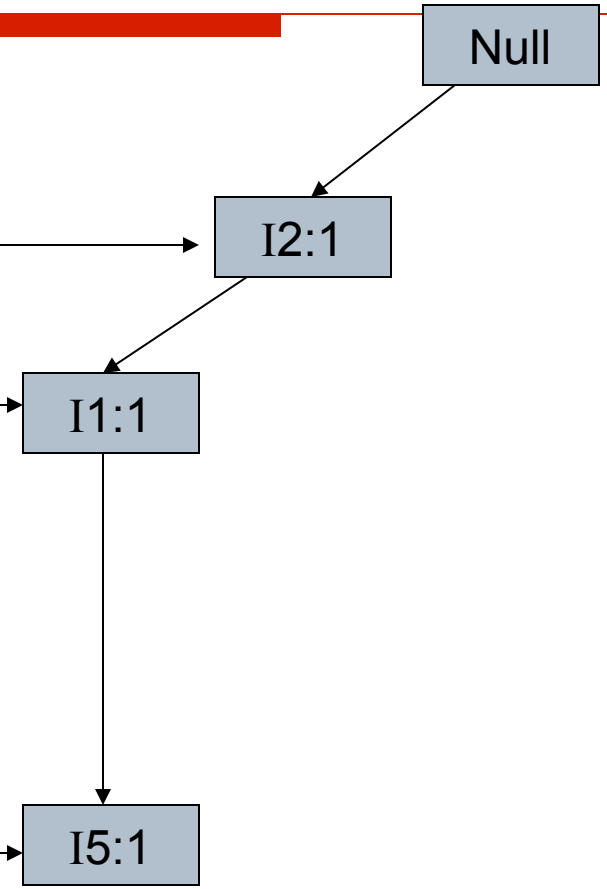
创建根结点和频繁项目表

Item-name	Node-head
I2	Null
I1	Null
I3	Null
I4	Null
I5	Null

Null

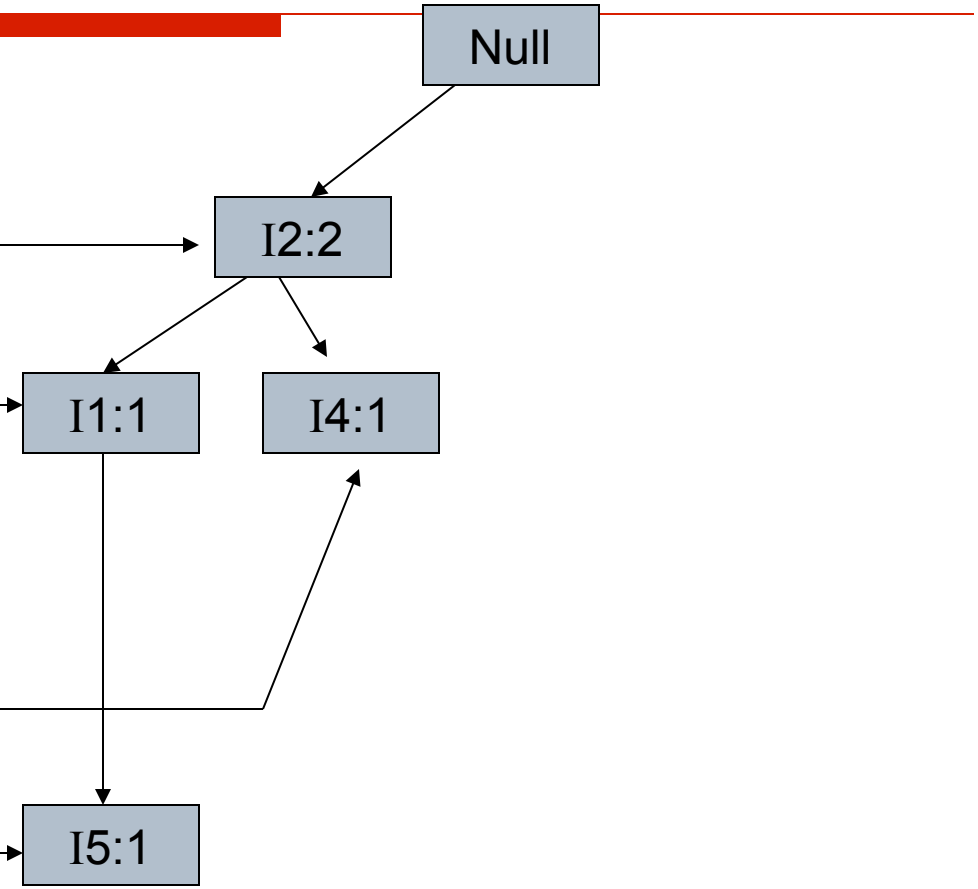
加入第一个事务(I2, I1, I5)

