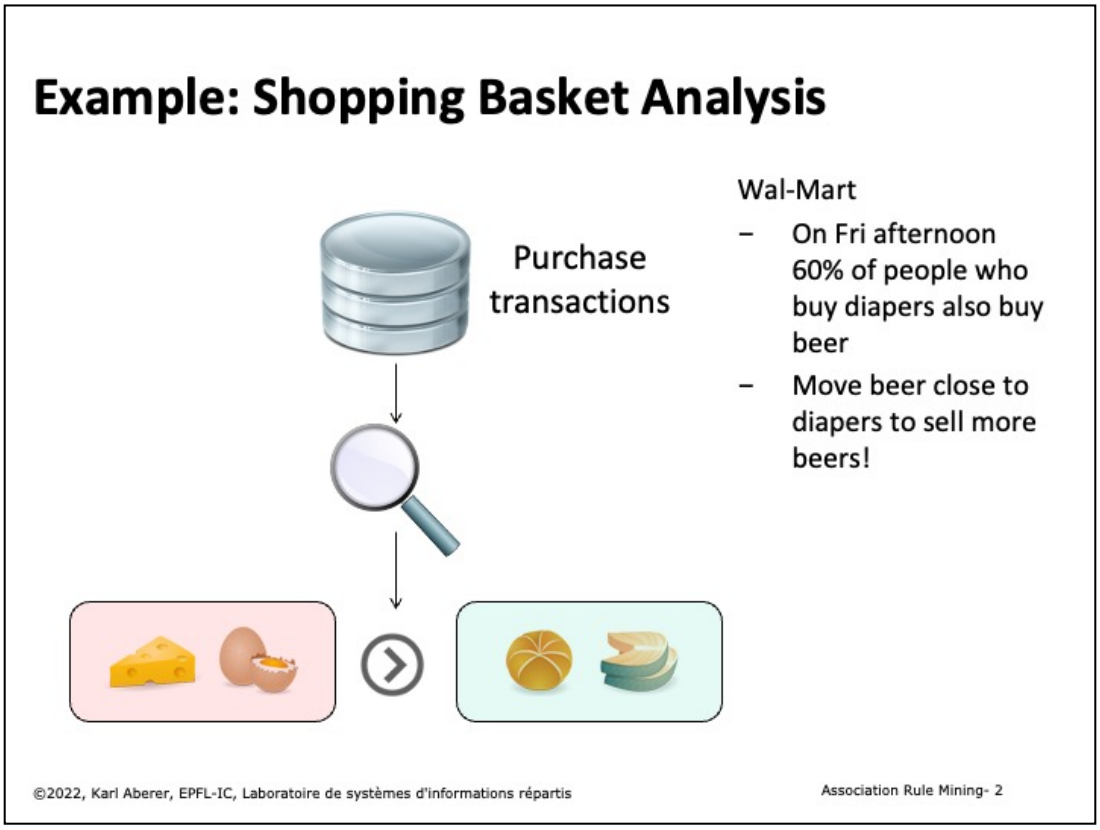
2.5 ASSOCIATION RULE MINING



The classical example of a data mining problem is "market basket analysis". Retail stores gather information on what items are purchased by their customers. The expectation is, by finding out what products are frequently purchased together (i.e., are associated with each other), identifying ways to optimize the sales of the products by better targeting certain groups of customers (e.g., by planning the layout of a store or planning advertisements). A well-known example was the discovery that people who buy diapers also frequently buy beers (probably exhausted fathers of small children). Therefore, nowadays one finds frequently beer close to diapers in supermarkets, and of course also chips close to beer. This type of problem was the starting point for one of the best-known data mining techniques: association rule mining.

2.2.5.1 Association Rules

We search association rules of the form

Body ➔ Head [support, confidence]

1. Body: predicate(*x*, item) a property of *x*
2. Head: predicate(*x*, item) a property likely to be implied by the Body
3. Support, confidence: measures of the validity of the rule

Example: buy(x, diapers) ➔ buy(x, beer) [0.5%, 60%]

Association rule mining is a technique for discovering unsuspected data dependencies and is one of the best-known data mining techniques. An association rule captures dependencies of different properties of an entity, denoted by x. For shopping basket analysis, the entity would be a shopping basket. The properties are expressed as predicates involving some items. The resulting rules state that if the entity has the property stated in the body, then the entity has also the property stated in the head with the probability stated by the confidence. The support expresses additionally how well-founded the rule is. We will introduce later in detail how these measures are defined.

In principle, such rules could be easily found by an exhaustive exploration of all possible dependencies, which is however prohibitively expensive. Association rule mining thus solves the problem of how to search efficiently for those dependencies.

Single- and Multi-dimensional Rules

Single-dimensional rules

buy(x, diapers) ➔ buy(x, beer) [0.5%, 60%]

Multi-dimensional rules

age(x, 19-25) ∧ buy(x, chips) ➔ buy(x, coke) [10%, 75%]

In the simplest case association rules use only a single predicate, such as buy. It is not hard to imagine that also other properties of the entities could be used in rules. For example, in the shopping basket example the age of the buyer could be another property of the shopping basket. Then we are talking about multi-dimensional rules.

From Multi- to Single-dimensional Rules

Use predicate/value pairs as items

age(x, 19-25) $\wedge$ buy(x, chips) ➔ buy(x, coke)

customer(x, age=19-25) $\wedge$ customer(x, buy=chips) ➔ customer(x, buy=coke)

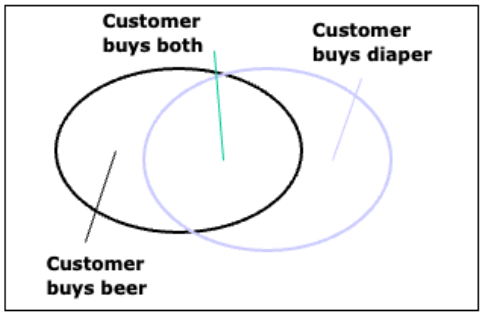
Simplified notation of single-dimensional rules

{diapers} ➔ {coke}

{age=19-25, buy=chips} ➔ {buy=coke}

However, it is straightforward to transform multi-dimensional association rules into single-dimensional rules, by combining predicate names and values into a new set of items of uniform type, consisting of predicate/value pairs. Therefore, we need to consider in the following only the case of single-dimensional rules. Since for similar rules there is only a single predicate applied to the entities, we can simplify the notation by dropping this predicate at all, and simply list the set of items in the body and head of the rules. This results in the usual notation adopted for association rules that captures dependencies among sets of items, or itemsets as they are usually called.

