part1 Frequent Itemset Mining & Association Rules

关联规则挖掘

Goal： 挖掘频繁被消费者同时购买的商品集合

Input:

Basket	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Output:

Rules Discovered:
{Milk} --> {Coke}
{Diaper, Milk} --> {Beer}

Frequent Itemsets

Goal : 发现商品basket中的frequent items

定义 Support of itme I (支持度)：几个关联的数据在数据集中出现的次数占总数据集的比重

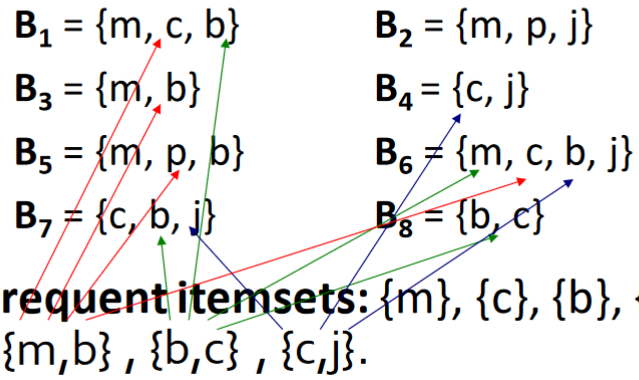
我们将support超过了阈值s的itemsete称之为frequent itemsets

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Support of
{Beer, Bread} = 2

例如

- **Items** = {milk, coke, pepsi, beer, juice}
- **Support threshold** = 3 baskets



- **Frequent itemsets:** {m}, {c}, {b}, {j}, {m,b} , {b,c} , {c,j}.

Association Rules

$\{i_1, i_2, \ldots, i_k\} \rightarrow j$ 表示如果某个basket中已经包括了所有的i_1到i_k个item，那么这个basket很有可能包括i_j

$Confidence\ of\ association\ rules$表示给出basket $I = \{i_1, \ldots, i_k\}$时，含有j的概率

$$conf(I \rightarrow j) = \frac{support(I \bigcup j)}{support(I)}$$

我们并不是关注所有的高概率关联规则，比如$X \rightarrow milk$可能发生的概率很高，但是这仅仅是因为milk会被经常购买，即milk和X是独立的。因此我们需要定义$Interest\ of\ an\ association\ rule I \rightarrow j$ 来平衡item本身被购买的概率和他成为某个basket的frequent item的概率,即找到那些"高价值的Association Rules"。

$$Interest(I \rightarrow j) = |conf(I \rightarrow j) - Pr[j]|$$

B_1 = {m, c, b} **B_2 = {m, p, j}**
B_3 = {m, b} **B_4= {c, j}**
B_5 = {m, p, b} **B_6 = {m, c, b, j}**
B_7 = {c, b, j} **B_8 = {b, c}**

- **Association rule: {m, b} →c**
  - **Support** = 2
  - **Confidence** = 2/4 = 0.5
  - **Interest** = |0.5 – 5/8| = 1/8
    - Item ***c*** appears in 5/8 of the baskets
    - The rule is not very interesting!

Association Rule Mining

我们现在的目标为找到$support >= s$且$confidence >= c$的关联规则

$$\text{conf}(I \to j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}$$

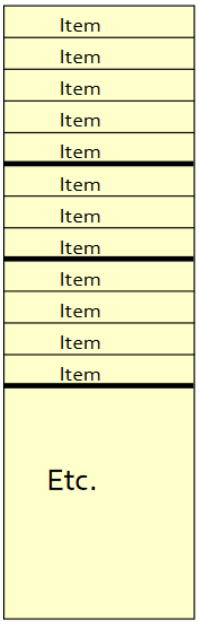
- **Step 1:** Find all frequent itemsets I
  - (we will explain this next)
- **Step 2:** **Rule generation**
  - For every subset A of I, generate a rule $A \to I \setminus A$
    - Since I is frequent, A is also frequent
    - **Variant 1:** Single pass to compute the rule confidence
      - confidence(**A,B→C,D**) = support(**A,B,C,D**) / support(**A,B**)
    - **Variant 2:**
      - **Observation:** If **A,B,C→D** is below confidence, then so is **A,B→C,D**
      - Can generate "bigger" rules from smaller ones!
  - **Output the rules above the confidence threshold**