Item-name	Node-head
I2	
I1	
I3	Null
I4	Null
I5	



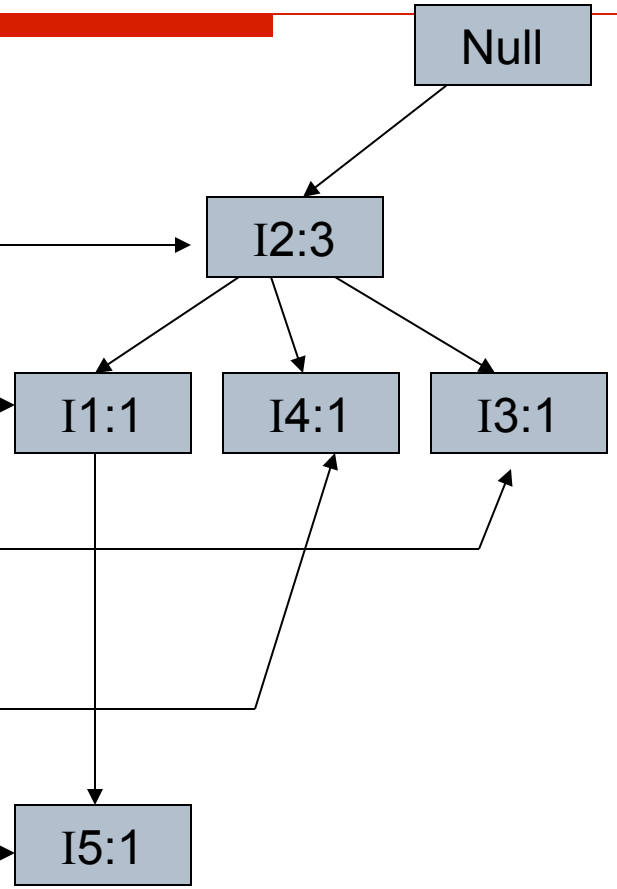
加入第二个事务(I2, I4)

Item-name	Node-head
I2	
I1	
I3	Null
I4	
I5	



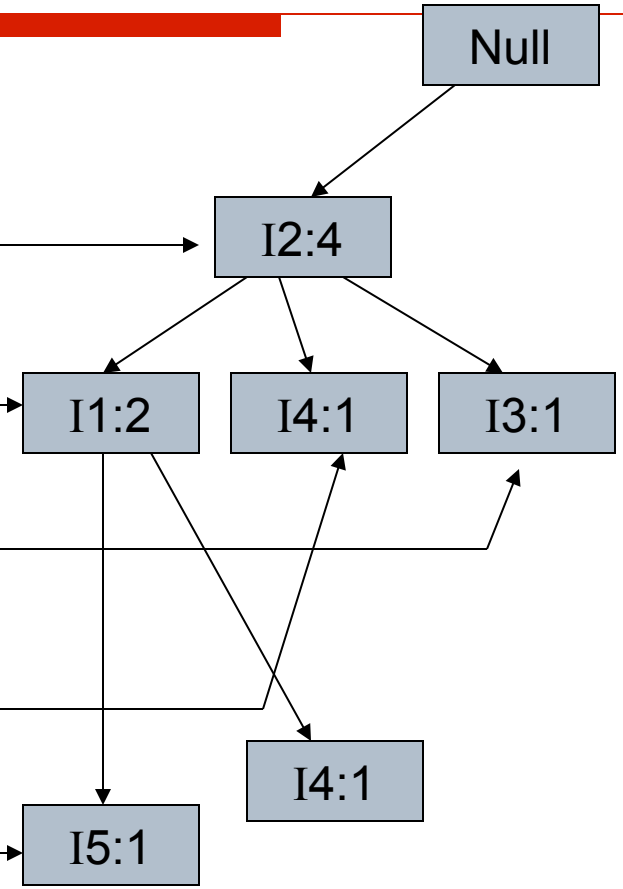
加入第三个事务(I2, I3)

