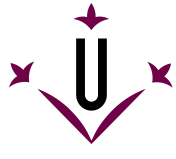
Mining Frequent Itemsets (2)

Ramon Béjar

Data mining - Master on Computer Science



Universitat
de Lleida

Mining Frequent Itemsets

Consider a sequence/set of **transactions**:

$$T = \{T_1, T_2, \ldots, T_m\}$$

where each T_i is a set of items that come from some catalog of possible items I. That is, $\forall T_i, T_i \subseteq I$.

Given a support threshold $\theta \in [0, 1]$, we say that a itemset $P \subseteq I$ is θ—frequent if its support among T is $\geq \theta$:

$$\frac{|\{T_i | P \subseteq T_i, T_i \in T\}|}{|T|} \geq \theta$$

The FP-Growth Algorithm

We have discussed in the A-Priori frequent itemsets mining algorithm that a major bottleneck in the performance is the possible increasing number of candidate itemsets as the size k considered increases in every stage of the algorithm.

We present next an algorithm, the FP-Growth, that avoids this candidate set generation of itemsets of size k (C_k) to be able to generate the actual set of frequent itemsets of size k (L_k). The algorithm achieves this improvement thanks to a data structure called a **prefix-search tree**. This tree is a *compact* representation of the set of transactions, but in such a way that allows a traversal of its branches to efficiently find the frequent itemsets of any size k scanning the tree only once.

Building a prefix-search-tree with FP-Growth

Consider the following set of transactions:

In [2]:
```
T = [['f','c','a','m','p','i'],
     ['f','b','a','c','m','o'],
     ['b','f','j'],
     ['c','b','p','s'],
     ['a','c','f','m','p','l']]
```

where the corresponding set L_1 with $\theta = 3/5$ is
$\{f:4, c:4, a:3, b:3, m:3, p:3\}$. The prefix-search tree for T represents in a compact way the five transactions, such that then we can extract any frequent subset with a recursive traversal of the tree.

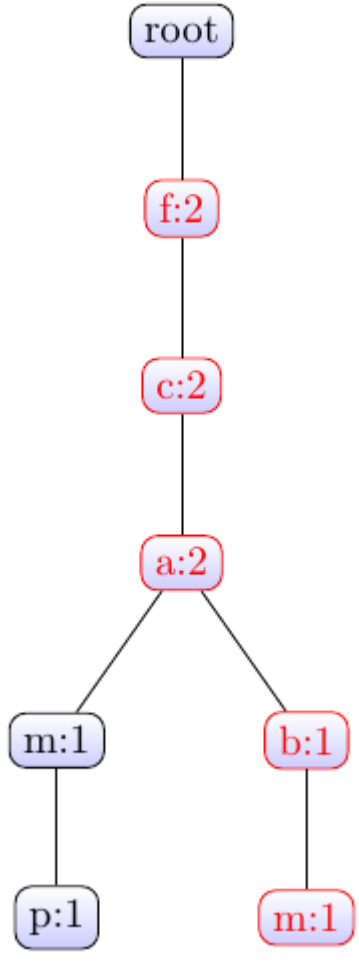
First, as we did with the A-Priori algorithm, we eliminate from every transaction elements not in L_1, but we also sort the remaining elements by descending number of ocurrences in $T\,(L_1)$:

In [3]:
```
TfilAndSorted = [['f','c','a','m','p'],
                 ['f','c','a','b','m'],
                 ['f','b'],
                 ['c','b','p'],
                 ['f','c','a','m','p']]
```

First step: insert $\{f, c, a, m, p\}$

- Starting from a tree with only the root node, we insert each element of $\{f, c, a, m,$ respecting the order of the elements.
- The number that labels each node represents the number of processed transaction the DB that contain the subset given by the path of items from the root to the node there is:
    - one subset $\{f\}$
    - one subset $\{f, c\}$
    - one subset $\{f, c, a\}$
    - one subset $\{f, c, a, m\}$
    - one subset $\{f, c, a, m, p\}$