第一步首先找到所有的频繁项集。第二步生成对应的关联规则。

Finding Frequent Itemsets

如果我们希望挖掘frequent itemsets，那么我们就需要去对每一个item进行count，那么我们就需要对于整个数据集进行遍历。

- **Back to finding frequent itemsets**
- Typically, data is kept in flat files rather than in a database system:
  - Stored on disk
  - Stored basket-by-basket
  - Baskets are **small** but we have many baskets and many items
    - Expand baskets into pairs, triples, etc. as you read baskets
    - Use *k* nested loops to generate all sets of size *k*



Items are positive integers,

通常来说我们的数据集会以basket-by-basket的方式存放在disk中，那么对于这些数据而言，花费时间最多的步骤就变成了I/Os的时间。在实践中，我们以某种顺序依次读取basket中的数据，称之为passes。我们对于某种关联挖掘算法的代价的定义按照passes的数目来衡量。

- For many frequent-itemset algorithms, **main-memory** is the critical resource
  - As we read baskets, we need to count something, e.g., occurrences of pairs of items
  - The number of different things we can count is limited by main memory
  - Swapping counts in/out is a disaster

因为main-memory的限制，所以我们不能以简单遍历的方式对于频繁项集进行挖掘。

Naïve Algorithm

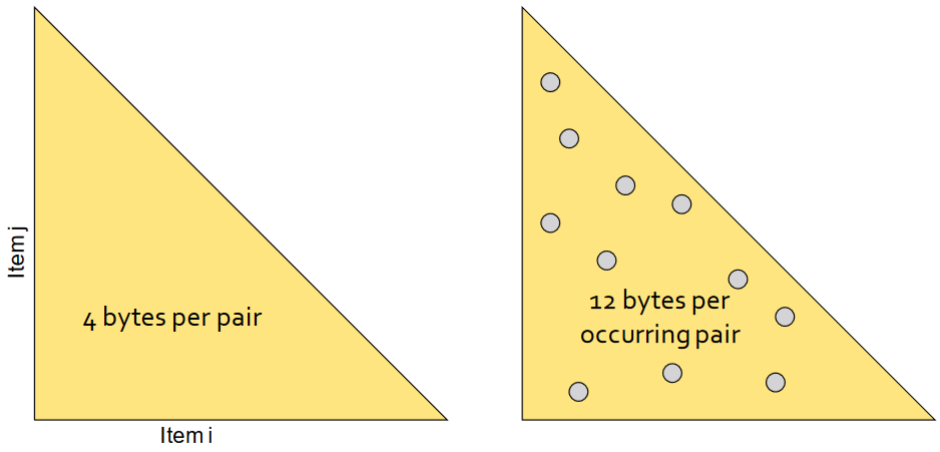
- **Naïve approach to finding frequent pairs**
- Read file once, counting in main memory the occurrences of each pair:
  - From each basket ***b*** of n_b items, generate its $n_b(n_b-1)/2$ pairs by two nested loops
  - A data structure then keeps count of every pair
- **Fails if (#items)^2 exceeds main memory**
  - **Remember:** #items can be 100K (Wal-Mart) or 10B (Web pages)
    - Suppose 10^5 items, counts are 4-byte integers
    - Number of pairs of items: $10^5(10^5-1)/2 \approx 5*10^9$
    - Therefore, $2*10^{10}$ (20 gigabytes) of memory is needed

假设我们对于某个basket寻找他们的frequent pairs，那么我们需要首先枚举所有潜在的frequent pairs可能。通过两层嵌套的循环，我们可以遍历basket来枚举所有可能的pairs。但是在大数据量的情况下，我们需要（$\#items$）^{2}的内存容量来保存循环的结果，而这往往会超出内存限制。

- **Approach 1:** Count all pairs using a matrix
- **Approach 2:** Keep a table of triples [*i*, *j*, *c*] = "the count of the pair of items {*i*, *j*} is *c*."
  - If integers and item ids are 4 bytes, we need approximately 12 bytes for pairs with count > 0
  - Plus some additional overhead for the hashtable