Item-name	Node-head
I2	
I1	
I3	
I4	
I5	



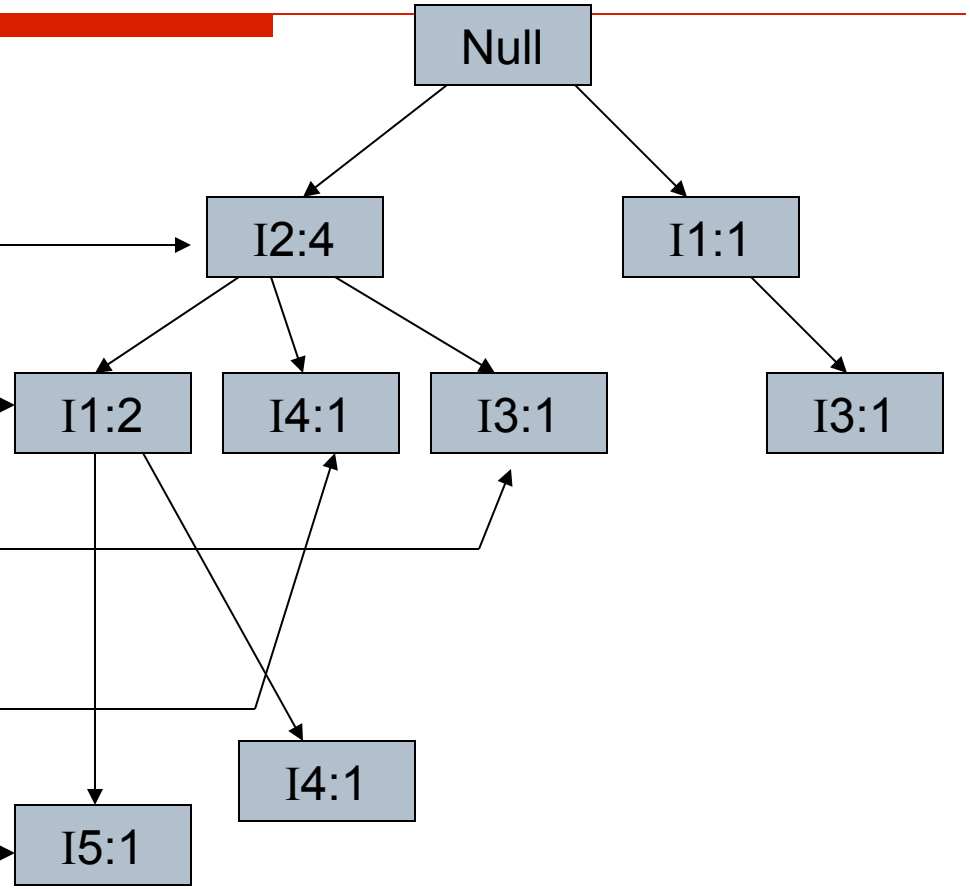
加入第四个事务(I2, I1, I4)

Item-name	Node-head
I2	
I1	
I3	
I4	
I5	



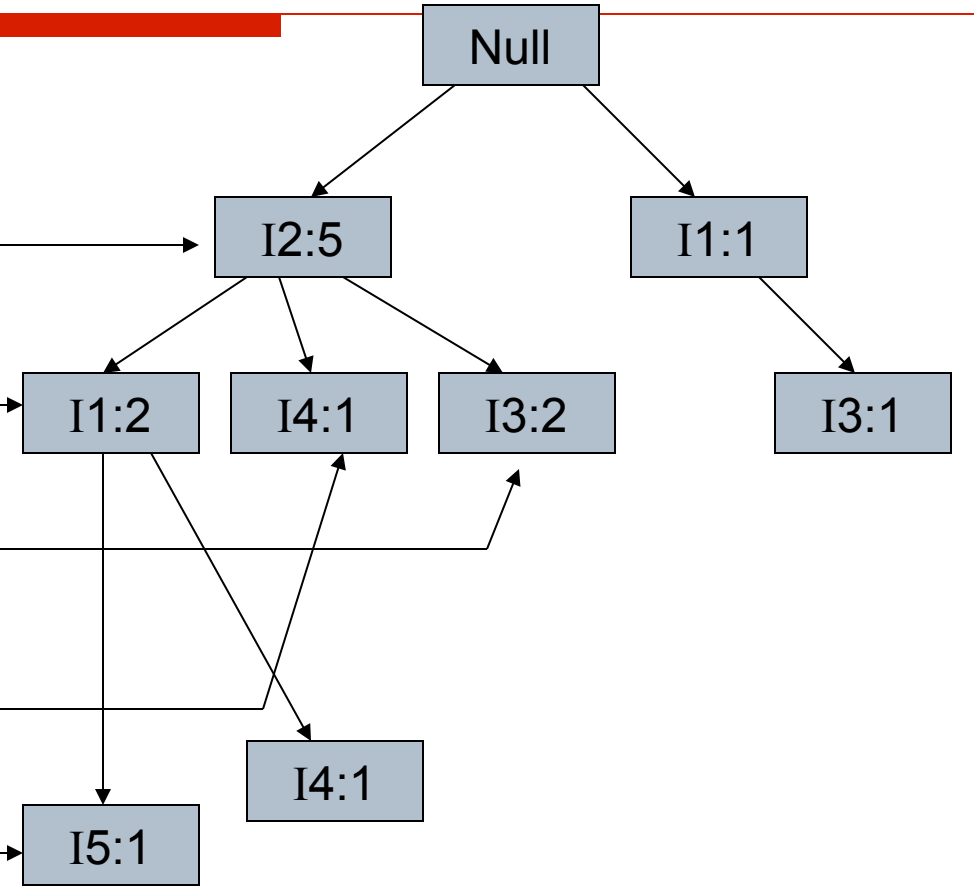
加入第五个事务(I1, I3)

