Chapter 6: Mining Association Rules in Large Database

Outline

- Association rule mining
- Mining single-dimensional Boolean association rules from transactional databases
- Mining multilevel association rules from transactional databases
- Mining multidimensional association rules from transactional databases and data warehouse
- From association mining to correlation analysis
- Constraint-based association mining
- Summary

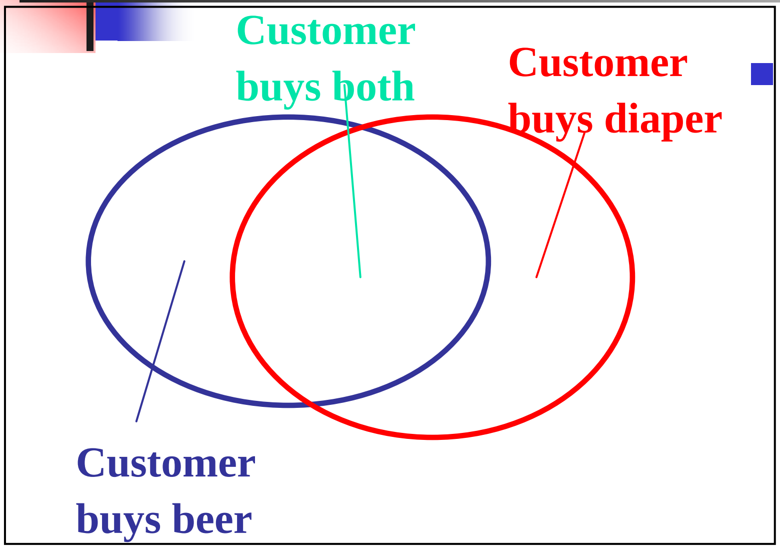
What Is Association Rule Mining?

- Association rule mining:
  - Finding frequent patterns, associations, correlations, or causal structures among sets of items or objects in transaction databases, relational databases, and other information repositories.
- Applications:
  - Basket data analysis, cross-marketing, catalog design, loss-leader analysis, clustering, classification, etc.
- Examples.
  - Rule form: "Body $\rightarrow$ Head [support, confidence]".
  - buys(x, "diapers") $\rightarrow$ buys(x, "beers") [0.5%, 60%]
  - major(x, "CS") ^ takes(x, "AI") $\rightarrow$ grade(x, "A") [1%, 75%]

Association Rule: Basic Concepts

- Given: (1) database of transactions, (2) each transaction is a list of items (purchased by a customer in a visit)
- Find: all rules that correlate the presence of one set of items with that of another set of items
  - E.g., *98% of people who purchase tires and auto accessories also get automotive services done*
- Applications
  - *** $\Rightarrow$ *Maintenance Agreement* (What the store should do to boost Maintenance Agreement sales)
  - *Home Electronics* $\Rightarrow$ *** (What other products should the store stocks up?)
  - Attached mailing in direct marketing

Rule Measures: Support and Confidence



- Find all the rules $X \& Y \Rightarrow Z$ with minimum confidence and support
  - support, s, probability that a transaction contains {X U Y U Z}
  - confidence, c, conditional probability that a transaction having {X U Y} also contains Z

Transaction ID	Items Bought
2000	A,B,C
1000	A,C
4000	A,D
5000	B,E,F

Let minimum support 50%, and minimum confidence 50%, we have

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

Association Rule Mining: A Road Map

- Boolean vs. quantitative associations (Based on the types of values handled)
  - buys(x, “SQLServer”) ^ buys(x, “DMBook”) → buys(x, “DBMiner”) [0.2%, 60%]
  - age(x, “30..39”) ^ income(x, “42..48K”) → buys(x, “PC”) [1%, 75%]
- Single dimension vs. multiple dimensional associations (see ex. Above)

Outline

- Association rule mining
- Mining single-dimensional Boolean association rules from transactional databases
- Mining multilevel association rules from transactional databases
- Mining multidimensional association rules from transactional databases and data warehouse
- From association mining to correlation analysis
- Constraint-based association mining
- Summary

Mining Association Rules—An Example

Transaction ID	Items Bought
2000	A,B,C
1000	A,C
4000	A,D
5000	B,E,F

Min. support 50%
Min. confidence 50%

Frequent Itemset	Support
{A}	75%
{B}	50%
{C}	50%
{A,C}	50%

For rule $A \Rightarrow C$:

support = support({A and C}) = 50%

confidence = support({A and C})/support({A}) = 66.6%

The Apriori principle:

Any subset of a frequent itemset must be frequent

Terminologies

- **itemset :** a set of items
- **k-itemset** : an itemset that contains k items
- **Minimum support and minimum support count**
- **Candidate k-itemset**
- **frequent k-itemset:** a k-itemset whose occurrence frequency is greater than or equal to a pre-defined **minimum support count**

- **minimum confidence**
- **strong association rules :** association rules that satisfy both the minimum support and the minimum confidence threshold

- **association rule** X ➔ Y,

  where X and Y are subsets of J, and X and Y do not share common item.