Basic Concepts



Transaction ID	Items Bought
2000	beer, diaper, milk
1000	beer, diaper
4000	beer, milk
5000	milk, eggs, apple

For a rule body ➔ head

Support: probability that body and head occur in transaction
p(body ∪ head)

Confidence: probability that if body occurs, also head occurs
p(head | body)

Association rule mining uses as input a set of transactions that consist of itemsets. We can illustrate the relationships among items using Venn diagrams. Each subset of the diagram represents a set of transactions containing some itemset or item. The measures of support and confidence for a rule body -> head are then defined as follows. The support gives the probability that a transaction contains all the items occurring in the rule, i.e., the probability that both all items of the body and of the head occur in the transaction. In other words, this is the probability that the rule applies to a transaction. The confidence is the probability that if the items of the body of a rule occur in the transaction, then also the items in the head of the rule occur in the transaction. In other words, this is the probability that the rule makes a correct prediction. The support expresses of how many relevant transactions exist for a rule, and therefore how well-founded it is, whereas the confidence expresses how reliable a rule is.

Support and Confidence

Transaction ID	Items Bought
2000	beer, diaper, milk
1000	beer, diaper
4000	beer, milk
5000	milk, eggs, apple

Rule 1: {beer} ➔ {diaper}

p({beer, diaper})=2/4
p({diaper} | {beer})=2/3

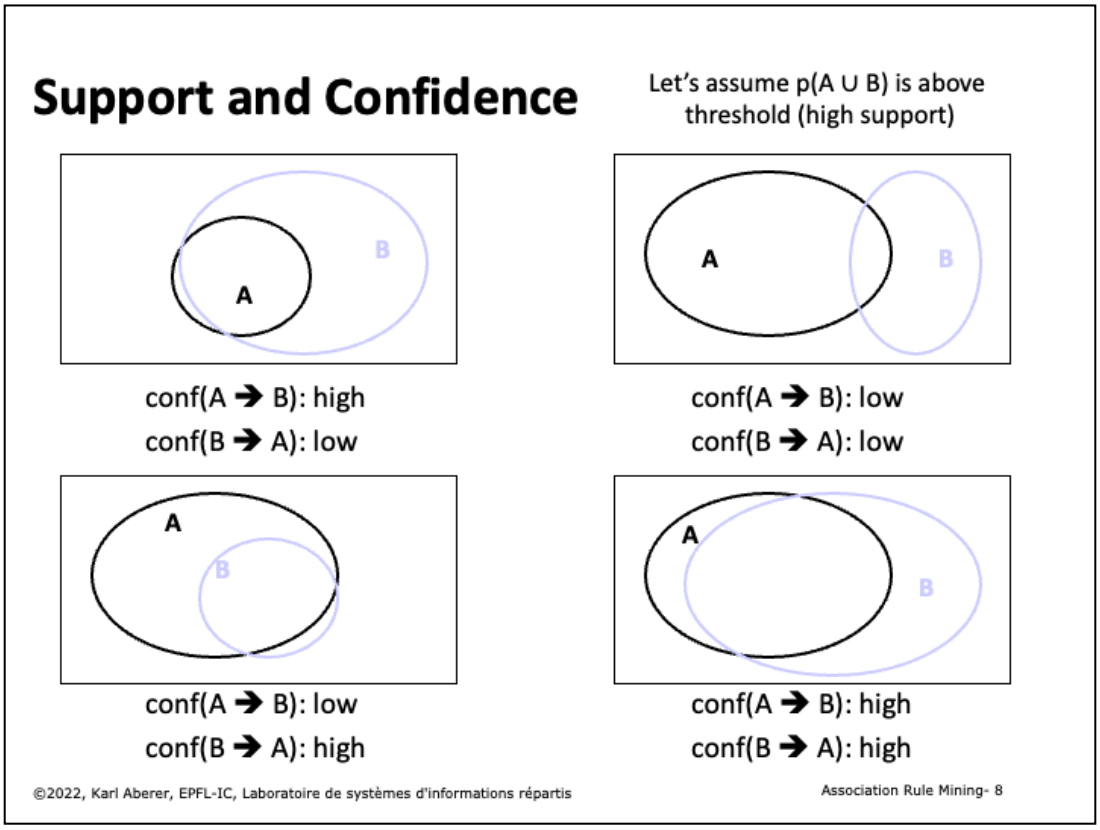
{beer} ➔ {diaper}[50%, 66%]

Rule 2: {diaper} ➔ {beer}

p({diaper, beer})=2/4
p({beer} | {diaper})=2/2

{diaper} ➔ {beer}[50%, 100%]

Here we compute support and confidence for two different rules.



Considering two itemsets A and B, we can define one association rule for each direction, A->B and B->A. Assuming that the support for the union of items in A and B is high, we can find 4 different combinations for the confidence of the two possible rules. This shows that an association rule is indeed directed, and not necessarily symmetric. A directed association rule, which is in general the interesting case, is related to an approximate subset relationship of the corresponding transaction sets.

Definition of Association Rules

Terminology and Notation

- Set of all items I, subset of I is called **itemset**
- **Transaction** (tid, T), $T \subseteq I$ itemset, transaction identifier tid
- Set of all transactions D (**database**), Transaction $T \in D$

Association Rules $A \rightarrow B$ [s, c]

- A, B itemsets ($A, B \subseteq I$)
- $A \cap B$ empty
- $s(A \rightarrow B) = s(A \cup B) = p(A \cup B) = count(A \cup B) / |D|$ (support)
- $c(A \rightarrow B) = p(B \mid A) = s(A \cup B) / s(A)$ (confidence)

Here we summarize the concepts and notations that are used for specifying and characterizing association rules. Association rules are defined for itemsets that are subsets of set of times. They are defined with respect to a set of transactions that are itemsets. The rules are defined such that the body and head of the rule have empty intersection. The support is defined as the probability of the union of the body and head to occur in a transaction. This probability is computed by counting the number of transactions that contain those items. The confidence is the conditional probability of the head of the rule occurring, given the body. This probability can be computed from the support of the rule, and the support of the body.

10 itemsets out of 100 contain item A, of which 5 also contain B. The rule A ➔ B has:

A. 5% support and 10% confidence
B. 10% support and 50% confidence
C. 5% support and 50% confidence
D. 10% support and 10% confidence

10 itemsets out of 100 contain item A, of which 5 also contain B. The rule B ➔ A has:

A. unknown support and 50% confidence
B. unknown support and unknown confidence
C. 5% support and 50% confidence
D. 5% support and unknown confidence

Association Rule Mining: Problem

Problem

Given a database D of transactions (tid, T)

Find

all rules A ➔ B [s, c]

such that $s > s_{min}$ (high support)

and $c > c_{min}$ (high confidence)

The problem of mining association rules is now defined as follows: Given a database of transactions, find all rules that correlate the presence of one itemset with that of another itemset where the support and the confidence are above a given threshold. In other words, find all association rules that have high support and confidence.

2.5.2 Apriori Algorithm

Two step approach:

1. Find **frequent** itemsets
   $J \subseteq I$ such that $p(J) > s_{min}$
2. Select **pertinent** rules
   A ➔ B such that $A \cup B = J$
   and
   $p(B|A) > c_{\min}$

We will approach the problem in two steps, by first considering only the problem of finding itemsets that satisfy the condition on having a high support, which is a necessary condition for a rule to be an association rule, and then extracting from those itemsets the rules that have a high confidence. We have already noticed in the confidence value can be computed from support values of itemsets. Thus it is natural to first identify itemsets with high support.