Item-name	Node-head
I2	
I1	
I3	
I4	
I5	



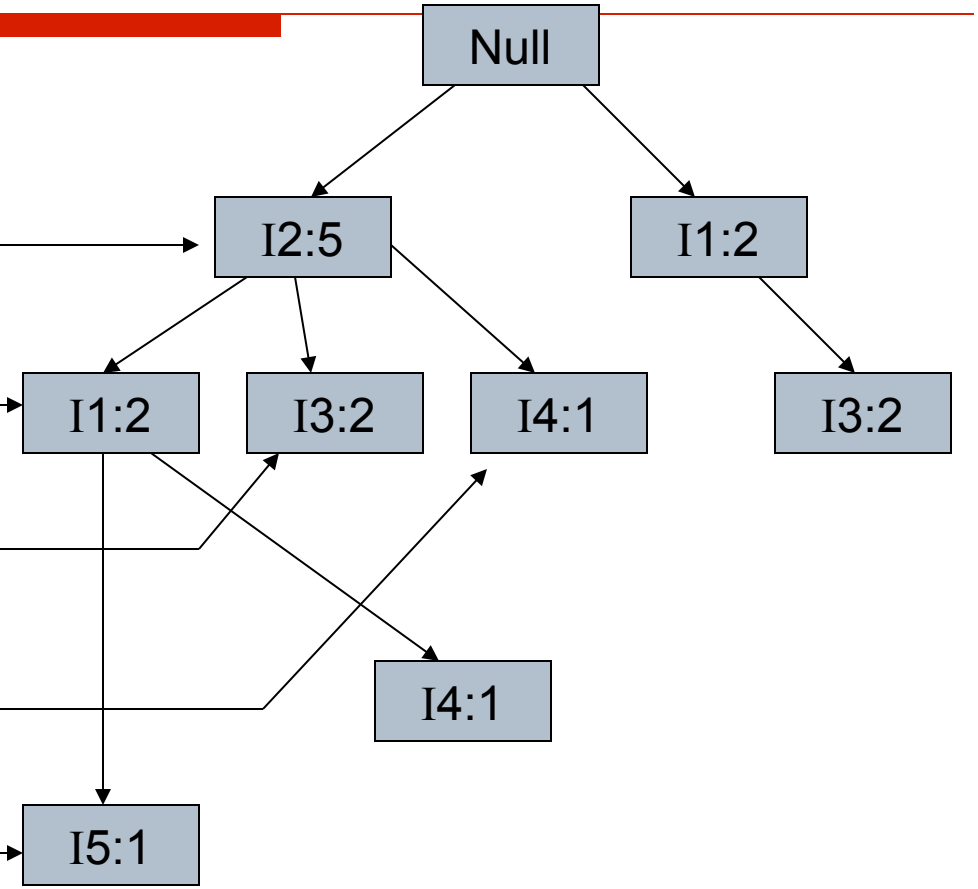
加入第六个事务(I2, I3)