- Support of the rule:

  The rule X ➔ Y holds in the transaction set D with **support** s, where s is the percentage of transactions in D that contain both X and Y. (Computed as P(X and Y))

- Confidence of the rule:

  The rule X ➔ Y has **confidence** c in the transaction set D if c is the percentage of transactions in D containing X that also contain Y. (Computed as P(Y|X))

Practice problem

- Given transaction database, D:

T1 bread, milk, banana, cereal, apple, sugar, flour, butter

T2 cereal, pear, sugar, salt, egg, flour, milk

T3 bread, milk, potato, onion, apple

T4 potato chip, orange juice, coke, ice cream

T5 coke, potato chip, sugar, flour, milk

Assume that : milk → apple is an association rule discovered, then

What is the support for this rule?

What is the confidence of this rule ?

how about {sugar, flour} → egg ?

Mining Frequent Itemsets: the Key Step

- Find the *frequent itemsets*: the sets of items that have minimum support
  - A subset of a frequent itemset must also be a frequent itemset
    - i.e., if {*AB*} is a frequent itemset, both {*A*} and {*B*} should be a frequent itemsets
    - For itemset {ABC}, if {BC} is not frequent, then {ABC} can never be frequent
  - Iteratively find frequent itemsets with cardinality from 1 to *k* *(k*-itemset*)*
- Use the frequent itemsets to generate association rules.

The Apriori Algorithm

- **Join Step**: C_k is generated by joining L_{k-1} with itself
- **Prune Step**: Any (k-1)-itemset that is not frequent cannot be a subset of a frequent k-itemset
- Pseudo-code:

  C_k: Candidate itemset of size k
  L_k : frequent itemset of size k

```
L_1 = {frequent items};
for (k = 1; L_k != ∅; k++) do begin
    C_{k+1} = candidates generated from L_k;
    for each transaction t in database do
        increment the count of all candidates in C_{k+1}
        that are contained in t
    L_{k+1} = candidates in C_{k+1} with min_support
end
return ∪_k L_k;
```

How to Generate Candidates?

- Suppose the items in L_{k-1} are listed in an order
- Step 1: self-joining L_{k-1}

  insert into $\boldsymbol{C_k}$

  select $\boldsymbol{p.item_1, p.item_2, \ldots, p.item_{k-1}, q.item_{k-1}}$

  from $\boldsymbol{L_{k-1}\ p, L_{k-1}\ q}$

  where $\boldsymbol{p.item_1=q.item_1, \ldots, p.item_{k-2}=q.item_{k-2}, p.item_{k-1} < q.item_{k-1}}$

- Step 2: pruning

  forall ***itemsets c in*** $\boldsymbol{C_k}$ do

  forall ***(k-1)-subsets s of c*** do

  if *(s is not in* L_{k-1}*)* **then delete** *c* **from** C_k

Example of Generating Candidates

- $L_3=\{abc, abd, acd, ace, bcd\}$
- Self-joining: L_3*L_3
  - *abcd* from *abc* and *abd*
  - *acde* from *acd* and *ace*
- Pruning:
  - *acde* is removed because *ade* is not in L_3
- $C_4=\{abcd\}$

The Apriori Algorithm — Example

Database D

TID	Items
100	1 3 4
200	2 3 5
300	1 2 3 5
400	2 5

Scan D → C_1

itemset	sup.
{1}	2
{2}	3
{3}	3
{4}	1
{5}	3

→ L_1

itemset	sup.
{1}	2
{2}	3
{3}	3
{5}	3

C_2

itemset
{1 2}
{1 3}
{1 5}
{2 3}
{2 5}
{3 5}

Scan D → C_2

itemset	sup
{1 2}	1
{1 3}	2
{1 5}	1
{2 3}	2
{2 5}	3
{3 5}	2

→ L_2

itemset	sup
{1 3}	2
{2 3}	2
{2 5}	3
{3 5}	2

C_3

itemset
{2 3 5}

Scan D → L_3

itemset	sup
{2 3 5}	2

Example

TID	List of items
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Given the transaction data, find all frequent item sets having minimum support count = 2

Step Two: Generate strong association rules from the frequent itemsets

$$\text{confidence}(X \rightarrow Y) = \frac{\text{support\_count}(X \cup Y)}{\text{support\_count}(X)}$$

support_count(X U Y) : number of transactions containing the itemsets X and Y, and

support_count(X) is the number of transactions containing the itemset X.