Frequent Itemsets

Transaction ID	Items Bought
2000	beer, diaper, milk
1000	beer, diaper
4000	beer, milk
5000	milk, eggs, apple

Frequent Itemset	Support
{beer}	0.75
{milk}	0.75
{diaper}	0.5
{beer,milk}	0.5
{beer, diaper}	0.5
{milk, diaper}	0.25

1. A ➔ B can be an association rule, only if A ∪ B is a frequent itemset
2. Any subset of a frequent itemset is also a frequent itemset (**Apriori property**)
$$p(J) > s_{min} \rightarrow p(J' \subseteq J) > s_{min}$$
3. Find frequent itemsets with increasing cardinality, from 1 to k, to reduce the search space

A necessary condition for finding an association rule of the form A->B is that it has a sufficiently high support. Therefore, for finding such rules, we can first identify itemsets within the transactions that occur sufficiently frequent. These are called *frequent itemsets*. Second, we can observe that any subset of a frequent itemset is necessarily also a frequent itemset. This is called the *apriori property*. The a priori property is the central idea in the algorithm we will introduce in the following. It will be used for multiple purposes. A first use of this property is the observation that we can search for frequent itemsets by searching them by increasing cardinality: once frequent itemsets of lower cardinality are found, only itemsets of larger cardinality need to be considered that contain one of the frequent itemsets already found. This allows us to dramatically reduce the search space.

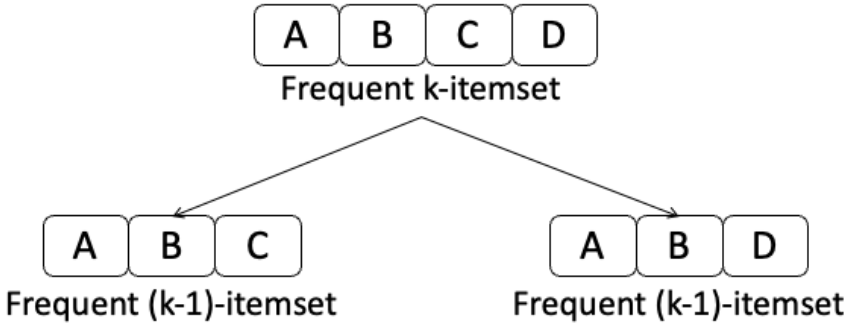
Exploiting the Apriori Property

If we know all frequent (k-1)-itemsets, which are candidate frequent k-itemset **X**?

1. Any (k-1)-itemset that is subset of **X** must be frequent
2. Any ordered (k-1)-itemset that is subset of **X** must be frequent
3. *In particular, two ordered (k-1)-itemsets that are subsets of **X** and differ only in their last position must be frequent*

Assume that we know frequent itemsets of size k-1 and want to construct an itemset of size k (a k-itemset). What properties does such a k-itemset have to satisfy? We can say, that any subset of size k-1 must be frequent. We can also order those subsets by their elements. We can that even say that two ordered subsets that differ only in the last position must be frequent. This gives is now a possibility to construct k-itemsets systematically.

Exploiting the Apriori Property: Candidates



The union of two (k-1)-itemsets that differs only by one item is a **candidate** frequent k-itemset

If two ordered subsets of size k-1 of a frequent itemset of size k that differ only in the last position must be frequent, we can invert the argument. Assume we know all frequent itemsets of size k-1 then, then the itemset of size can only be frequent if it can be constructed from two k-1 itemsets in exactly this way. An itemset of size k constructed this way is then called a candidate frequent k-itemset. We do not know whether it is frequent indeed. The construction is a necessary condition, but not a sufficient one.

Example

Transaction ID	Items Bought
2000	beer, diaper, milk
1000	beer, diaper
4000	beer, milk
5000	milk, eggs, apple

Frequent Itemset	Support
{beer}	0.75
{milk}	0.75
{diaper}	0.5
{beer,milk}	0.5
{beer, diaper}	0.5
{milk, diaper}	0.25

{beer} and {milk} differ by one item, thus {beer,milk} is a candidate 2-itemset

{beer, milk} and {beer, diaper} differ by one item, thus {beer, diaper, milk} is a candidate 3-itemset

Note that the candidate itemsets generated by the join step are not necessarily frequent itemsets, as the example illustrates.

Exploiting the Apriori Property: JOIN step

Given the frequent (k-1)-itemsets, L_{k-1}, we can construct a candidate set C_k by joining two (k-1)-itemsets that differ by exactly 1 item in the last position

Algorithm (**JOIN**)

1. Sort the itemsets in L_{k-1}
2. Find all pairs with the same first k-2 items, but different k-1^{th} item
3. Join the two itemsets in all pairs and add to C_k

In order to create candidates efficiently we consider only one specific way of constructing candidates. We only combine itemsets that have the first k-2 elements identical and differ in their last position. If a k-itemset is frequent it necessarily is (also) constructed this specific way.

For example, if we have

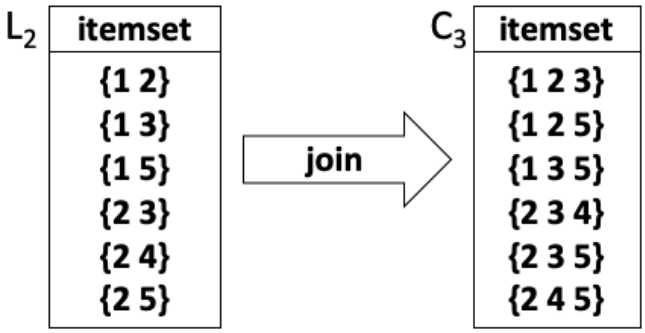
ABC

ACD

BCD

If we combine ACD and BCD (which also differ by one item) we would get ABCD. But ABCD have been already generated by combining ABC and ABD.

JOIN step: Example

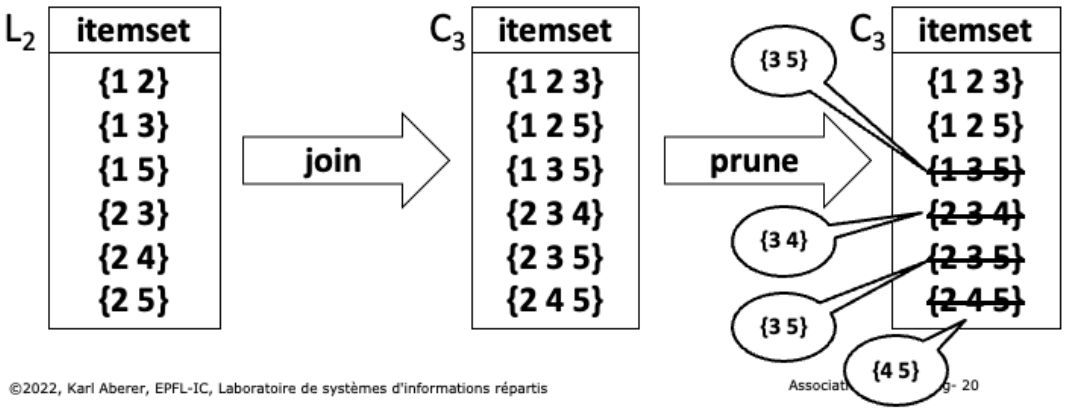


Here we illustrate the join step for itemsets of size 2, to construct itemsets of size 3. Only pairs of itemsets of size 2 are considered that coincide in their first position.