对于上述情况，可以给出两种解决方法。

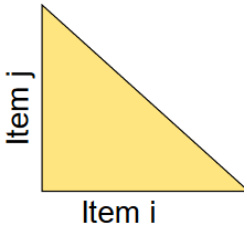
1. 使用矩阵来保存所有可能出现的pairs数目
2. 使用三元元组的形式保存出现的pairs结果，即（item i, item j, 共现次数）



Triangular Matrix　　Triples (item i, item j, count)

Comparing the Two Approaches

- **Approach 1: Triangular Matrix**
  - **n** = total number items
  - Count pair of items {*i*, *j*} only if *i*<*j*
  - Keep pair counts in lexicographic order:
    - {1,2}, {1,3},..., {1,*n*}, {2,3}, {2,4},...,{2,*n*}, {3,4},...
  - Pair {*i*, *j*} is at position: **[n(n - 1) - (n - i)(n - i + 1)]/2 + (j - i)**
  - Total number of pairs ***n*(*n* –1)/2**; total bytes= O(***n*²**)
  - **Triangular Matrix** requires 4 bytes per pair



- **Approach 2** uses **12 *bytes*** per occurring pair *(but only for pairs with count > 0)*
- Approach 2 beats Approach 1 if less than **1/3** of possible pairs actually occur