- **The basic idea**: given a frequent itemset l, generate all associate rules based on l

- **Direct approach:**
  - for each frequent itemset l, generate all nonempty subsets of l
  - for every nonempty subset s of l, output the rule "s → (l-s) " if

    support_count(l) / support_count(s) >= minimum confidence

Given : a is a subset of l, a' is a subset of a

support (l) <= support (a) <= support(a')

confidence (a → (l - a)) = support(l) / support (a)

>=

confidence (a' → (l - a')) = support (l) / support (a')

Similarly,

Confidence of ((l - a) → a) = support(l) / support (l-a)

<=

confidence of ((l-a') → a') = support(l) / support(l-a')

Therefore, If (l-a') →a' is not a strong rule, then none of the rules of the form (1-a) →a can be strong

- **The apriori approach for rule generation**
  - Basis: If (l-a') →a' is not a strong rule, then none of the rules of the form (1-a) →a can be strong, (a' is a subset of a)
  - Approach :
    - Start by generating rules that have a single consequent
    - Increase size of consequent to 2 by the "ap_gen" function, based on only the successful single consequents
    - Continue to increase the number of consequents, (equivalently, decreasing the size of the antecedent)…, one item at a time, until the antecedent becomes empty

Rule generation algorithm:

for all large k-itemsets l_k, k>= 2 do

begin

H_1 = {consequences of rules from l_k with one item in the consequent };

Call ap-genrules(l_k, H_1);

end

- Use Apriori-gen to create the consequents in rules:
  - Join: form the consequent part of the rule, with successively larger sizes
  - Prune: eliminate no-hope candidates

Procedure **ap-genrules** (l_k: large k-itemset,

H_m: set of m-item consequents)

If (k > m+1) then begin

H_{m+1} = apriori-gen(H_m);

Forall h_{m+1} belongs to H_{m+1} do begin

Conf = support(l_k)/support(l_k- h_{m+1});

If (Conf >= minimum_confidence) then

Output the rule (l_k - h_{m+1}) → h_{m+1} With

confidence = Conf, support = support(l_k);

Else

Delete h_{m+1} from H_{m+1}; // why?

end

Call ap-genrules(l_k, H_{m+1});

end

end

Example

TID	List of items
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Given the transaction data, find all association rules having minimum support count = 2, and minimum confidence of 70%.

Example

TID	List of items
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Given the transaction data, find all association rules having minimum support count = 2, and minimum confidence of 70%.

We already derived frequent itemsets for this TD as:
{{I1}, {I2}, {I3}, {I4}, {I5},
{I1, I2}, {I1, I3}, {I1, I5},
{I2, I3}, {I2, I4}, {I2, I5}
{I1, I2, I3}, {I1, I2, I5}}

Suppose l_k = {I1, I2, I5}

Implementation of Apriori

- Hash tree
- Hash tree is used for two steps of Apriori
  - Test whether "subset s of candidate itemset is in L_{k-1}"
  - Test whether "a candidate itemset C_k is in a transaction t"

How to Count Supports of Candidates?

- Why counting supports of candidates in a problem?
  - The total number of candidates can be huge
  - One transaction may contain many candidates
- Method:
  - Candidate itemsets are stored in a *hash-tree*
  - *Leaf* node of hash-tree contains a list of itemsets and counts
  - *Interior* node contains a hash table
  - *Subset function*: finds all the candidates contained in a transaction

Subset(candidate itemset, L_{k-1})

- Goal : check "is there any (k-1) subset of c that is not in L_{k-1} ?
- Approach:
  - All items in L_{k-1} are stored in a hash tree
  - For each non-empty (k-1) subsets of a k-itemset,

    checking whether a (k-1) subset is in L_{k-1} takes O(1)

Subset (c_k, t)

- Assumption:
  - Items in transactions are ordered
  - Items in candidate set are ordered
  - Candidate c_k are put in a hash tree
- Approach:
  - At root level, hash on every item in the transaction,
  - At level i,

    if it is an interior node, hash on every item following the ith item,

    if it is a leaf node, check if the candidate c is in the list

        if yes, update the counter for that candidate

Is Apriori Fast Enough? — Performance Bottlenecks

- The core of the Apriori algorithm:
  - Use frequent (k – 1)-itemsets to generate candidate frequent k-itemsets
  - Use database scan and pattern matching to collect counts for the candidate itemsets
- The bottleneck of *Apriori*: candidate generation
  - Huge candidate sets:
    - 10^4 frequent 1-itemset will generate 10^7 candidate 2-itemsets
    - To discover a frequent pattern of size 100, e.g., $\{a_1, a_2, \ldots, a_{100}\}$, one needs to generate $2^{100} \approx 10^{30}$ candidates.
  - Multiple scans of database:
    - Needs ($n + 1$) scans, n is the length of the longest pattern

Methods to Improve Apriori's Efficiency

- Hash-based itemset counting: A *k*-itemset whose corresponding hashing bucket count is below the threshold cannot be frequent
- Transaction reduction: A transaction that does not contain any frequent k-itemset is useless in subsequent scans
- Partitioning: Any itemset that is potentially frequent in DB must be frequent in at least one of the partitions of DB
- Sampling: mining on a subset of given data, lower support threshold + a method to determine the completeness
- Dynamic itemset counting: add new candidate itemsets only when all of their subsets are estimated to be frequent

AproriTid–Transaction reduction method

- Objectives: as the size of the frequent itemsets increases,
  - reduce the length of each transaction
  - reduce the number of transactions necessary to check for support
- Method:
  - Database D is not used for counting support after the first pass.
  - The set $\hat{C}_k$ <Tid, $\{X_k\}$> is used for counting support afterwards, where $\{X_k\}$ is a potentially large k-itemset present in the transaction with identifier Tid.