Exploiting the Apriori Property: PRUNE Step

A k-itemset in the candidate set C_k might still contain (k-1)-itemsets that are not frequent (k-1)-itemsets

Eliminate them (**PRUNE**)



The k-itemsets constructed in the join step are not necessarily frequent k-itemsets. One possible reason is that they contain some subsets of items which are not a frequent itemset. These can be eliminated in a prune step. By checking if every (k-1) itemset of a candidate itemset is indeed a frequent (k-1) itemset some of the candidate itemsets can be eliminated.

Final Step

For the remaining k-itemset in the candidate set C_k eliminate those that are not frequent by counting how often they occur in the database

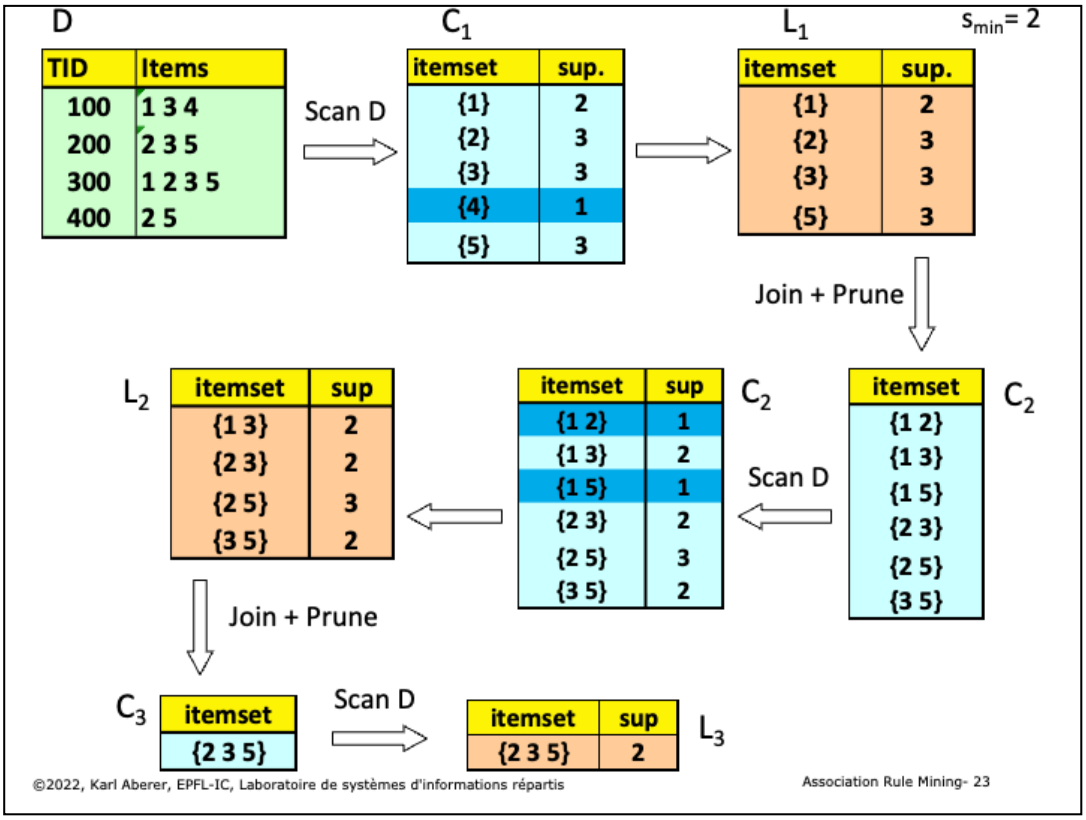
- The final step is the most expensive (full access to database D)
- Advantage: Performed only for a smaller number of candidate itemsets, reduced by **JOIN** and **PRUNE**

After the prune step is completed, still the remaining candidates can be non-frequent. Therefore, the frequency of these itemsets needs to be determined by accessing the database and counting the number of their occurrences. An important aspect of the apriori algorithm is that this last step is the most expensive one, since it requires access to the complete database. By performing the join and prune steps the operation needs to be performed only for a reduced number of itemsets.

Apriori Algorithm

$k := 1, L_k := \{ J \subseteq I : |J|=1 \wedge p(J) > s_{min} \}$ //frequent items
while $L_k \neq \emptyset$
    $C'_{k+1} := JOIN(L_k)$ // join itemsets in L_k that differ by one element
    $C_{k+1} := PRUNE(C'_{k+1})$ // remove non-frequent itemsets
    for all (id,T) ∈ D // compute the frequency of candidate
        for all $J \in C_{k+1}, J \subseteq T$
            count(J)++
        end for
    end for
    $L_{k+1} := \{ J \subseteq C_{k+1} : p(J) > s_{min} \}$ // add candidate frequent itemsets
    k := k+1
end while
return $\cup_k L_k$

This is the summary of the complete apriori algorithm.



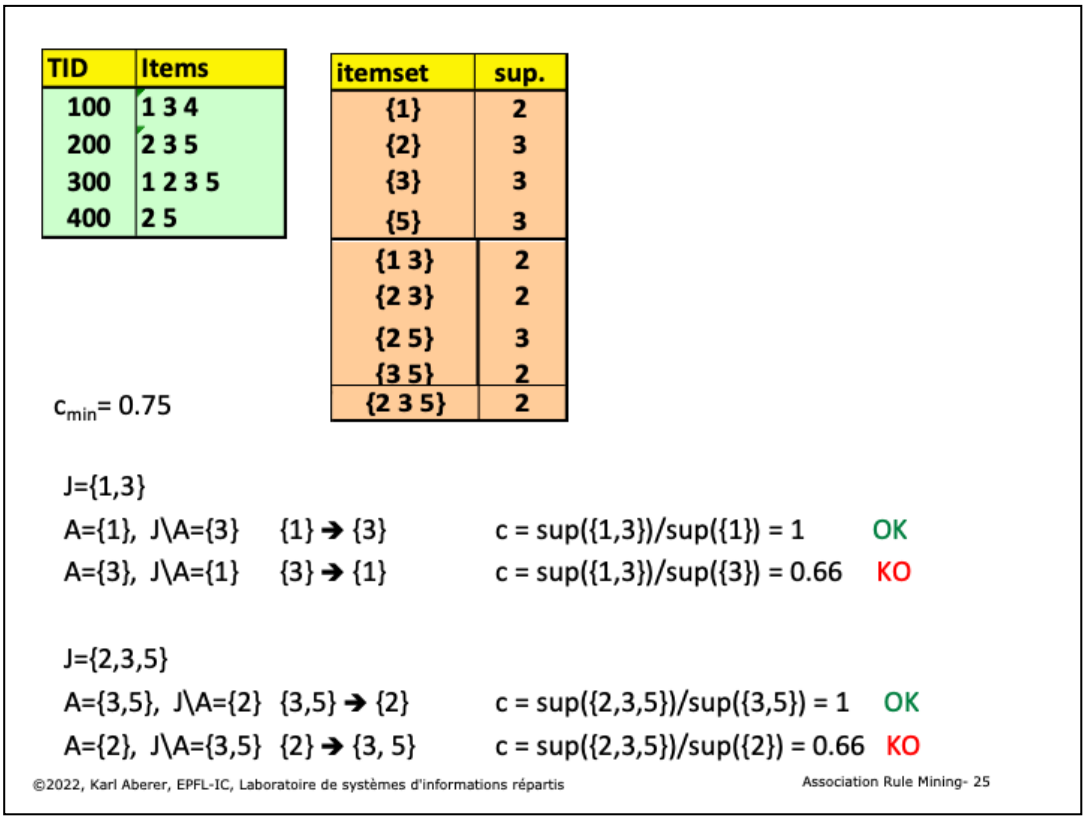
Here we give a complete example of the exection the apriori algorithm. Notice, in this example, how the scan steps (when determining the frequency with respect to the database) eliminate certain itemsets. Pruning does not lead to any elimination of itemsets in this example. The algorithm built 9+3 (frequent + non frequent) = 12 itemsets out of $2^5 = 32$ total possible itemsets