Item-name	Node-head
I2	
I1	
I3	
I4	
I5	



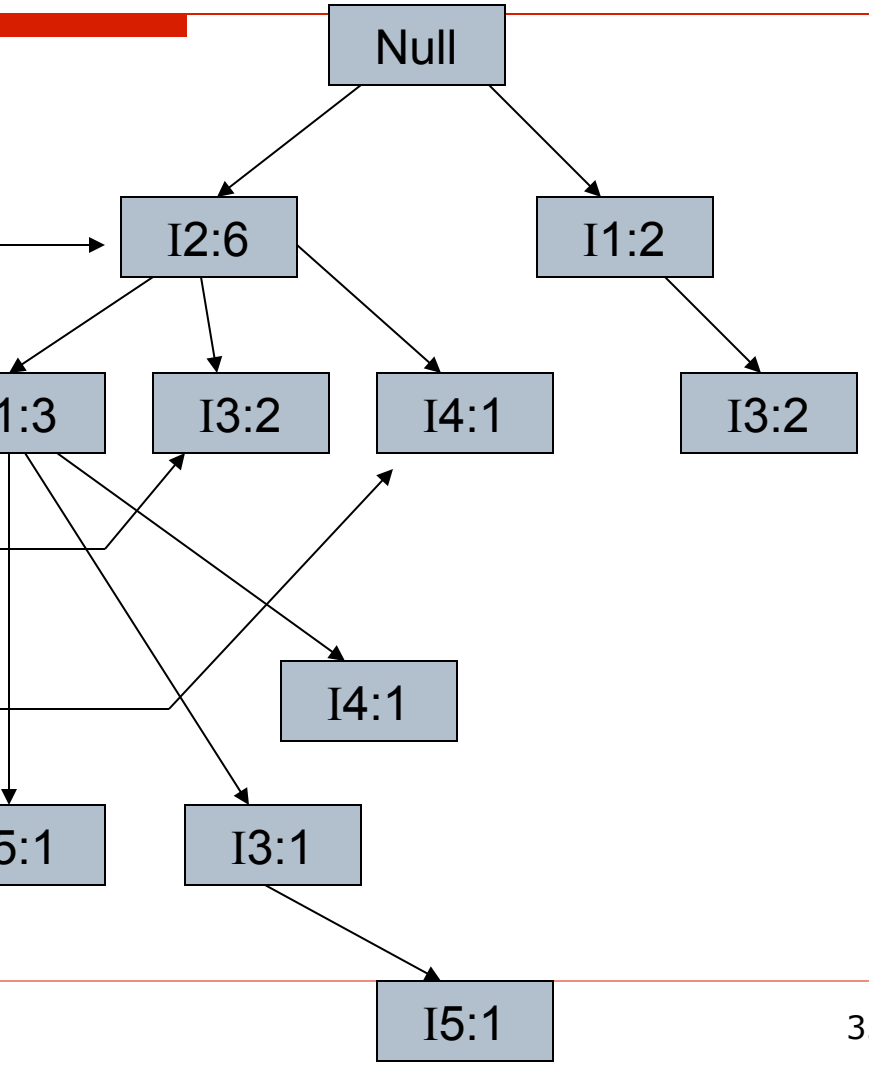
加入第七个事务(I1, I3)

Item-name	Node-head
I2	
I1	
I3	
I4	
I5	



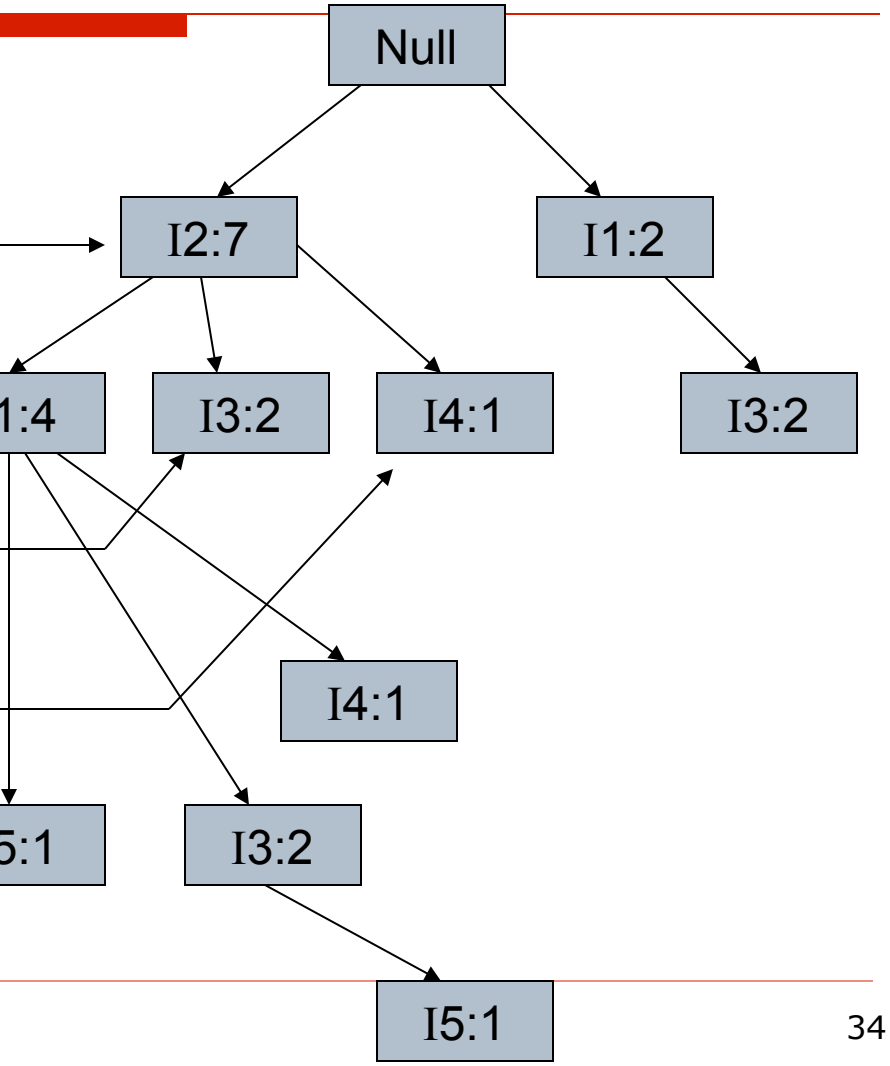
加入第八个事务(I2, I1, I3, I5)