Example

TID	items
T1	I1, I2, I3, I5
T2	I2, I4
T3	I2, I6
T4	I1, I2, I4, I5
T5	I1, I2
T6	I1, I2, I3, I5
T7	I1, I2, I3

AprioriTid Algorithm

L_1 = {large 1-itemsets}; $\hat{C}_1$ = database D;

For (k=2; L_{k-1}!=0; k++) **do begin**

C_k = apriori-gen (L_{k-1}) ; // New Candidate

$\hat{C}_k$ = 0;

Forall entries t in $\hat{C}_{k-1}$ **do begin**

// determine candidates contained in the transaction t.TID

C_t = {c in C_k | (c[1],c[2], ..., c[k-2], c[k-1] in $\hat{C}_{k-1}$) and (c[1], c[2], ..., c[k-2], c[k]) in $\hat{C}_{k-1}$)};

Forall candidates c in C_t **do** c.count ++;

if (C_t != 0) **then** $\hat{C}_k$ += < t.TID, C_t> // add Ct to $\hat{C}_k$

End

L_k = {c in C_k | c.count >= minimum_support_count}

Answer = Answer union L_k;

End

Association Rule Discovery:

Part Two:

Mining Frequent Patterns without Candidate Generation

The FP-Growth Approach

Construction of FP-tree

- Step 1: scan the transaction database D once. Collect the set of frequent items F and their supports. Sort F in support descending order as L, the list of frequent items.
- Step 2: create the root of an FP-tree, and label it as "null". For each transaction T in D do the following:
  - select and sort the frequent items in T according to the order of L. Let the sorted frequent item list in T be [p|P], where p is the first element and P is the remaining list.

Call **insert_tree**([p|P], T), which is performed as follows:

- if T has a child N such that *N.item-name = p.item-name*, then increment N’s count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same item-name via the node-link structure. If P is non-empty, call **insert-tree**(P, N) recursively.

Construction of FP-tree from a Transaction Database (An example)

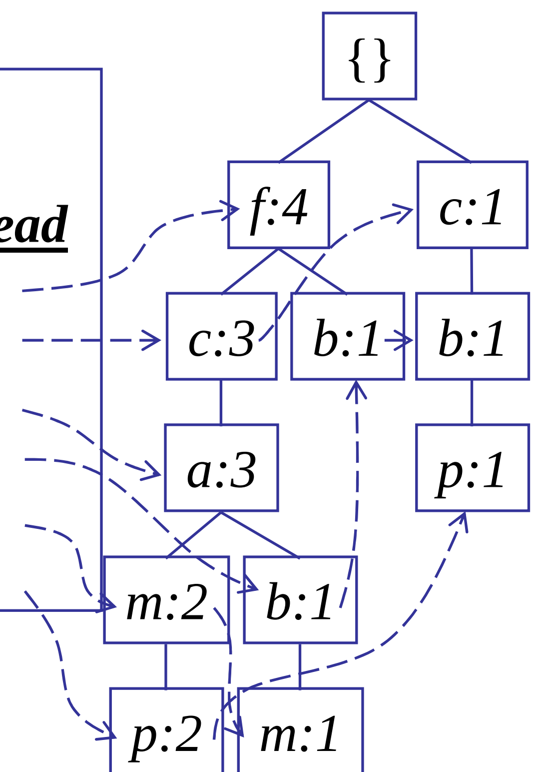
TID	***Items bought***	***(ordered) frequent items***
100	***{f, a, c, d, g, i, m, p}***	***{f, c, a, m, p}***
200	***{a, b, c, f, l, m, o}***	***{f, c, a, b, m}***
300	***{b, f, h, j, o, w}***	***{f, b}***
400	***{b, c, k, s, p}***	***{c, b, p}***
500	***{a, f, c, e, l, p, m, n}***	***{f, c, a, m, p}***

min_support_count=3

1. Scan DB once, find frequent 1-itemset (single item pattern)
2. Order frequent items in frequency descending order
3. Scan DB again, construct FP-tree