Select pertinent rules

```
R := ∅                                          // initial set of rules
L := Apriori(D)                                 // set of frequent itemsets
for all J ∈ L
        for all A ⊆ J, A != ∅
                r := A ➔ J \ A                  // create candidate rule
                if c(A ➔ J \ A) = s(J)/s(A)> c_min
                        R := R ∪ r
                end if
        end for
end for
```


Once the frequent itemsets are found with the apriori algorithm, the derivation of relevant association rules is straightforward. One checks for every frequent itemset whether there exists a subset A that can occur as the body of a rule. For doing that, the support count, i.e., the frequency of the itemset in the database, which was obtained during the execution of the Apriori algorithm, is used to compute the confidence as a conditional probability. Note that also L\A is a frequent itemset, and therefore the support count is available for that set as well from the apriori algorithm.

Note that $c(A \rightarrow J \setminus A) = s((J \setminus A) \cup A)/s(A) = s(J)/s(A)$.



Here we illustrate the extraction of association rules using the known set of frequent itemsets.

Given the frequent 2-itemsets {1,2}, {1,4}, {2,3} and {3,4}, how many 3-itemsets are generated and how many are pruned?

A. 2, 2

B. 1, 0

C. 1, 1

D. 2, 1

After the join step, the number of k+1-itemsets ...

A. is equal to the number of frequent k-itemsets
B. can be equal, lower or higher than the number of frequent k-itemsets
C. is always higher than the number of frequent k-itemsets
D. is always lower than the number of frequent k-itemsets

If rule {A,B} ➔ {C} has confidence c_1 and rule {A} ➔ {C} has confidence c_2, then ...

A. $c_2 >= c_1$
B. $c_1 > c_2$ and $c_2 > c_1$ are both possible
C. $c_1 >= c_2$

Interesting Association Rules

Not all high-confidence rules are interesting

	$Coffee$	$\overline{Coffee}$	
Tea	150	50	200
$\overline{Tea}$	650	150	800
	800	200	1000

{Tea} ➔ {Coffee} has support 0.15 and confidence 0.75

- But 80% of people drink coffee anyway
- Drinking tea decreases the probability to drink coffee!

High confidence and support do not necessarily imply that a rule is interesting or useful. We illustrate this by the following example. The matrix indicates the number of entities that like or do not like coffee or tea. If we consider the rule {Tea} –> {Coffee} we will find that it has support 0.15 and confidence 0.75, which we may consider as very high. At the first glance this looks like a good rule. Looking more carefully, we will realize that in fact 80% of people drink coffee. In view of this, the rule just holds, because people in general drink a lot of coffee. Even worse, considering that 80% of people drink coffee, among the people drinking tea the propensity to drink coffee is less pronounced, only 75%. In conclusion, drinking tea has a negative impact on drinking coffee which is the opposite to what the rule suggests. This motivates the need for alternative measures to evaluate whether a rule is interesting.

Alternative Measures of Interest

Added Value (A, B itemsets)

$$AV(A \rightarrow B) = confidence(A \rightarrow B) - support(B)$$

Interesting rules are those with high positive or negative interest values (usually above 0.5)

Alternative: $Lift(A \rightarrow B) = \frac{confidence(A \rightarrow B)}{support(B)}$

Example:

- AV({Tea} ➔ {Coffee}) = 0.75 – 0.8 = -0.05
- Lift({Tea} ➔ {Coffee}) = 0.75 / 0.8 = 0.9375

Quantitative Attributes

Transforming quantitative (numeric ordered values) into categorical ones

- *Static discretisation* into predefined bins
- *Dynamic discretisation* based on the distribution of the data

The rules depend on the chosen discretisation!!

(1) age{*x*, [18,19]} ∧ live{*x*, Lausanne} ➔ work{*x*, student}

(2) age{*x*, [20,21]} ∧ live{*x*, Lausanne} ➔ work{*x*, student}

vs.

(1) age{*x*, [18,21]} ∧ live{*x*, Lausanne} ➔ work{*x*, student}

Association rules can also be derived for quantitative data. In order to obtain finite itemsets, the quantitative domains are transformed to discrete ones using discretization. Methods of discretization are widely used in machine learning for data preprocessing. Two basic approaches are static discretization, splitting the data in bins of uniform size, and dynamic discretization splitting the data in bins with uniform number of elements. Also, semantic aspects can be taken into consideration when performing discretization, as the example illustrates. A static discretization could create there bins that are semantically less meaningful.

Improving Apriori for Large Datasets

1. Transaction reduction
   - A transaction that does not contain any frequent k-itemset is useless in subsequent scans
2. Sampling
   - Mining on a sampled subset of DB, with a lower support
3. Partitioning (SON algorithm)
   - Any itemset that is potentially frequent in a DB must be frequent in at least one of the partitions of the DB

Though the basic Apriori algorithm is designed to work efficiently for large datasets, there exist several possible improvements:

•Transactions in the database that turn out to contain no frequent k-itemsets can be omitted in subsequent database scans.

•The sampling method selects samples from the database and searches for frequent itemsets in the sampled database, using a correspondingly lower threshold for the support.

•One identify first frequent itemsets in partitions of the database, that fit in memory. This method exploits the idea that if an itemset is not frequent in any of the partitions (local frequent itemset) then it is also not be frequent in the whole database.