但是上述数据结构不可以解决当item本身的大小就已经超过了内存的情况。

A-priori Algorithm

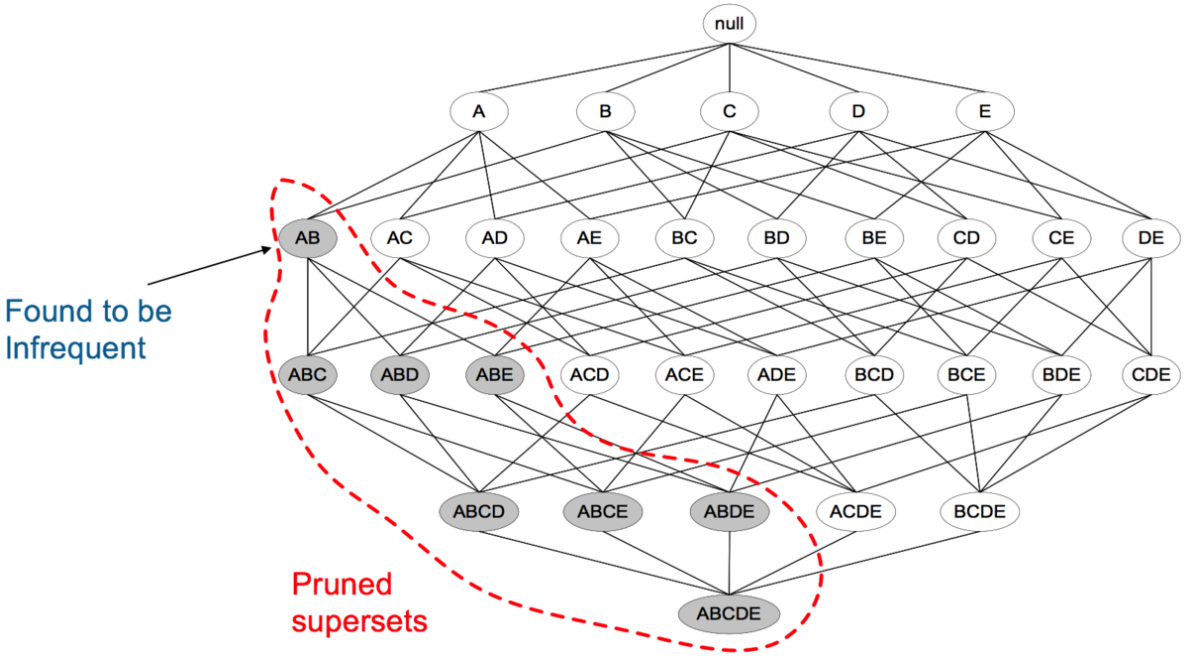
Two Passes Algorithm

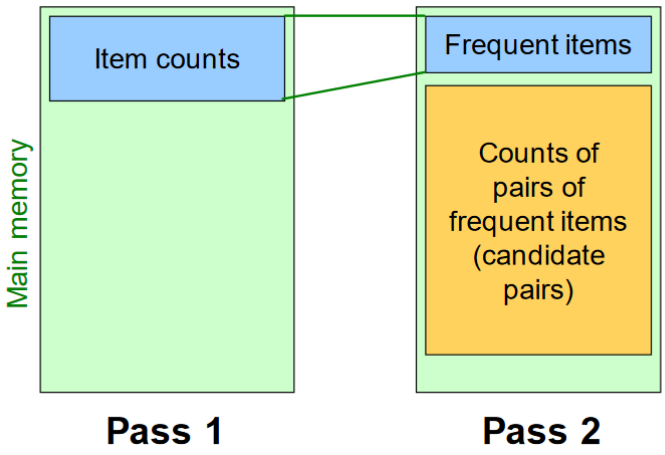
- **Pass 1:** Read baskets and count in main memory the # of occurrences of each **individual item**
  - Requires only memory proportional to #items
- **Items that appear $\geq s$ times are the frequent items**
- **Pass 2:** Read baskets again and keep track of the count of only those pairs where both elements are frequent (from Pass 1)
  - Requires memory (for counts) proportional to square of the number of **frequent** items (not the square of total # of items)
  - Plus a list of the frequent items (so you know what must be counted)

Apriori定律1：如果一个集合是频繁项集，则它的所有子集都是频繁项集。
例如：假设一个集合{A,B}是频繁项集，即A、B同时出现在一条记录的次数大于等于最小支持度min_support，则它的子集{A},{B}出现次数必定大于等于min_support，即它的子集都是频繁项集。
Apriori定律2：如果一个集合不是频繁项集，则它的所有超集都不是频繁项集。
举例：假设集合{A}不是频繁项集，即A出现的次数小于 min_support，则它的任何超集如{A,B}出现的次数必定小于min_support，因此其超集必定也不是频繁项集。

以上两个定律使得我们在计算频繁项集的时候不需要枚举所有的可能性，而只是选择其中的某些子集进行枚举。在Pass1时，我们对于所有的individual items进行计数，此时我们只需要$(\#items)$的内存空间。在pass2时，我们只追踪在pass1中表现为frequent的items并对他们两两配对进行pairs的计数，此时我们不需要$(\#items)$的内存，只需要$(\#frequent\ items)$的内存。

下图表示当我们发现{A,B}是非频繁集时，就代表所有包含它的超集也是非频繁的，即可以将它们都剪除。





C_k: Candidate itemsets of size k

L_k : frequent itemsets of size k

L_1 = {frequent 1-itemsets};

for $(k = 1; L_k \neq \emptyset; k++)$

  C_{k+1} = GenerateCandidates(L_k)