Header Table

Item	*frequency*	*head*
f	*4*	
c	*4*	
a	*3*	
b	*3*	
m	*3*	
p	*3*	



Benefits of the FP-tree Structure

- Completeness
  - Never breaks a long pattern of any transaction
  - Preserves complete information for frequent pattern mining
- Compactness
  - Reducing irrelevant info—infrequent items are gone
  - Items in frequency descending order: the more frequently occurring, the more likely to be shared
  - Never be larger than the original database (not count node-links and the *count* field)
  - The compression ratio could be 20 ~ 100

Mining Frequent Patterns with FP-trees

- Idea: Recursively grow frequent patterns by pattern and database partition
- Terminology:
  - *prefix path (P) of an item a_i*: a path from the root of the FP-tree to the parent node of a_i
  - *prefix paths (Ps) of an item a_i in a FP-tree*: all prefix paths of a_i in the FP-tree
  - *transformed prefix path of a_i for path P :* a_i's prefix path with the frequency count of every node in P adjusted to the same as the count for a_i
  - *a_i's conditional pattern base :* the set of transformed prefix paths of a_i
  - a_i's conditional FP-tree : the FP-tree built based on a_i's conditional pattern base

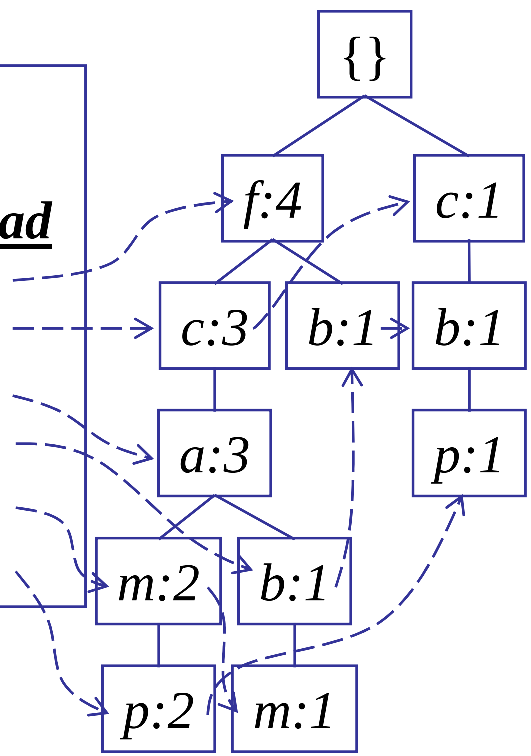
- Frequent pattern growth method
  - For each frequent item, construct its conditional pattern-base, and then its conditional FP-tree
  - Repeat the process on each newly created conditional FP-tree
  - Until the resulting FP-tree is empty, or it contains only one path—single path will generate all the combinations of its sub-paths, each of which is a frequent pattern

From FP-tree to Conditional Pattern-Base

- Starting at the frequent item header table in the FP-tree
- Traverse the FP-tree by following the link of each frequent item *p*
- Accumulate all of *transformed prefix paths* of item *p* to form *p*'s conditional pattern base

Header Table

Item	*frequency*	*head*
f	4	
c	4	
a	3	
b	3	
m	3	
p	3	

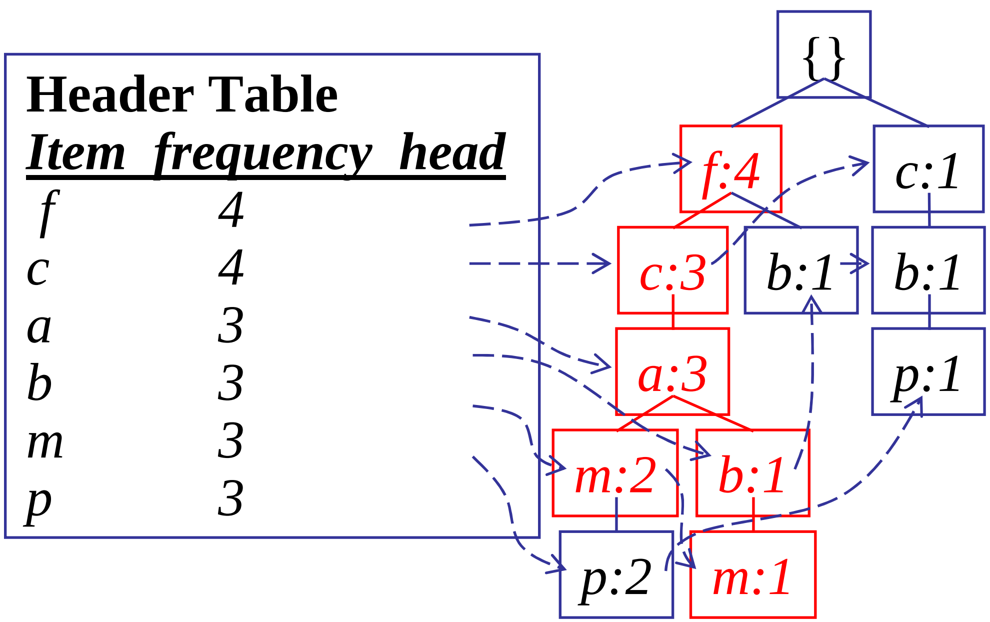


***Conditional* pattern bases**

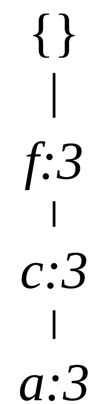
item	*cond. pattern base*
c	***f:3***
a	***fc:3***
b	***fca:1, f:1, c:1***
m	***fca:2, fcab:1***
p	***fcam:2, cb:1***