Sampling

Approach

1. Randomly sample transactions with probability p
2. Detect frequent itemsets with support p*s
3. Eliminate **false positives** by counting frequent itemsets on complete data after discovery

Refinements

- If we assume that the m transactions are randomly sorted, we can just choose the first p*m ones
- **False negatives** can be reduced by choosing a lower threshold, e.g., 0.9 p*s

Random sampling data before searching association rules must consider two issues: false positives and false negatives. False positives can be dealt with by verifying the support for the itemsets on the complete transaction set. For false negatives, no such possibility exists. By decreasing the support threshold, the number of false negatives can be reduced, at the cost of testing more candidate itemsets when scanning the complete dataset.

A further optimization can be achieved, when we know that the transactions occur in random order. Then sampling can be replaced by simple using the first p*m elements of the database.

Partitioning

Approach

1. Divide transactions in 1/p partitions and repeatedly read partitions into main memory
2. Detect in-memory algorithm to find all frequent itemsets with support threshold p*s
3. An itemset becomes a candidate if it is found to be frequent in **at least** one partition
4. On a **second pass**, count all the candidate itemsets and determine which are frequent in the entire set of transactions

With partitioning we are not sampling the dataset but processing the complete transaction set partition by partition. If an itemset is frequent in one partition (with a correspondingly adapted threshold), then it becomes a candidate. At this stage it is not sure that it is also frequent for the whole transaction set. Therefore, in a second pass itemsets that are frequent are determined by scanning the whole transaction database.

Partitioning – Why it works

Key “monotonicity” idea

- an itemset cannot be frequent in the entire set of transactions unless it is frequent in at least one subset
- Proof: If for all partitions support is below p*s, the total support is less than (1/p) p*s = s!

The second pass is needed, since the condition of being frequent in at least one partition is necessary, but not sufficient.

Partitioning – Distributed Version

Partitioning lends itself to distributed data mining

Transactions distributed among many nodes

- Compute frequent itemsets at each node
- Distribute candidates to all nodes
- Accumulate the counts of all candidates

MapReduce implementation!

Partitioning is an embarrassingly parallel algorithm, and can therefore be naturally executed in a distributed environment, e.g., using Map-Reduce

A false negative in sampling can only occur for itemsets with support smaller than …

A. the threshold s
B. p*s
C. p*m
D. None of the above

2.5.3 FP Growth

Frequent itemset discovery without candidate itemset generation

Proceeds in 2 steps:

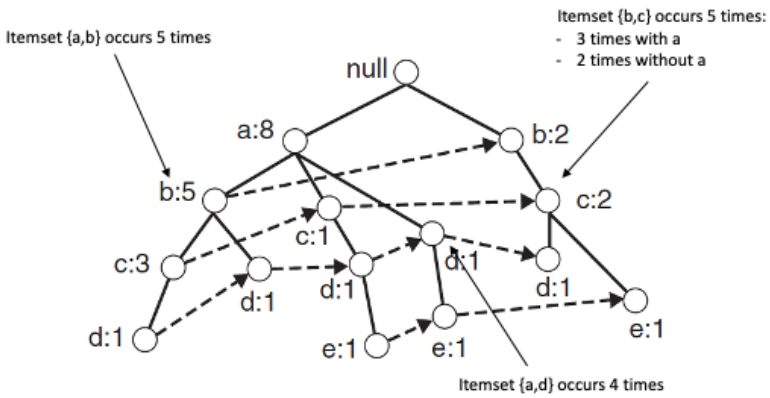
1. Build a data structure, called the FP-tree
   – Requires 2 passes over the dataset
2. Extract frequent itemsets directly from the FP-tree

Though Apriori is as the original method the most popular association rule mining algorithm, other algorithms have been devised that attempt to provide better performance under certain circumstances. FP Growth is one example of such an algorithm that is mainly aiming at main memory implementations (and thus less suitable for distributed implementation). It does not generate candidates for frequent itemsets but constructs them directly using a data structure called FP-Tree. The FP-Tree allows to extract directly the association rules.

FP-Tree Data Structure

Transaction Data Set

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	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}



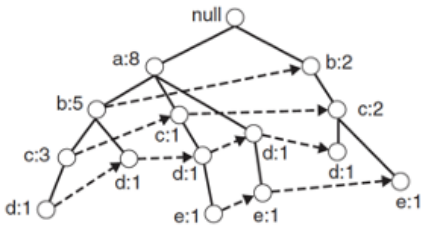
- Items are sorted by decreasing support
- Nodes correspond to single items
- Counters indicate frequency of itemset from root to node
- Different nodes for the same item are connected by a linked list

The structure of an FP-Tree is comparable to that of a trie. In order to construct the FP-Tree, first items in the itemsets are sorted. The nodes in the tree correspond to items, and paths from the root correspond to itemsets. Itemsets sharing the same prefix of items, share the same path prefix in the tree. Counters at the nodes indicate how frequent the itemset corresponding to the path from the root to the node is. If we consider the left most path, we see that item **a** occurs 8 times, itemset ab 5 times, **abc** 3 times, and **abcd** once. The same item can occur in the tree in multiple places. The different occurrences are connected as linked list. For itemsets that occur in different paths (e.g., bc, which occurs together with **a** and without **a**), the frequencies can be computed by following the linked lists of the last item in the itemset and testing whether the remaining items are present in the path.

FP-Tree: Item Sorting

Transaction Data Set

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	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}



Item frequencies
a: 8
b: 7
c: 6
d: 5
e: 3

FP-Tree Size: The FP-Tree is more compact the more common prefixes the itemsets share

- Sorting items by decreasing support increases this probability

Sorting items in itemsets by decreasing support is an important heuristics to keep the FP-Tree compact, as more frequent items tend to be on the top of the tree which increases the likelihood that different itemsets share a common prefix. In the example the items were sorted already according to their frequencies. If it is not the case, they need to be reordered according to their frequencies.

Step 1: FP-Tree Construction

Requires 2 passes over the dataset

Pass 1: Compute support for each item

- Pass over the transaction set
- Sort items in order of decreasing support

Pass 2: Construct the FP-Tree data structure

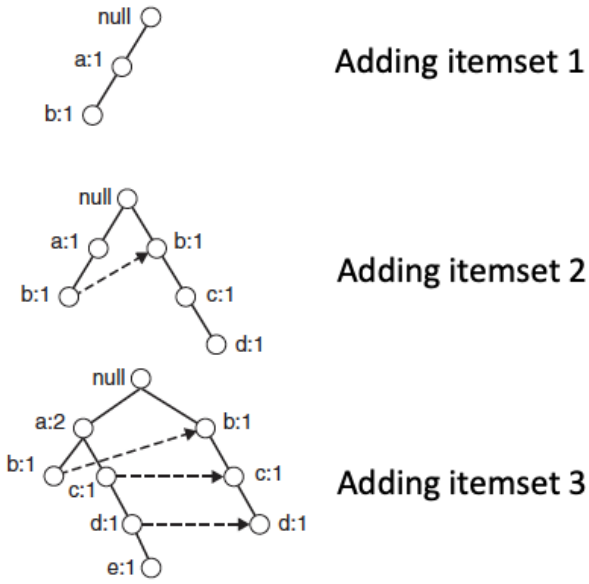
- Tree is expanded one itemset at a time

To construct the FP-Tree one proceeds in two passes (not to confuse with the two steps of the overall FP-Growth algorithm mentioned in the beginning). In a first pass the transactions are scanned to compute the support for each single item. The items are then sorted in decreasing order. This order is being used to sort itemsets before the construction of the FP-Tree. In the second pass the FP-Tree is constructed by adding one itemset at a time.