Item-name	Node-head
I2	
I1	
I3	
I4	
I5	



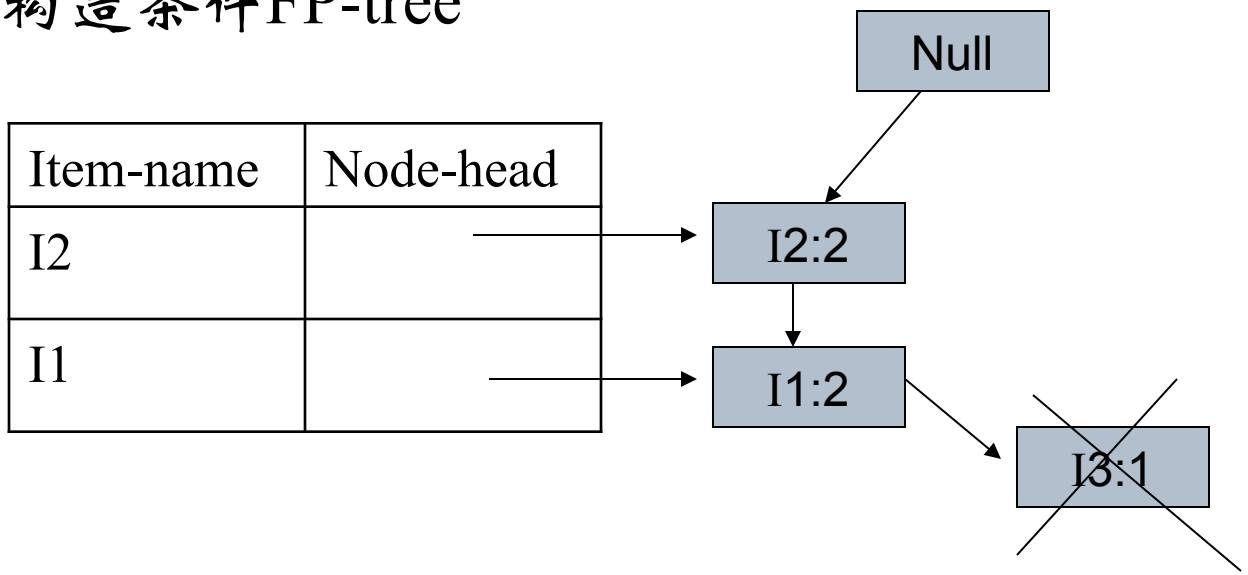
加入第九个事务(I2, I1, I3)