From Conditional Pattern-Bases to Conditional FP-trees

- For each pattern-base
  - Accumulate the count for each item in the base
  - Construct the FP-tree for the frequent items of the pattern base



m-conditional **pattern base:**
fca:2, fcab:1



m-conditional **FP-tree**

All frequent patterns relate to *m*

m,

fm, cm, am,

fcm, fam, cam,

fcam

Principles of Frequent Pattern Growth

- Pattern growth theorem
  - Let α be a frequent itemset in DB, B be α's conditional pattern base, and β be an itemset in B. Then $\alpha \cup \beta$ is a frequent itemset in DB iff β is frequent in B.
- Ex. "*abcdef*" is a frequent pattern, if and only if
  - "*abcd*" is a frequent pattern, and
  - "*ef*" is frequent in *abcd*s conditional pattern-base (i.e., the set of transactions containing "*abcd*")

Procedure FP-growth(Tree, α)

if Tree contains a single path P then

for each combination (β) of the nodes in P

generate patterns β union α w/ support = min_support of nodes in β

else for each a_i in the header of Tree {

generate pattern $\beta = a_i$ union α with

support=a_i.support

construct β`s conditional pattern base and then $\beta$`s conditional FP_tree Tree $_\beta$

if Tree $_\beta$!= 0 then

call **FP_growth**(Tree $_\beta$, β)

}

Practice example

Apply FP-Growth algorithm to find all frequent itemsets from the following transaction database:

TID	list of items
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Correctness and Completeness of Conditional Pattern-Bases

- Completeness (node-link property)
  - For any frequent item a_i, all the possible frequent patterns that contain a_i can be obtained by following a_i's node-links, starting from a_i's head in the FP-tree header
- Correctness (transformed prefix path property)
  - To calculate the frequent patterns for a node a_i in a path P, only the prefix sub-path of a_i in P need to be accumulated, and its frequency count should carry the same count as node a_i.

A Special Case: Single FP-tree Path

- Suppose a (conditional) FP-tree T has a single path P
- The complete set of frequent patterns of T can be generated by enumeration of all the combinations of the sub-paths of P