  for each transaction t in database do

    increment count of candidates in C_{k+1} that are contained in t

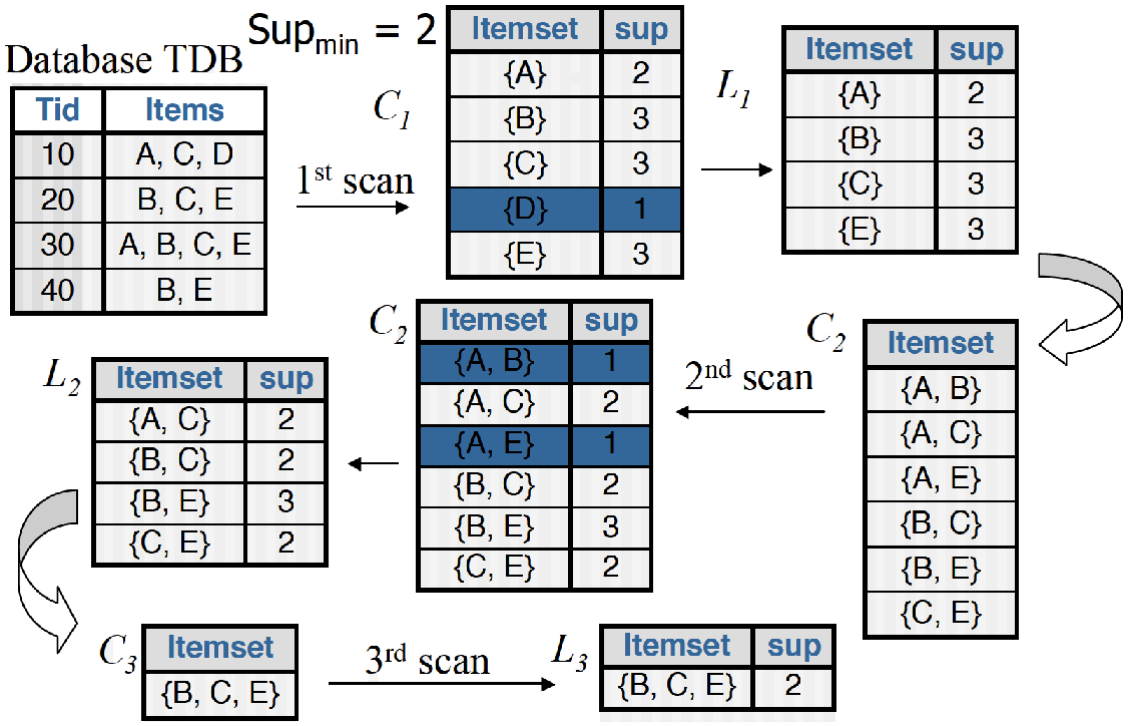
  endfor

  L_{k+1} = candidates in C_{k+1} with support ≥***min_sup***

endfor

return $\bigcup_k L_k$;

最开始数据库里有4条交易，{A、C、D}，{B、C、E}，{A、B、C、E}，{B、E}，使用min_support=2作为支持度阈值，最后我们筛选出来的频繁集为{B、C、E}。



。

- **Hypothetical steps of the A-Priori algorithm**
  - C_1 = { {b} {c} {j} {m} {n} {p} }
  - Count the support of itemsets in C_1
  - Prune non-frequent. We get: L_1 = { b, c, j, m }
  - Generate C_2 = { {b,c} {b,j} {b,m} {c,j} {c,m} {j,m} }
  - Count the support of itemsets in C_2
  - Prune non-frequent. L_2 = { {b,m} {b,c} {c,m} {c,j} }
  - Generate C_3 = { {b,c,m} {b,c,j} {b,m,j} {c,m,j} } **
  - Count the support of itemsets in C_3
  - Prune non-frequent. L_3 = { {b,c,m} }

当上图生成C3的时候，由于$\{b.c.j\}$中的子集$\{b,j\}$没有出现在L2中（其他两个同理），因此我们在生成上述三个集合之后，可以直接删除他们。

PCY Algorithm

我们注意到，在A-priori Algorithm的第一步scanning individual items中，大部分的内存是闲置的。那么我们是否可以通过利用pass1中闲置的内存，来减少pass2中的内存占用呢?

Pass1 for PCY :

```
FOR (each basket) :
    FOR (each item in the basket):
        add 1 to item's count;
New in PCY:
    FOR (each pair of items in the basket):
        hash the pair to a bucket;
        add 1 to the count for that bucket;
```

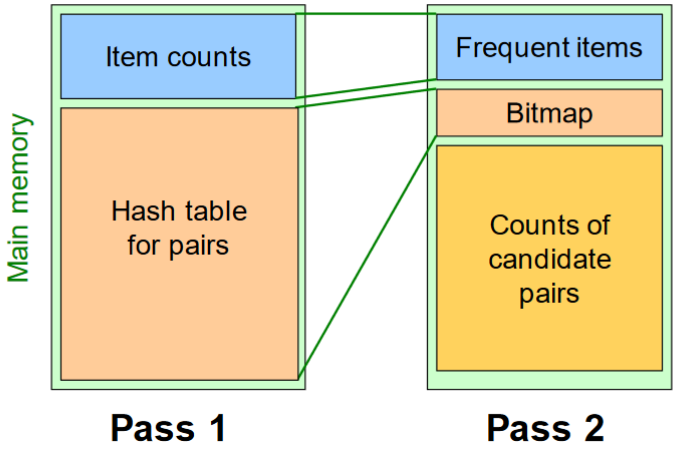
由上图可以看出，PCY除了对于individual items进行计数，还要枚举所有的item pairs。对于枚举的pairs进行哈希映射，映射到bucket上并且对于bucket进行计数。虽然对于哈希映射后的bucket进行计数，bucket counts大于阈值s的item pairs不一定是frequent pairs。但是对于bucket counts小于阈值s的pairs来说，他们一定不是frequent pairs。 因此PCY 在pass1中，使用了闲置的内存，为pass2排除了部分non frequent pairs。因此pass2中，PCY只需要对hash bucket 中大于阈值s的frequent pairs进行计数即可。