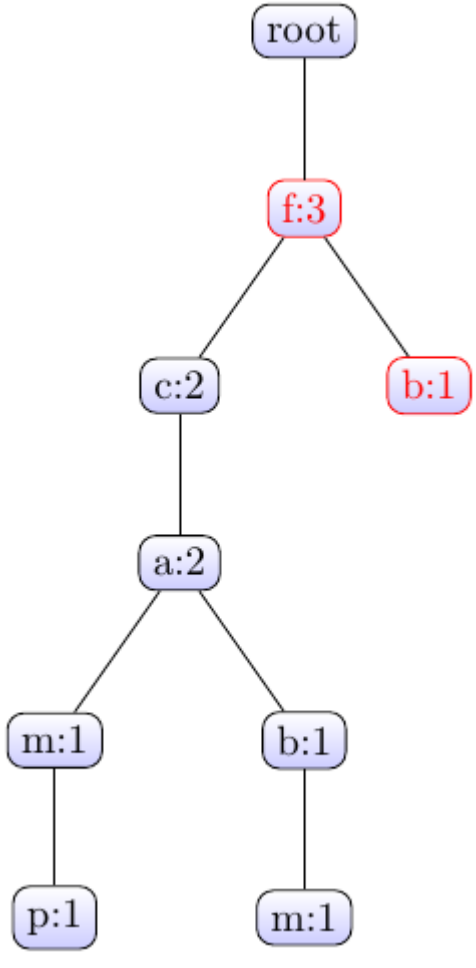
Second step: insert $\{f, c, a, b, m\}$



For the second transaction, the first three elements (f, c, a) are found in the same order starting from the root, so we simply increment their counters But when we arrive to the

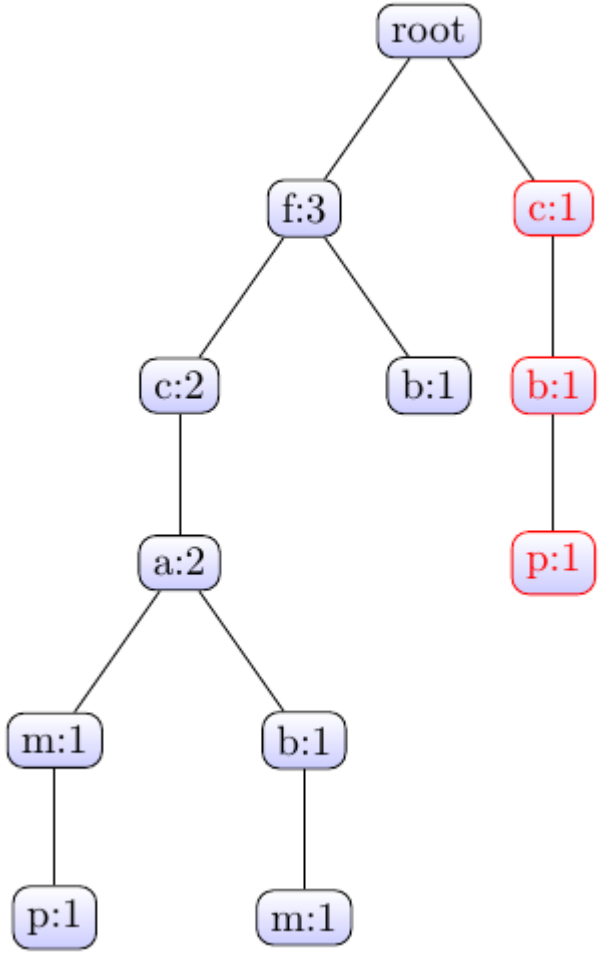
element b, we need to make a new sub-branch (from the element a) with initial counters fo
and m equal to 1