Step 1: FP-Tree Construction

Transaction Data Set

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	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}

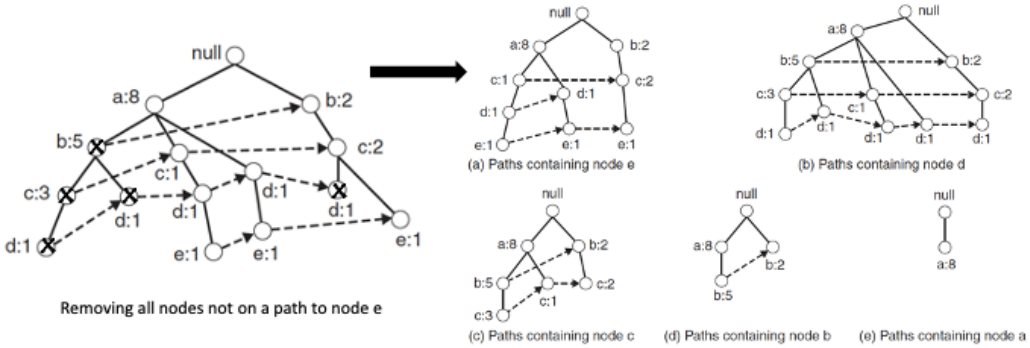


Constructing the FP-Tree proceeds in a way analogous to constructing a trie structure. The itemsets in the transaction database are processed sequentially. The items in an itemset are processed one by one. If for the current item there exists already a node in the FP-Tree, no new node is created. If it does not exist, a new branch is created for the item. The number of occurrences is updated by 1 at each node that is visited (since each subset of the itemset is also an itemset). Finally, links are added between nodes with the same labels.

Step 2: Frequent Itemset Extraction

For each item extract the tree with paths ending in the item

- Used to find frequent itemsets containing the item



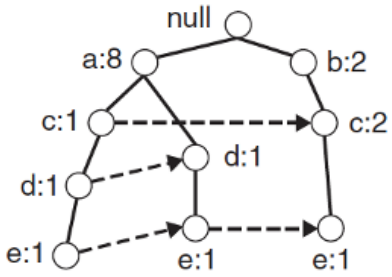
After the FP-Tree has been constructure in step 1, In the second step frequent itemsets are extracted from it. For each item, first a subtree is extracted from the full FP-Tree that contains only the paths ending in the item. For extracting those subtrees, the linked lists are helpful. One needs to traverse just the corresponding list and retain the paths from the traversed nodes to the root. The figures illustrates the resulting subtrees for the 5 items.

Step 2: Divide and Conquer Strategy

Start with item with lowest support (e.g., item *e*) and extract its tree

Check whether the item has sufficient support (e.g., 2)

- Add up counts along the list of the item
- e.g., item *e* has support 3



For extracting frequent itemsets, the algorithm starts with the item that has the lowest support. It extracts its corresponding tree, as described before. Next the support of the item can be computed by traversing the leaves of the tree, following the links. In the example, item **e** would this have support 3. If the support is above threshold, then processing proceeds to find larger itemsets containing the item, otherwise processing can stop since no frequent itemsets containing this item can exist.

Step 2: Divide and Conquer Strategy

If item has sufficient support, check for itemsets ending in the item

- Example: if item *e* is frequent check whether itemsets *de, ce, be, ae* are frequent
- again, in order of lowest support

In order to check whether these itemsets are frequent compute a **conditional FP-Tree**

If we find that the current item has sufficient support (e.g., item e), then we check next whether the itemsets consisting of two items and ending in the item have also sufficient support. For doing so we derive a **conditional FP-Tree** from the tree that we have constructed for the item before. The conditional tree is constructed in three steps:

- First, the support counts in the tree are updated such that only the number of itemsets containing the current item is considered. This can be performed traversing the paths from the bottom to the root.
- Second, the leaf nodes corresponding to the current item are removed
- Finally, all nodes that have an insufficient support count are also removed from the conditional FP Tree

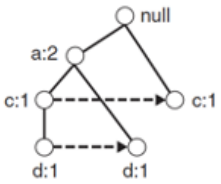
Conditional FP-Tree

Corresponds to the FP-Tree that would be built if

- we only consider transactions containing a particular itemset
- then omit this itemset (e.g., {e})
- drop items that are not frequent in the subset of the transaction set (e.g., {b})

Transaction Data Set

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	{a}
8	{a,b,e}
9	{a,b,d}
10	{b,c,e}

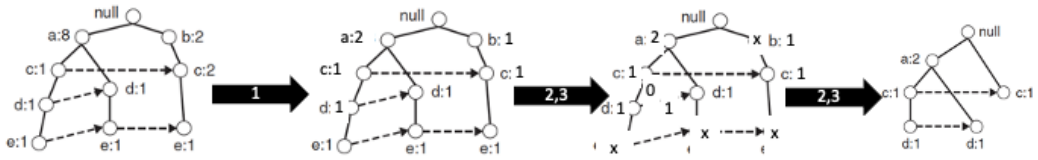


One way to understand the construction of the conditional FP Tree is that it is the FP Tree that would be constructed when we only consider the transactions that consider the current item (or itemset) and then remove **e** and any non-frequent item from those itemsets. In the example this would be item **e** that we consider, and item **b** would be eliminated, since not frequent in the remaining transactions.