{}
|
f:3
|
c:3
|
a:3

➔

All frequent patterns concerning *m*

m,

fm, cm, am,

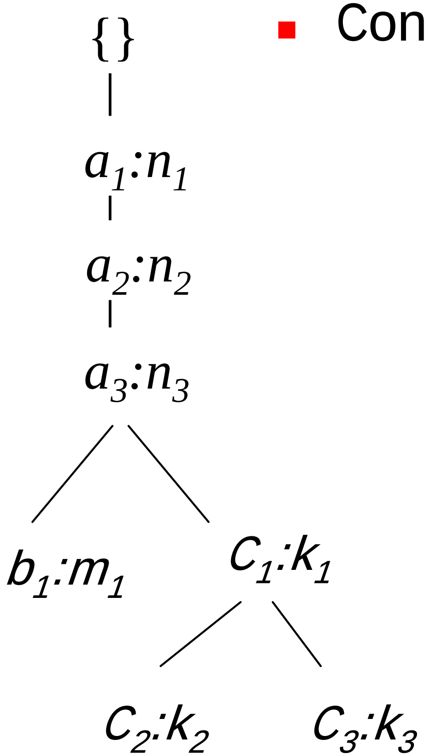
fcm, fam, cam,

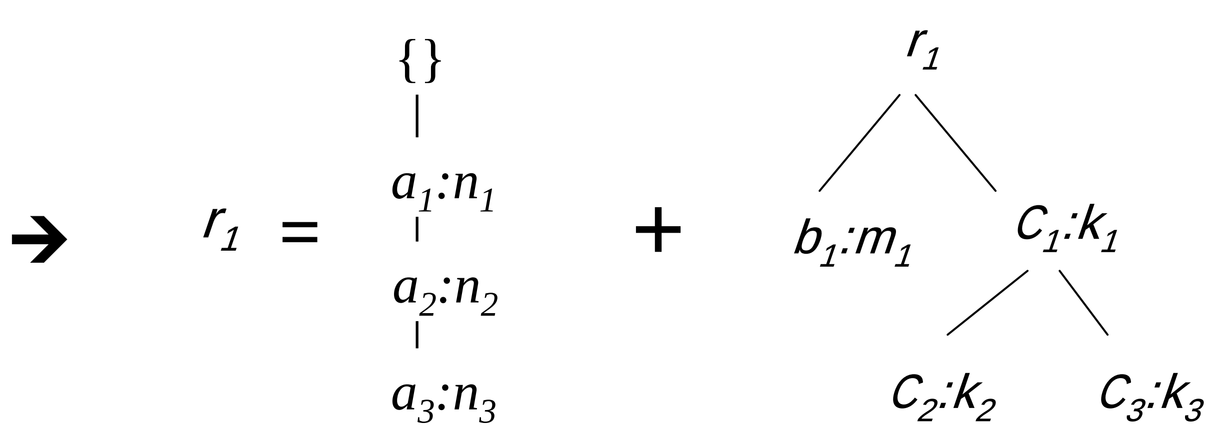
fcam

***m-conditional* FP-tree**

A More General (Special) Case: Single Prefix Path in FP-tree

- Suppose a (conditional) FP-tree T has a shared single prefix-path P
- Mining can be decomposed into two parts
  - Reduction of the single prefix path into one node
  - Concatenation of the mining results of the two parts