Item-name	Node-head
I2	
I1	
I3	
I4	
I5	



第二步、FP-growth

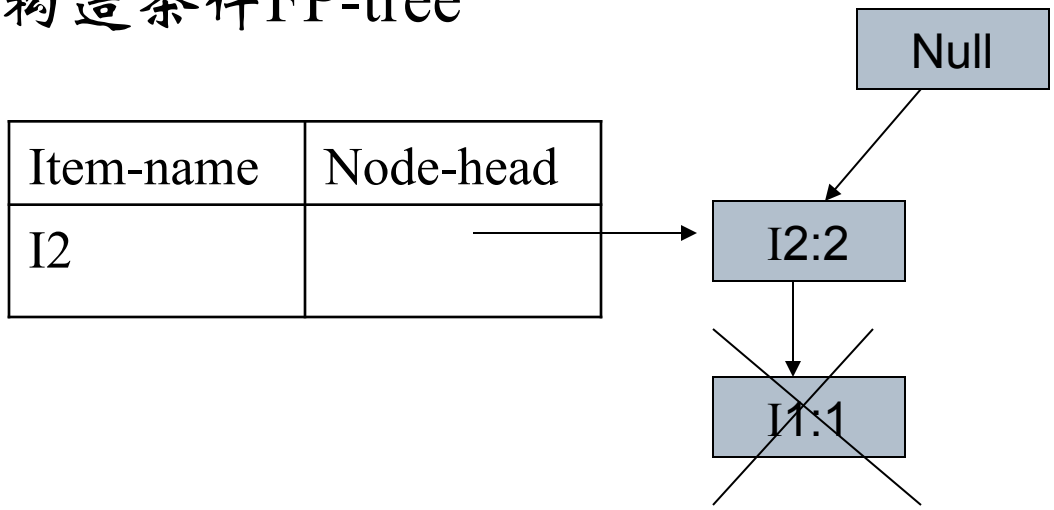
- 首先考虑I5，得到条件模式基
- <(I2,I1:1)>、<I2,I1,I3:1>
- 构造条件FP-tree



- 得到I5频繁项集：{{I2,I5:2},{I1,I5:2},{I2,I1,I5:2}}

第二步、FP-growth

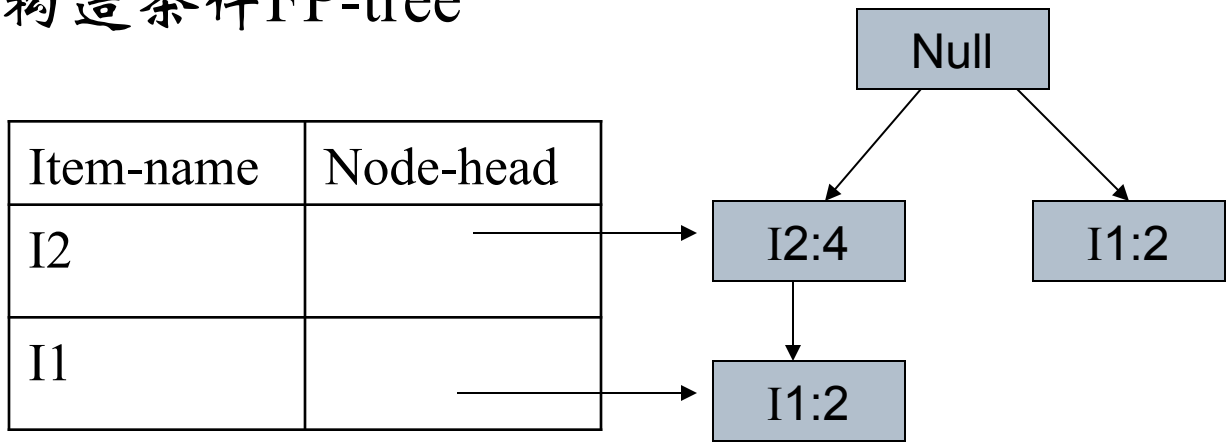
- 接着考虑I4，得到条件模式基
- <(I2,I1:1)>、<I2:1>
- 构造条件FP-tree



- 得到I4频繁项集：{{I2,I4:2}}

第二步、FP-growth

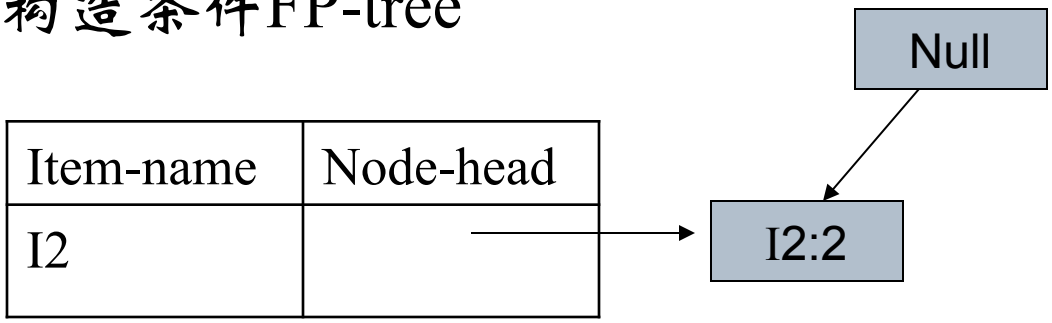
- 然后考虑I3，得到条件模式基
- <(I2,I1:2)>、<I2:2>、 <I1:2>
- 构造条件FP-tree