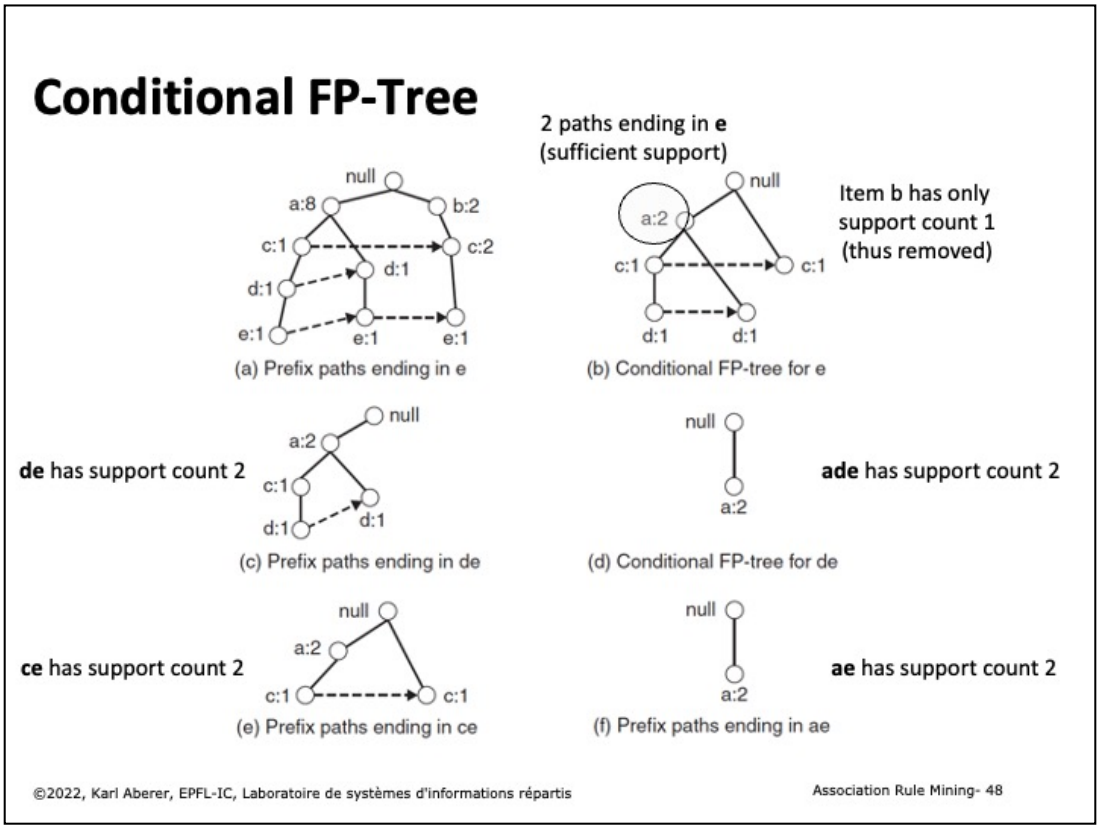
Deriving Conditional FP-Tree

The conditional FP-Tree can be derived from extracted tree

1. Update support counts to itemsets containing the item
2. Remove the nodes of the item
3. Remove nodes with insufficient support count



Here we illustrate the constructing of the conditional FP-Tree for **e**. Note, when we update the support counts, for the node **b**:2, there are are two itemsets that contribute to the support count of **b**. The itemset {b} and the itemset {b,c,e}. Since only one of them contains the item **e**, the support count for **b** is changed to b. Similarly for **a**:8 and **c**:2. Since the total support for **b** drops below 2, the node is finally removed from the tree.



Here we show all conditional FP-Trees that would be constructed from 2-itemsets ending in item **e**. Once the conditional FP Trees have been constructed, we can read off the support for all 2-itemsets, by traversing the lists at the leaf level (as we did first for item **e** itself). Once this step is finished, the algorithm continues in the same way for the remaining 2-itemsets. For example, for itemset **de** we would now construct a conditional FP Tree, by first extracting from tree (b) the tree with paths ending in **de**, resulting in tree (c), and then extracting the conditional FP Tree for itemset **de** (tree (d)).

Result

Frequent itemsets detected in this order

Suffix	Frequent Itemsets
e	{e}, {d,e}, {a,d,e}, {c,e},{a,e}
d	{d}, {c,d}, {b,c,d}, {a,c,d}, {b,d}, {a,b,d}, {a,d}
c	{c}, {b,c}, {a,b,c}, {a,c}
b	{b}, {a,b}
a	{a}

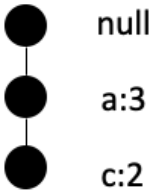
This table summarizes the frequent itemsets in the order they are detected, i.e., starting from the items with the lowest support. The first row contains exactly the itemsets containing the item **e**, in the order in which they are detected.

In the first pass over the database of the FP Growth algorithm

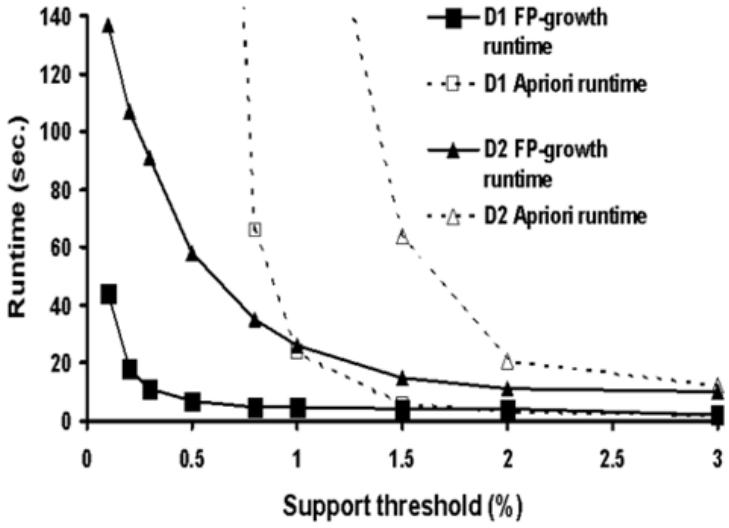
A. Frequent itemsets are extracted
B. A tree structure is constructed
C. The frequency of items is computed
D. Prefixes among itemsets are determined

The FP tree below is …

A. not valid, b is missing
B. not valid, since count at leaf level larger than 1
C. possible, with 2 transactions {a}
D. possible, with 2 transactions {a,c}



Performance Comparison



This performance comparison shows the behavior of FP Growth as compared to apriori. As the support threshold gets smaller, and therefore more rules are detected, the runtime of both algorithms increases. For smaller thresholds, FP Growth exhibtis increasingly better performance as compared to apriori., whereas for larger thresholds the advantage is less pronounced and disappears.

Summary FP Growth

Advantages

- Only 2 passes over the dataset
- Compresses dataset
- (Generally) much faster than Apriori

Disadvantages

- Works less efficiently for high support thresholds
- Must run in main memory
- Difficult to find distributed implementation

FP Growth Works less efficiently for high support thresholds, because it prunes only single items.

Components of Data Mining Algorithms

1. Pattern structure/model representation
   - **association rules**
2. Scoring function
   - **support, confidence**
3. Optimisation and search
   - **JOIN, PRUNE**
   - **FP-Tree, ordering of items**
4. Data management
   - **transaction reduction, partitioning, sampling**

Our study of association rule mining nicely illustrates the 4 main aspects of every data mining algorithm. The model type and scoring function are common to all machine learning algorithms. When it comes to large datasets also algorithmic optimizations and efficient management of large datasets become important questions.

References

Textbook

- http://www.mmds.org/mmds/v2.1/ch06-assocrules.pdf
- Jiawei Han, *Data Mining: concepts and techniques*, Morgan Kaufman, 2000, ISBN 1-55860-489-8
- Pang-Ning Tan, Michael Steinbach, Vipin Kumar: *Introduction to Data Mining*, Addison-Wesley. Chapter 6: Association Analysis: Basic Concepts and Algorithms

Some relevant research literature

- R. Agrawal, T. Imielinski, and A. Swami. *Mining association rules between sets of items in large databases*. SIGMOD'93, 207-216, Washington, D.C.
- J. Han, J. Pei and Y. Yin. *Mining Frequent patterns without candidate generation*. SIGMOD 2000, 1–12.