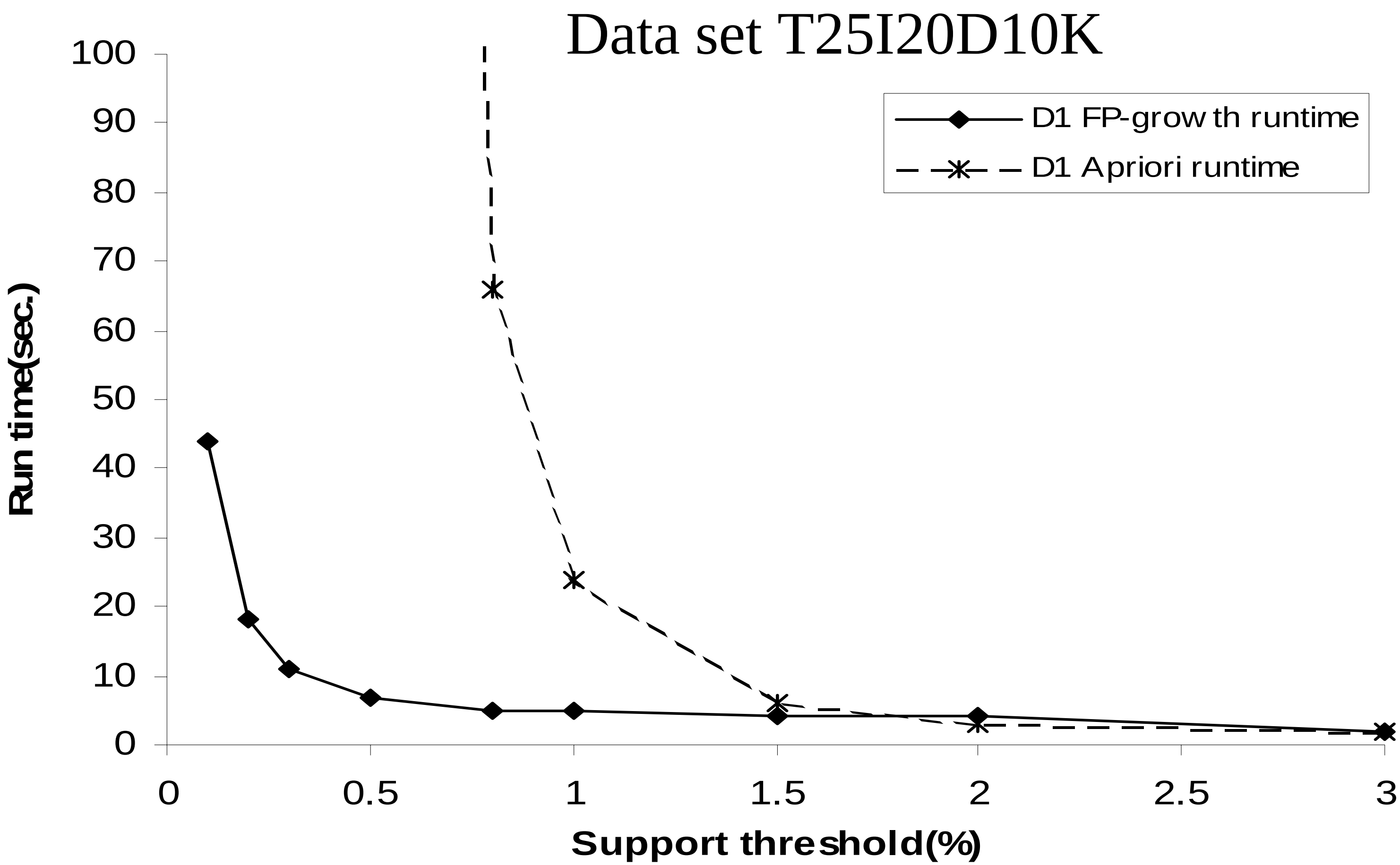




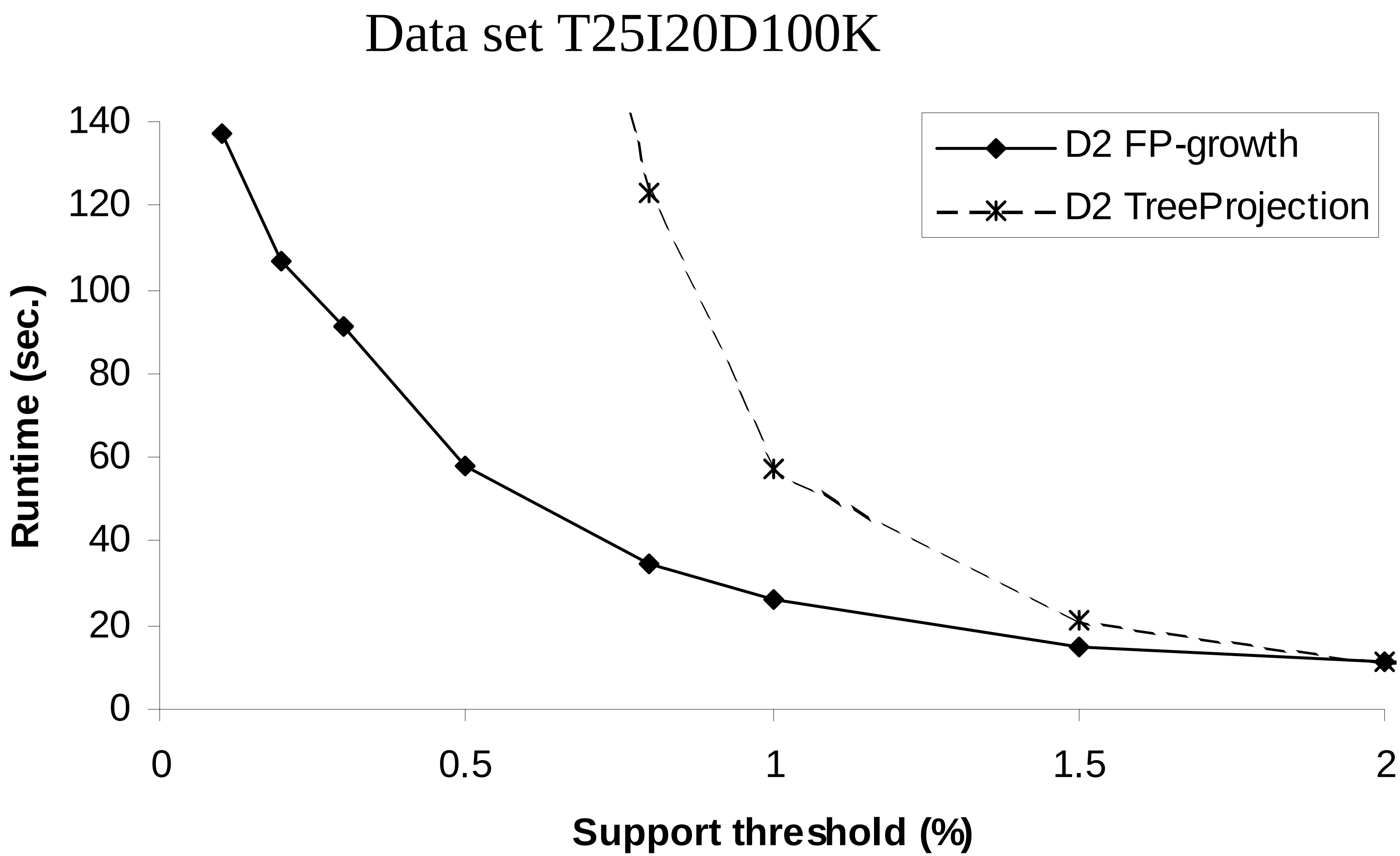
Why Is FP-Growth the Winner?

- Divide-and-conquer:
  - decompose both the mining task and DB according to the frequent patterns obtained so far
  - leads to focused search of smaller databases
- Other factors
  - no candidate generation, no candidate test
  - compressed database: FP-tree structure
  - no repeated scan of entire database
  - basic ops—counting and FP-tree building, not pattern search and matching

FP-growth vs. Apriori: Scalability With the Support Threshold



FP-growth vs. Tree-Projection: Scalability with Support Threshold



Summary

- Association rule mining
  - probably the most significant contribution from the database community in KDD
  - A large number of papers have been published
- Many interesting issues have been explored
- An interesting research direction
  - Association analysis in other types of data: spatial data, multimedia data, time series data, etc.