- 由于此树不是单一路径，因此需要递归挖掘I3

第二步、FP-growth

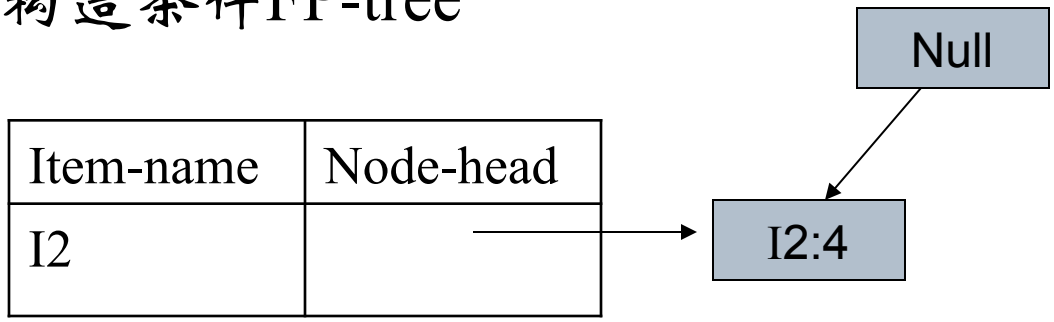
- 递归考虑I3，此时得到I1条件模式基
- <(I2:2)>,即I1I3的条件模式基为<(I2:2)>
- 构造条件FP-tree



- 得到I3的频繁项目集{{I2,I3:4},{I1,I3:4},{I2,I1,I3:2}}

第二步、FP-growth

- 最后考虑I1，得到条件模式基
- <(I2:4)>
- 构造条件FP-tree



- 得到I1的频繁项目集{{I2,I1:4}

FP - tree 算法的优缺点

- 优点
  - FP-tree 算法只需对事务数据库进行二次扫描
  - 避免产生的大量候选集.
- 缺点
  - 要递归生成条件数据库和条件FP-tree，所以内存开销大
  - 只能用于挖掘单维的布尔关联规则.

数据与例子

见数据与对应python脚本

Spark与FP-growth

FP-growth

The FP-growth algorithm is described in the paper Han et al., Mining frequent patterns without candidate generation, where "FP" stands for frequent pattern. Given a dataset of transactions, the first step of FP-growth is to calculate item frequencies and identify frequent items. Different from Apriori-like algorithms designed for the same purpose, the second step of FP-growth uses a suffix tree (FP-tree) structure to encode transactions without generating candidate sets explicitly, which are usually expensive to generate. After the second step, the frequent itemsets can be extracted from the FP-tree. In `spark.mllib`, we implemented a parallel version of FP-growth called PFP, as described in Li et al., PFP: Parallel FP-growth for query recommendation. PFP distributes the work of growing FP-trees based on the suffices of transactions, and hence more scalable than a single-machine implementation. We refer users to the papers for more details.

`spark.mllib`'s FP-growth implementation takes the following (hyper-)parameters:

- `minSupport`: the minimum support for an itemset to be identified as frequent. For example, if an item appears 3 out of 5 transactions, it has a support of 3/5=0.6.
- `numPartitions`: the number of partitions used to distribute the work.

Spark与FP-growth

Examples

Scala Java Python

`FPGrowth` implements the FP-growth algorithm. It take an `RDD` of transactions, where each transaction is an `List` of items of a generic type. Calling `FPGrowth.train` with transactions returns an `FPGrowthModel` that stores the frequent itemsets with their frequencies.

Refer to the `FPGrowth` Python docs for more details on the API.

```
from pyspark.mllib.fpm import FPGrowth

data = sc.textFile("data/mllib/sample_fpgrowth.txt")
transactions = data.map(lambda line: line.strip().split(' '))
model = FPGrowth.train(transactions, minSupport=0.2, numPartitions=10)
result = model.freqItemsets().collect()
for fi in result:
    print(fi)
```

感谢大家！

恳请大家批评指正！