Third step: insert $\{f, b\}$



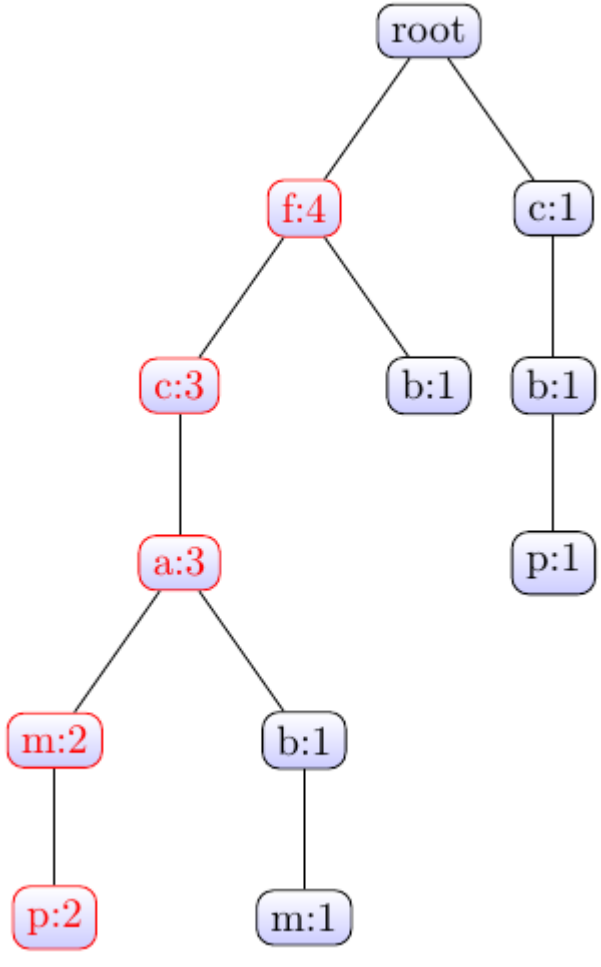
For the third transation, we increment the counter for f And make a new sub-branch from b, with initial counter 1

Fourth step: insert $\{c, b, p\}$



The next transaction, starts with an element, c, that is not f, so we make a new branch fro
root of the tree with initial counters equal to 1.

Fith step: insert $\{f, c, a, m, p\}$



Finally, when we insert the last transaction, $\{f, c, a, m, p\}$, because it is already represen the first branch of the tree, processing their elements by order simply causes to increment counters of that branch

So, the prefix-search tree represents in a compact way:

- the four transactions that contain the most frequent element (f), with the subtree with root f. Observe that this subtree contains three subbranches (that share some nodes), where the branch f-c-a-m-p encodes the information of the first and last transactions of the DB, the branch f-c-a-b-m encodes the second transaction, and the branch f-b the third transaction.
- Finally, the branch c-b-p represents the fourth transaction.

Given a DB of transactions and the ordered set L1 computed with a particular minSupport, the following algorithm (pseudo-code) can be used to build such prefix-search tree

```
def buildFPtree( L1, DB ):
   # create initial tree with only a root
   fptree = (root,())
   L1 = sortByFrequency(L1)
   for tran in DB:
      sortedtran = sortByFrequency(tran,L1)
      insertTree(sortedtran,fptree)
   return fptree
```

```
def insertTree( sortedtran, fptree ):
   firstit = head(sortedtran)
   if (fptree has child node with name firstit):
       fpsubtree = getsubtree( fptree, firstit )
       incrementRootCounter( fpsubtree )
   else:
      fpsubtree = ((firstit,1),())
      addchild(fptree,fpsubtree)
   tailtran = sortedtran.delete(firstit)
   if (tailtran not empty) insertTree( tailtran, fpsubtree )
```

Observe that the overall process for building the prefix-search tree needs two scans of the transaction DB, one for building the ordered set L_1, and a second one for building the tree from L_1 and the DB of transactions, where each transaction is processed once.

Once we have such tree, it is possible to compute the **full set of frequent subsets** with the given minSupport, from our DB of transactions with a recursive algorithm that traverses the tree recursively. The details of such algorithm (and of the previous prefix-search tree construction) can be found in the following paper:

> *Jiawei Han, Jian Pei, Yiwen Yin, Runying Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. Data Mining Knowledge Discovery. 8(1): 53-87 (2004). Link: http://hanj.cs.illinois.edu/pdf/dami04_fptree.pdf (http://hanj.cs.illinois.edu/pdf/dami04_fptree.pdf)*

A distributed FPgrowth

Spark includes a distributed version of FPgrowth, that works by creating different independent prefix-search trees in different partitions. The tree of each partition is used to count different patterns (**ordered subsets**), such that a given pattern will be responsability of the tree of one specific partition. This algorithm is described in the paper:

> *Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, Edward Y. Chang. PFP: parallel FP-growth for query recommendation. RecSys 2008: 107-114. Link: http://dl.acm.org/citation.cfm?doid=1454008.1454027 (http://dl.acm.org/citation.cfm?doid=1454008.1454027)*

They key of the algorithm is how it divides the possible patterns to count in the different prefix-search trees.