- **Replace the buckets by a bit-vector:**
  - **1** means the bucket count exceeded the support s (call it a **frequent bucket**); **0** means it did not
- **4-byte integer counts are replaced by bits, so the bit-vector requires 1/32 of memory**
- Also, decide which items are frequent and list them for the second pass

我们在对于bucket进行计数的时候，还可以使用01向量代替原本的计数，将内存消耗缩小到1/32。

- Count all pairs $\{i, j\}$ that meet the conditions for being a **candidate pair**:
  1. Both i and j are frequent items
  2. The pair $\{i, j\}$ hashes to a bucket whose bit in the bit vector is **1** (i.e., a **frequent bucket**)
- **Both conditions are necessary for the pair to have a chance of being frequent**

pass2中的规则仍然需要满足pairs的子集也是frequent items。



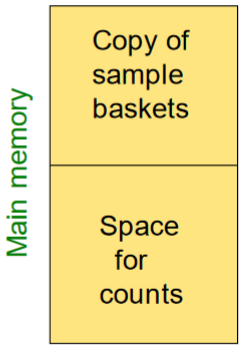
Frequent Itemset In <= 2 Passes

A-priori 和 PCY 都需要K passes来生成大小为K的频繁项集。我们是否可以用忽略掉某些frequent itemsets为代价，在2 Passes以内来寻找到大小为K的频繁项集?

Random Sampling

只是对于whole data的sample运行A priori 算法。因此找到的频繁项集可能有缺失。

- **Take a random sample of the market baskets**
- **Run a-priori or one of its improvements in main memory:**
  - So we don't pay for disk I/O each time we increase the size of itemsets
  - Reduce support threshold proportionally to match the sample size
    - Example: if your sample is 1/100 of the baskets, use *s*/100 as your support threshold instead of *s*.



- **To avoid false positives:** Optionally, verify tha the candidate pairs are truly frequent in the entire data set by a second pass

- **But you don't catch sets that are frequent in the whole data but not in the sample**
  - Smaller threshold, e.g., *s*/125, helps catch more truly frequent itemsets
    - But requires more space

- **SON algorithm tries to deal with this (next)**

SON Algorithm

每次读取whole data的子集进入main memory，运行algorithm。

- **SON Algorithm:** Repeatedly read small subsets of the baskets into main memory and run an in-memory algorithm to find all frequent itemsets
  - **Note:** we are not sampling, but processing the entire file in memory-sized chunks

- An itemset becomes a **candidate** if it is found to be frequent in ***any*** one or more subsets of the baskets.

- On a **second pass**, count all the candidate itemsets and determine which are frequent in the entire set

- **Key "monotonicity" idea:** An itemset cannot be frequent in the entire dataset unless it is frequent in at least one subset

- However, even with SON algorithm we still don't know whether we found **all** frequent itemsets
- **Toivonen's algorithm solves this (next)**

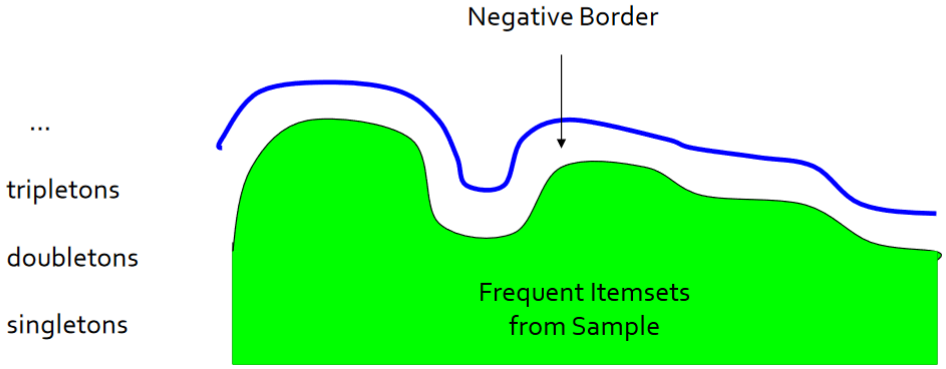
Toivonen Algorithm

Pass 1:

- **Start with a random sample, but lower the threshold slightly for the sample:**
  - **Example:** If the sample is 1% of the baskets, use $s/125$ as the support threshold rather than $s/100$
- Find frequent itemsets in the sample
- Add the **negative border** to the itemsets that are frequent in the sample :
  - **Negative border:** An itemset is in the negative border if it is not frequent in the sample, but *all* its immediate subsets are
    - Immediate subset = "delete exactly one element"

在每次itemset中加入negative border，即本身不是frequent set但是删去任意一个元素之后的子集都是frequent的set。

- $\{A,B,C,D\}$ is in the negative border if and only if:
  1. It is not frequent in the sample, but
  2. All of $\{A,B,C\}$, $\{B,C,D\}$, $\{A,C,D\}$, and $\{A,B,D\}$ are.

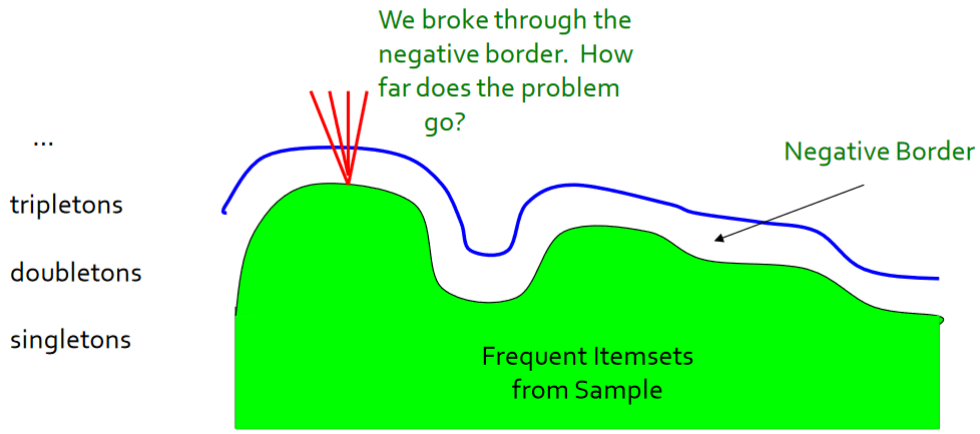


Toivonen's Algorithm

- **Pass 1:**
  - Start with the random sample, but lower the threshold slightly for the subset
  - Add to the itemsets that are frequent in the sample the **negative border** of these itemsets
- **Pass 2:**
  - Count all **candidate frequent itemsets from the first pass**, and also count sets in their **negative border**
- If **no** itemset from the negative border turns out to be frequent, then we found *all* the frequent itemsets.
  - What if we find that something in the negative border is frequent?
    - We must start over again with another sample!
    - Try to choose the support threshold so the probability of failure is low, while the number of itemsets checked on the second pass fits in main-memory.

如果所有的candidate frequent itemsets 的negative border中没有frequent set，那么我们就找到了全部frequent set。

If Something in the Negative Border Is Frequent . . .



Theorem:

- If there is an itemset *S* that is frequent in full data, but not frequent in the sample, then the negative border contains at least one itemset that is frequent in the whole.

Proof by contradiction:

- Suppose not; i.e.;
  1. There is an itemset *S* frequent in the full data but not frequent in the sample, and
  2. Nothing in the negative border is frequent in the full data
- Let *T* be a smallest subset of *S* that is not frequent in the sample (but every subset of T is)
- *T* is frequent in the whole (*S* is frequent + monotonicity).
- But then *T* is in the negative border (contradiction)