Dividing patterns by element groups

Assume we order elements by some given fixed order (in our case by their position in L_1). Then, if we divide the elements of L_1 in m different groups, the distributed version of FP-growth assigns a pattern of ordered elements:

$$[e_1, e_2, \ldots, e_k]$$

to the group where e_k (the last element) belongs. So, any pattern will be **uniquely assigned to one group**.

Then, given a transaction that may have elements from different groups: $t = [t_1, t_2, \ldots, t_h]$, the algorithm creates different **group-dependent transactions** from t as follows:

```
def makeGroupDependentTransactions(t,groupId):
    groupOft = dict()
    for i = h-1 to 0:
        if (groupId(t[i]) not in groupOft):
           groupOft[ groupId(t[i]) ] = t[0:i]
```

So, an input transaction t will generate $n \geq 1$ different transactions (each one with a different slice from t) that will be assigned to n different groups (partitions in our distributed map-reduce algorithm). Observe that in any case $n \leq m$ (a transaction is sliced at most the number of groups we have).

To understand better how this function makes the different group partitions, consider the following set of six transactions already sorted by the set $L_1 = [c, a, m, f, p]$:

$$t_1 = \{c, a, m, f, p\}$$
$$t_2 = \{c, a, m\}$$
$$t_3 = \{c, a, f\}$$
$$t_4 = \{c, a, m, p\}$$
$$t_5 = \{c, f\}$$
$$t_6 = \{a, m\}$$

and assume we have divided elements in two groups: group 1: {f,c} and group 2: {a,m,p}

Then, these are the different group dependent transactions generated for each group:

input transaction	group {f,c}	group {a,m,p}
$\{c, a, m, f, p\}$	$\{c, a, m, f\}$	$\{c, a, m, f, p\}$
$\{c, a, m\}$	$\{c\}$	$\{c, a, m\}$
$\{c, a, f\}$	$\{c, a, f\}$	$\{c, a\}$
$\{c, a, m, p\}$	$\{c\}$	$\{c, a, m, p\}$
$\{c, f\}$	$\{c, f\}$	
$\{a, m\}$		$\{a, m\}$

Two remarks:

- group dependent transactions with only one element can be ignored, as counting singleton sets is already done in L_1. So, we can consider that the group dependent transactions are: $$ \begin{array}{ll}

```
\textrm{group(fc)} = & \{ \{c,a,m,f \}, \{ c,a,f \}, \{ c,f\} \} \\
\textrm{group(amp)} = & \{ \{ c,a,m,f,p \}, \{ c,a,m \}, \{ c,a \}, \{c,
a,m,p \},\{ a,m \} \}
```

  \end{array} $$
- There is redundant information between both groups of transactions, but the search-tree of a group will be used to count **only patterns** that end with an element of the group. The way of assigning group dependent transactions to groups ensures that the number of transactions that contain a pattern ending with element 'e' will be preserved in the group of transactions associated with 'e'

FP-growth in RDD library

Let's see how we use this algorithm in spark when looking for all the frequent subsets of the following sample set of transactions:

In [4]:
```
beeranddiapers1 = [['beer','diapers','cheese'],
                   ['beer','diapers','pizza'],
                   ['beer','diapers','pizza','cheese'],
                   ['beer','diapers','milk'],
                   ['beer','diapers','milk','pizza'],
                   ['beer','diapers','toothpaste'],
                   ['beer','diapers','icecream'],
                   ['beer','diapers','pizza','yogurt']]
```

In [5]:
```
from pyspark.mllib.fpm import FPGrowth as rddFPGrowth

# make a RDD with a list of lists
transactions = sc.parallelize( beeranddiapers1 )
model = rddFPGrowth.train(transactions, minSupport=0.25, numPartitions=4)
```

In [6]:
```
# Collect back the set of ALL frequent itemsets with minSupport
# Beware !, this set could be exponentially large
rddfreqitemsets = model.freqItemsets()
result = rddfreqitemsets.collect()
for fi in result:
    print( fi )
```

```
FreqItemset(items=['cheese'], freq=2)
FreqItemset(items=['cheese', 'diapers'], freq=2)
FreqItemset(items=['cheese', 'diapers', 'beer'], freq=2)
FreqItemset(items=['cheese', 'beer'], freq=2)
FreqItemset(items=['beer'], freq=8)
FreqItemset(items=['diapers'], freq=8)
FreqItemset(items=['diapers', 'beer'], freq=8)
FreqItemset(items=['pizza'], freq=4)
FreqItemset(items=['pizza', 'diapers'], freq=4)
FreqItemset(items=['pizza', 'diapers', 'beer'], freq=4)
FreqItemset(items=['pizza', 'beer'], freq=4)
FreqItemset(items=['milk'], freq=2)
FreqItemset(items=['milk', 'diapers'], freq=2)
FreqItemset(items=['milk', 'diapers', 'beer'], freq=2)
FreqItemset(items=['milk', 'beer'], freq=2)
```

However, because model.freqItemsets() is a RDD, we can instead save them to a data file, or filter only some of them to be analyzed by our program back in the driver application:

In [7]:
```
pairs = rddfreqitemsets.filter( lambda x  : len(x.items) == 2  ).collect()

print ( "Frequent pairs: " )
for fi in pairs:
    print( fi )

print ( "\nFrequent triples: " )
triples = rddfreqitemsets.filter( lambda x  : len(x.items) == 3  ).collect()
for fi in triples:
    print ( fi )
```

```
Frequent pairs:
FreqItemset(items=['cheese', 'diapers'], freq=2)
FreqItemset(items=['cheese', 'beer'], freq=2)
FreqItemset(items=['diapers', 'beer'], freq=8)
FreqItemset(items=['pizza', 'diapers'], freq=4)
FreqItemset(items=['pizza', 'beer'], freq=4)
FreqItemset(items=['milk', 'diapers'], freq=2)
FreqItemset(items=['milk', 'beer'], freq=2)

Frequent triples:
FreqItemset(items=['cheese', 'diapers', 'beer'], freq=2)
FreqItemset(items=['pizza', 'diapers', 'beer'], freq=4)
FreqItemset(items=['milk', 'diapers', 'beer'], freq=2)
```

FP-growth in dataframes library

This algorithm is also implemented in the ml data frames library of spark:

In [9]:
```
from pyspark.ml.fpm import FPGrowth as dfFPGrowth
sqlCtx = pyspark.sql.SparkSession(sc)
df = sqlCtx.createDataFrame( [ (id,beeranddiapers1[id]) for id in range(8) ], [
"id","values"] )
df.show()
```

```
+---+--------------------+
| id|              values|
+---+--------------------+
|  0|[beer, diapers, c...|
|  1|[beer, diapers, p...|
|  2|[beer, diapers, p...|
|  3|[beer, diapers, m...|
|  4|[beer, diapers, m...|
|  5|[beer, diapers, t...|
|  6|[beer, diapers, i...|
|  7|[beer, diapers, p...|
+---+--------------------+
```

In [12]:
```
fpGrowth = dfFPGrowth(itemsCol="values", minSupport=0.25, numPartitions=4)
model = fpGrowth.fit(df)
model.freqItemsets.show()
```

```
+--------------------+----+
|               items|freq|
+--------------------+----+
|            [cheese]|   2|
|   [cheese, diapers]|   2|
|[cheese, diapers,...|   2|
|      [cheese, beer]|   2|
|              [beer]|   8|
|           [diapers]|   8|
|     [diapers, beer]|   8|
|             [pizza]|   4|
|    [pizza, diapers]|   4|
|[pizza, diapers, ...|   4|
|       [pizza, beer]|   4|
|              [milk]|   2|
|     [milk, diapers]|   2|
|[milk, diapers, b...|   2|
|        [milk, beer]|   2|
+--------------------+----+
```

In [ ]: