Mining Frequent Patterns: Apriori v.s. FP-Growth

Ling Liu

Slides are adapted from Mining Frequent Patterns without Candidate Generation (SIGMOD200, Pei and Han), The FP-Growth/Apriori Debate by Jeffrey R. Ellis

Outline

- Frequent Pattern Mining: Problem statement and an example
- Review of Apriori-like Approaches
- FP-Growth:
  - Overview
  - FP-tree:
    - structure, construction and advantages
  - FP-growth:
    - FP-tree →conditional pattern bases → conditional FP-tree →frequent patterns
- Performance Comparison

Frequent Patterns

- Frequent pattern: a pattern (a set of items, subsequences, substructures, etc.) that occurs frequently in a data set
  - itemset: A set of one or more items
  - k-itemset: $X = \{x_1, \ldots, x_k\}$
  - Mining algorithms
    - Apriori
    - FP-growth

Tid	Items bought
10	Beer, Nuts, Diaper
20	Beer, Coffee, Diaper
30	Beer, Diaper, Eggs
40	Nuts, Eggs, Milk
50	Nuts, Coffee, Diaper, Eggs, Beer

Support & Confidence

- Support
  - (absolute) support, or, support count of X: Frequency or occurrence of an itemset X
  - (relative) support, s, is the fraction of transactions that contains X (i.e., the probability that a transaction contains X)
  - An itemset X is frequent if X's support is no less than a minsup threshold
- Confidence (association rule: X→Y)
  - $sup(X \cup Y)/sup(x)$ (conditional prob.: $Pr(Y|X) = Pr(X \wedge Y)/Pr(X)$)
  - confidence, c, conditional probability that a transaction having X also contains Y
  - Find all the rules X→Y with minimum support and confidence
    - $sup(X \cup Y) \geq minsup$
    - $sup(X \cup Y)/sup(X) \geq minconf$

Frequent Pattern Mining: An Example

Given a transaction database DB and a minimum support threshold ξ, find all frequent patterns (item sets) with support no less than ξ.

Input: DB:

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

Minimum support: ξ =3

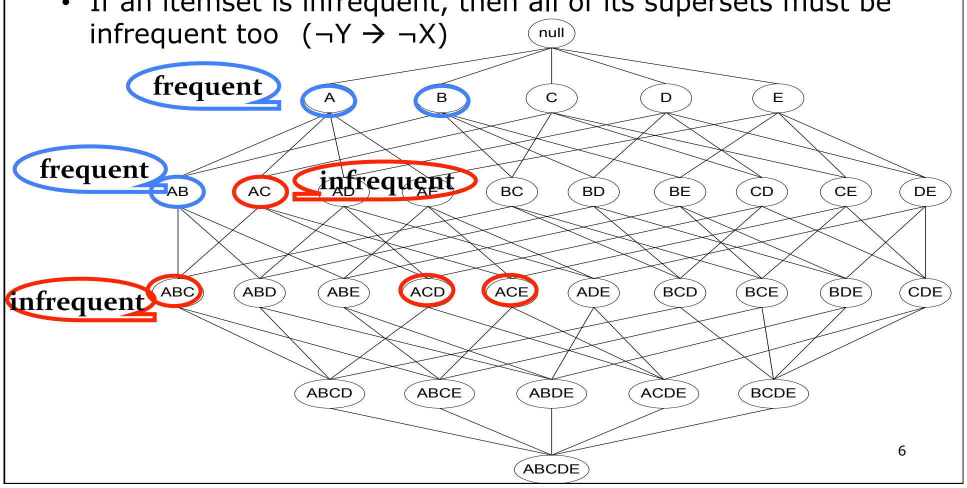
Output: all frequent patterns, i.e., *f, a, …, fa, fac, fam, fm,am…*

Problem Statement: How to **efficiently** find all frequent patterns?

5

Apriori Principle

- If an itemset is frequent, then all of its subsets must also be frequent (X → Y)
- If an itemset is infrequent, then all of its supersets must be infrequent too (¬Y → ¬X)



6

Apriori

- Main Steps of Apriori Algorithm:
  - Use frequent (k – 1)-itemsets (L_{k-1}) to generate candidates of frequent k-itemsets C_k
  - Scan database and count each pattern in C_k , get frequent k-itemsets (L_k) .

Candidate Generation

Candidate Test

- E.g. ,

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

Apriori	*iteration*
C1	*f,a,c,d,g,i,m,p,l,o,h,j,k,s,b,e,n*
L1	*f, a, c, m, b, p*
C2	*fa, fc, fm, fp, ac, am, ...bp*
L2	*fa, fc, fm, ...*
...	

7

Apriori: A Candidate Generation & Test Approach

- Initially, scan DB once to get frequent 1-itemset
- Loop
  - Generate length (k+1) candidate itemsets from length k frequent itemsets
  - Test the candidates against DB
  - Terminate when no frequent or candidate set can be generated

8

Generate candidate itemsets

- **Example**

  Frequent 3-itemsets:

  {1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 3, 4}, {1, 3, 5}, {2, 3, 4}, {2, 3, 5} and {3, 4, 5}

  - Candidate 4-itemset:

    {1, 2, 3, 4}, {1, 2, 3, 5}, {1, 2, 4, 5}, {1, 3, 4, 5}, {2, 3, 4, 5}

  - Which need not to be counted?

    {1, 2, 4, 5} & {1, 3, 4, 5} & {2, 3, 4, 5}

9

Mining Frequent Itemsets without Candidate Generation

- In many cases, the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain.

- However, it suffer from two nontrivial costs:
  - It may generate a huge number of candidates (for example, if we have 10^4 1-itemset, it may generate more than 10^7 candidata 2-itemset)
  - It may need to scan database many times

Association Rules with Apriori

Minimum support=2/9

Minimum confidence=70%

Scan D for count of each candidate →

C_1

Itemset	Sup. count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

Compare candidate support count with minimum support count →

L_1

Itemset	Sup. count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

Generate C_2 candidates from L_1 →

C_2

Itemset
{I1, I2}
{I1, I3}
{I1, I4}
{I1, I5}
{I2, I3}
{I2, I4}
{I2, I5}
{I3, I4}
{I3, I5}
{I4, I5}

Scan D for count of each candidate →

C_2

Itemset	Sup. count
{I1, I2}	4
{I1, I3}	4
{I1, I4}	1
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2
{I3, I4}	0
{I3, I5}	1
{I4, I5}	0

Compare candidate support count with minimum support count →

L_2

Itemset	Sup. count
{I1, I2}	4
{I1, I3}	4
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2

Generate C_3 candidates from L_2 →

C_3

Itemset
{I1, I2, I3}
{I1, I2, I5}

Scan D for count of each candidate →

C_3

Itemset	Sup. count
{I1, I2, I3}	2
{I1, I2, I5}	2

Compare candidate support count with minimum support count →

L_3

Itemset	Sup. count
{I1, I2, I3}	2
{I1, I2, I5}	2

Performance Bottlenecks of Apriori

- Bottlenecks of *Apriori*: candidate generation
  - Generate 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.
  - Candidate Test incur multiple scans of database: each candidate

12

Bottleneck of Frequent-pattern Mining

- Multiple database scans are costly
- Mining long patterns needs many passes of scanning and generates lots of candidates
  - To find frequent itemset $i_1i_2\ldots i_{100}$
    - # of scans: 100
    - # of Candidates: $\binom{100}{1} + \binom{100}{2} + \ldots + \binom{100}{100} = 2^{100}-1 = 1.27*10^{30}$!
- **Bottleneck:** candidate-generation-and-test
- **Can we avoid candidate generation?**

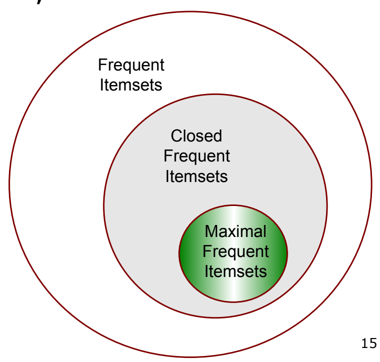
Example Alternatives to Apriori

- FP-Growth
  - Finding frequent itemsets without candidate generation
- CHARM
  - Based on concept of Closed Itemset
  - CLOSET
    - Han, Pei implementation of Closed Itemset

Maximal vs Closed Frequent Itemsets

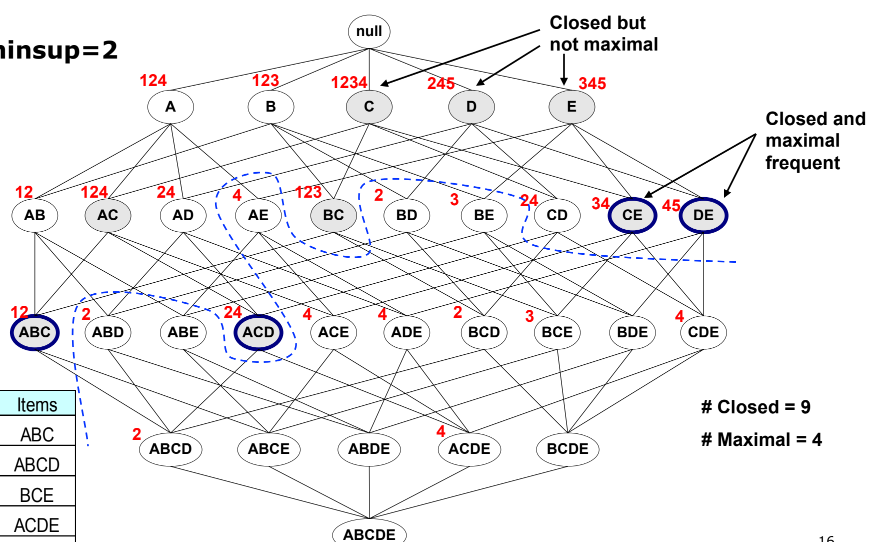
- An itemset X is a max-pattern if X is frequent and there exists no frequent super-pattern Y ɔ X
- An itemset X is closed if X is frequent and there exists no super-pattern Y ɔ X, with the same support as X

Closed Frequent Itemsets are Lossless: the support for any frequent itemset can be deduced from the closed frequent itemsets



Maximal vs Closed Frequent Itemsets

minsup=2



TID	Items
1	ABC
2	ABCD
3	BCE
4	ACDE
5	DE

Algorithms to find frequent pattern

- **Apriori**: uses a generate-and-test approach – generates candidate itemsets and tests if they are frequent
  - Generation of candidate itemsets is expensive (in both space and time)
  - Support counting is expensive
    - Subset checking (computationally expensive)
    - Multiple Database scans (I/O)
- **FP-Growth**: allows frequent itemset discovery without candidate generation. Two step:
  - 1.Build a compact data structure called the FP-tree
    - 2 passes over the database
  - 2.extracts frequent itemsets directly from the FP-tree
    - Traverse through FP-tree

17

FP-Growth Algorithm

- Association Rule Mining
  - Generate Frequent Itemsets
    - Apriori generates candidate sets
    - FP-Growth uses specialized data structures (no candidate sets)
  - Find Association Rules
    - Outside the scope of both FP-Growth & Apriori
- Therefore, FP-Growth is a competitor to Apriori

Overview of FP-Growth: Ideas

- Compress a large database into a compact, *Frequent-Pattern tree* (FP-tree) structure
  - highly compacted, but complete for frequent pattern mining
  - avoid costly repeated database scans
- Develop an efficient, FP-tree-based frequent pattern mining method (FP-growth)
  - A divide-and-conquer methodology: decompose mining tasks into smaller ones
  - Avoid candidate generation: sub-database test only.

FP-tree:
Construction and Design

Mining Frequent Patterns without Candidate Generation (SIGMOD2000)

Construct FP-tree

FP-Tree is constructed using 2 passes over the data-set:

Two Steps:

1. Scan the transaction DB for the first time, find frequent items (single item patterns) and order them into a list L in frequency descending order.

   e.g., L={f:4, c:4, a:3, b:3, m:3, p:3}

   In the format of (item-name, support)

2. For each transaction, order its frequent items according to the order in L; Scan DB the second time, construct FP-tree by putting each frequency ordered transaction onto it.

FP-tree Example: step 1

Step 1: Scan DB for the first time to generate L

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

L

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

By-Product of First Scan of Database

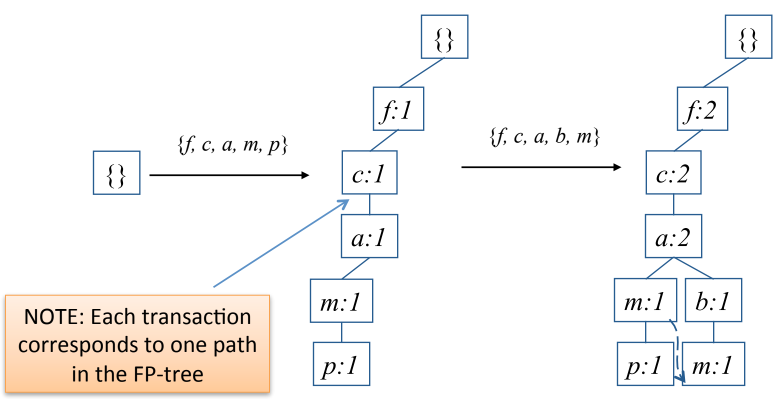
FP-tree Example: step 2

Step 2: scan the DB for the second time, order frequent items in each transaction

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*}	{*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*}

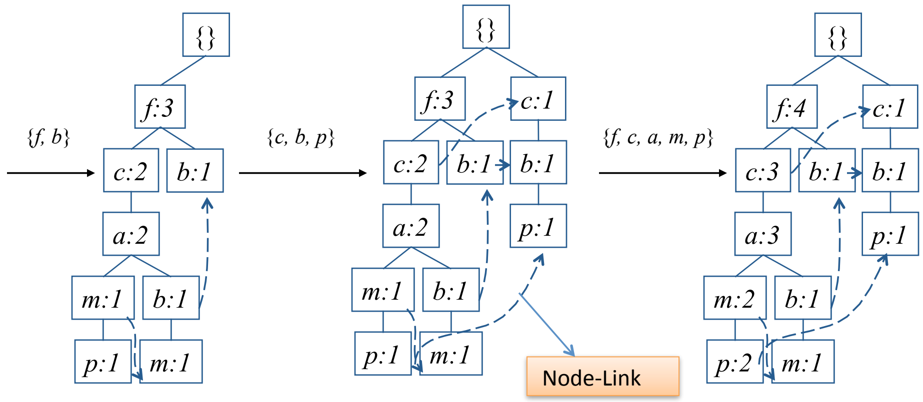
FP-tree Example: step 2

Step 2: construct FP-tree



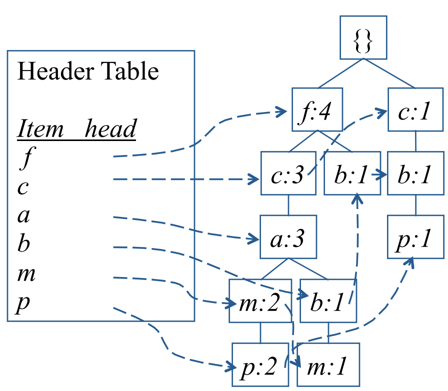
FP-tree Example: step 2

Step 2: construct FP-tree



Construction Example

Final FP-tree



FP-Tree Definition

- FP-tree is **a frequent pattern tree** . Formally, FP-tree is a tree structure defined below:

  1. One root labeled as “null", a set of *item prefix sub-trees* as the children of the root, and a *frequent-item header table*.

  2. Each node in *the item prefix sub-trees* has three fields
     - item-name : register which item this node represents,
     - count, the number of transactions represented by the portion of the path reaching this node,
     - node-link that links to the next node in the FP-tree carrying the same item-name, or null if there is none.

  3. Each entry in the *frequent-item header table* has two fields,
     - item-name, and
     - head of node-link that points to the first node in the FP-tree carrying the item-name.

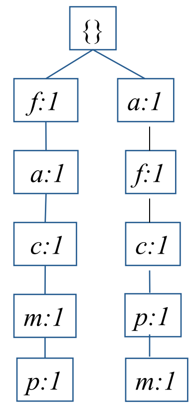
Advantages of the FP-tree Structure

- The most significant advantage of the FP-tree
  - Scan the DB only twice and twice only.
- Completeness:
  - the FP-tree contains all the information related to mining frequent patterns (given the min-support threshold). Why?
- Compactness:
  - The size of the tree is bounded by the occurrences of frequent items
  - The height of the tree is bounded by the maximum number of items in a transaction

Questions?

- Why descending order?
- Example 1:

TID	*(unordered) frequent items*
100	{*f, a, c, m, p*}
500	{*a, f, c, p, m*}

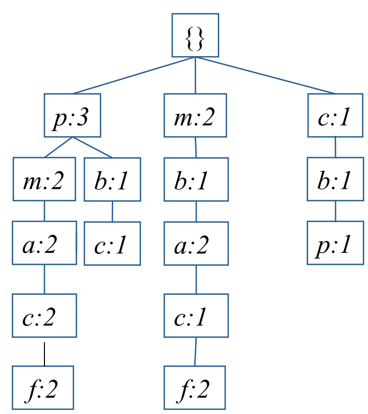


Questions?

- Example 2:

TID	*(ascended) frequent items*
100	{*p, m, a, c, f*}
200	{*m, b, a, c, f*}
300	{*b, f*}
400	{*p, b, c*}
500	{*p, m, a, c, f*}

This tree is larger than FP-tree, because in FP-tree, more frequent items have a higher position, which makes branches less



FP-Growth Algorithm

FP-CreateTree

Input: DB, min_support

Output: FP-Tree

1. Scan DB & count all frequent items.
2. Create null root & set as current node.
3. For each Transaction T
   – Sort T' s items.
   – For each sorted Item I
     • Insert I into tree as a child of current node.
     • Connect new tree node to header list.

- Two passes through DB
- Tree creation is based on number of items in DB.
- Complexity of CreateTree is O(|DB|)

FP-growth:
Mining Frequent Patterns Using FP-tree

Mining Frequent Patterns Using FP-tree

- General idea (divide-and-conquer)
  Recursively grow frequent patterns using the FP-tree: looking for shorter ones recursively and then concatenating the suffix:
  - 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)

Three Major Steps

Starting the processing from the end of list L:

Step 1:

Construct **conditional pattern base** for each item in the header table

Step 2

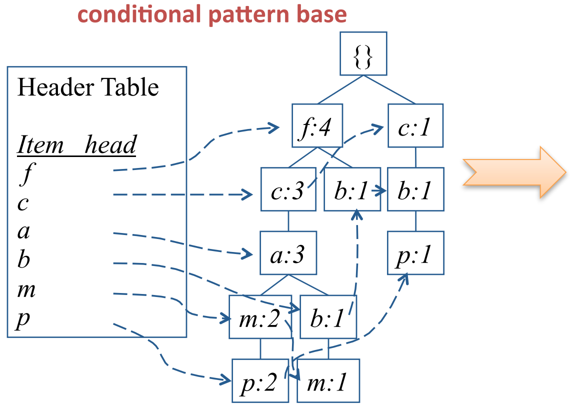
Construct **conditional FP-tree** from each conditional pattern base

Step 3

Recursively mine conditional FP-trees and grow frequent patterns obtained so far. If the conditional FP-tree contains a single path, simply enumerate all the patterns

Step 1: Construct Conditional Pattern Base

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



***Conditional* pattern bases**

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

Properties of FP-Tree

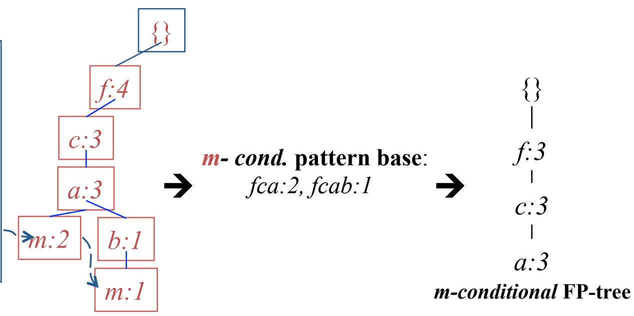
- Node-link property
  – For any frequent item a_i, all the possible frequent patterns that contain a_i can be obtained by following a_i's node-links, starting from a_i's head in the FP-tree header.
- Prefix path property
  – To calculate the frequent patterns for a node a_i in a path P, only the prefix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

Step 2: Construct Conditional FP-tree

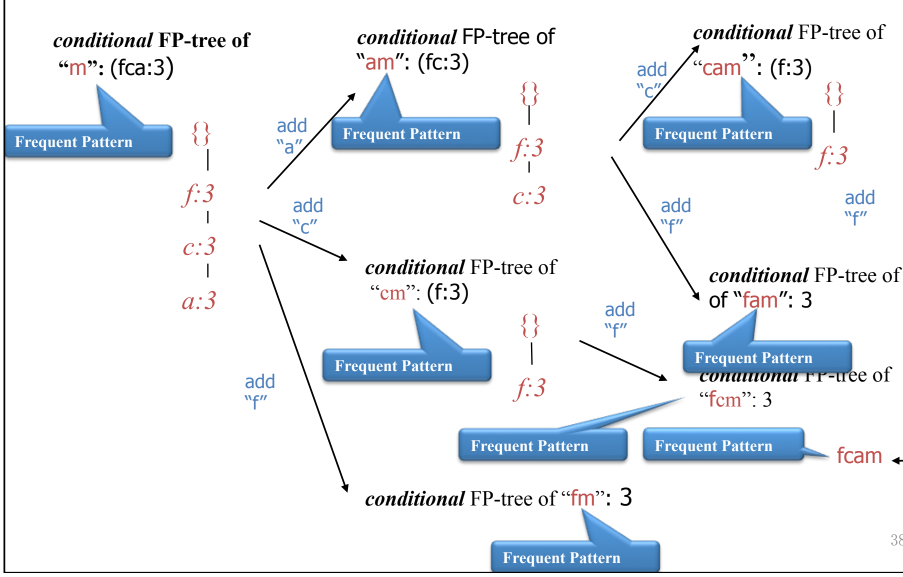
- For each pattern base
  - Accumulate the count for each item in the base
  - Construct the conditional FP-tree for the frequent items of the pattern base

Header Table

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



Step 3: Recursively mine the conditional FP-tree



Principles of FP-Growth

- Pattern growth property
  - 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.
- Is "*fcabm* " a frequent pattern?
  - "fcab" is a branch of m's conditional pattern base
  - "b" is **NOT** frequent in transactions containing "fcab "
  - "bm" is **NOT** a frequent itemset.

Conditional Pattern Bases and Conditional FP-Tree

Item	Conditional pattern base	Conditional FP-tree
p	{(fcam:2), (cb:1)}	{(c:3)}|p
m	{(fca:2), (fcab:1)}	{(f:3, c:3, a:3)}|m
b	{(fca:1), (f:1), (c:1)}	Empty
a	{(fc:3)}	{(f:3, c:3)}|a
c	{(f:3)}	{(f:3)}|c
f	Empty	Empty

order of L

40

Single FP-tree Path Generation

- Suppose an FP-tree T has a single path P. The complete set of frequent pattern of T can be generated by enumeration of all the combinations of the sub-paths of P

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

m-conditional FP-tree

➔

All frequent patterns concerning *m*: combination of {f, c, a} and *m*

m,

fm, cm, am,

fcm, fam, cam,

fcam

Summary of FP-Growth Algorithm

- Mining frequent patterns can be viewed as first mining 1-itemset and progressively growing each 1-itemset by mining on its conditional pattern base recursively

- Transform a frequent k-itemset mining problem into a sequence of k frequent 1-itemset mining problems via a set of conditional pattern bases

Efficiency Analysis

Facts: usually

1. FP-tree is much smaller than the size of the DB
2. Pattern base is smaller than original FP-tree
3. Conditional FP-tree is smaller than pattern base

→ mining process works on a set of usually much smaller pattern bases and conditional FP-trees

→ Divide-and-conquer and dramatic scale of shrinking

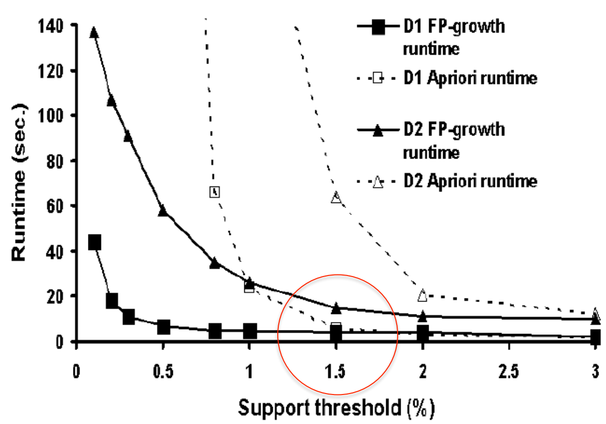
Experiments:
Performance Evaluation

Mining Frequent Patterns without Candidate Generation (SIGMOD2000)

Experiment Setup

- Compare the runtime of FP-growth with classical Apriori
- Runtime vs. min_sup
  - Runtime per itemset vs. min_sup
  - Runtime vs. size of the DB (# of transactions)
- Synthetic data sets : frequent itemsets grows exponentially as minisup goes down
  - D1: T25.I10.D10K
    - 1K items
    - avg(transaction size)=25
    - avg(max/potential frequent item size)=10
    - 10K transactions
  - D2: T25.I20.D100K
    - 10k items

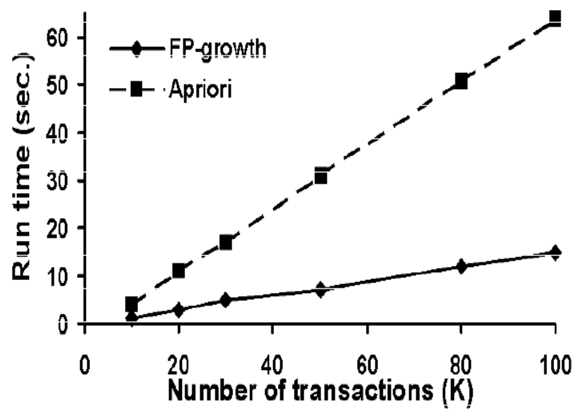
Scalability: runtime vs. min_sup (w/ Apriori)



Mining Frequent Patterns without Candidate Generation (SIGMOD2000)

46

Scalability: runtime vs. # of Trans. (w/ Apriori)



* Using D2 and min_support=1.5%

Mining Frequent Patterns without Candidate Generation (SIGMOD2000)

47

Conclusion Remarks

- FP-tree: a novel data structure storing compressed, crucial information about frequent patterns, compact yet complete for frequent pattern mining.
- FP-growth: an efficient mining method of frequent patterns in large Database: using a highly compact FP-tree, divide-and-conquer method in nature.

Improvements to FP-Growth

- None currently reported
- Ideas
  - New algorithm that is based on FP-Growth
  - Distributes FP-Trees among processors
- No reports of complexity analysis or accuracy of FP-Growth

The debate between Apriori and FP-growth

Zheng, Kohavi, Mason – "Real World Performance of Association Rule Algorithms"

Algorithm Analysis Results

- FP-Growth IS NOT inherently faster than Apriori
  - Intuitively, it appears to condense data
  - Mining scheme requires some new work to replace candidate set generation
  - Recursion obscures the additional effort
- FP-Growth may run faster than Apriori in circumstances
- No guarantee through complexity which algorithm to use for efficiency

Real World Applications

- Zheng, Kohavi, Mason – "Real World Performance of Association Rule Algorithms"
  - Collected implementations of Apriori, FP-Growth, CLOSET, CHARM, MagnumOpus
  - Tested implementations against 1 artificial and 3 real data sets
  - Time-based comparisons generated

Apriori & FP-Growth

- Apriori
  - Implementation from creator Christian Borgelt (GNU Public License)
  - C implementation
  - **Entire dataset loaded into memory**
- FP-Growth
  - Implementation from creators Han & Pei
  - Version – February 5, 2001

Other Algorithms

- CHARM
  - Based on concept of Closed Itemset
  - e.g., { A, B, C } – AB→C, A→C, B→C, AC→B, A→B, C→B, etc.
- CLOSET
  - Han, Pei implementation of Closed Itemset
- MagnumOpus
  - Generates rules directly through search-and-prune technique

Four Datasets

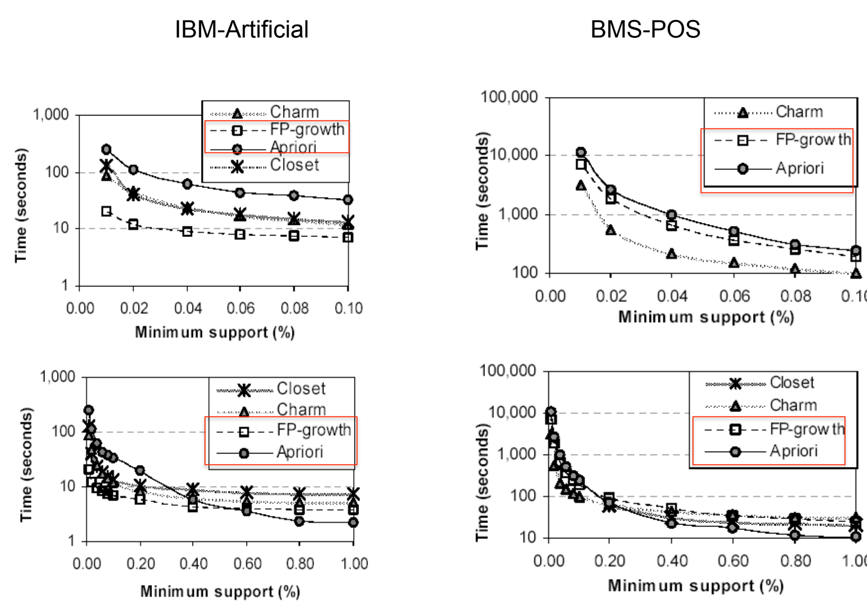
- IBM-Artificial
  - Generated at IBM Almaden (T10I4D100K)
  - Often used in association rule mining studies
- BMS-POS
  - Years of point-of-sale data from retailer
- BMS-WebView-1 & BMS-WebView-2
  - Months of clickstream traffic from e-commerce web sites

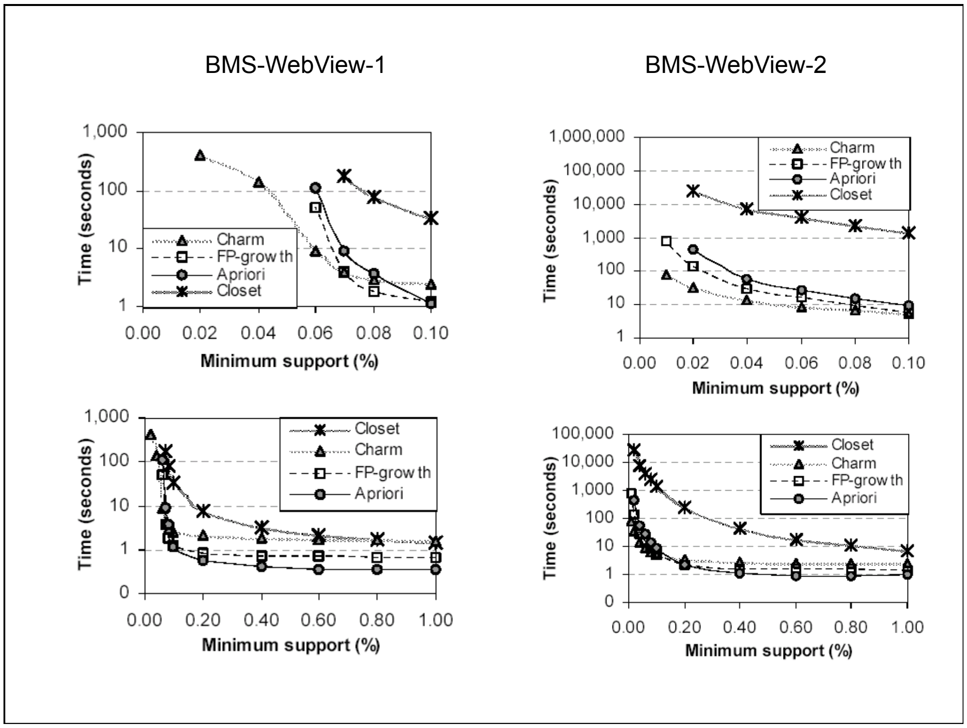
Dataset Characteristics

	Transac-tions	Distinct Items	Maximum Trans. Size	Average Trans. Size
IBM-Artificial	100,000	870	29	10.1
BMS-POS	515,597	1,657	164	6.5
BMS-WebView-1	59,602	497	267	2.5
BMS-WebView-2	77,512	3,340	161	5.0

Experimental Considerations

- Hardware Specifications
  - Dual 550MHz Pentium III Xeon processors
  - 1GB Memory
- Support { 1.00%, 0.80%, 0.60%, 0.40%, 0.20%, 0.10%, 0.08%, 0.06%, 0.04%, 0.02%, 0.01% }
- Confidence = 0%
- No other applications running (second processor handles system processes)





Study Results – Real Data

- At support >= 0.20%, Apriori performs as fast as or better than FP-Growth
- At support < 0.20%, Apriori completes whenever FP-Growth completes
  - Exception – BMS-WebView-2 @ 0.01%
- When 2 million rules are generated, Apriori finishes in 10 minutes or less
  - Proposed – Bottleneck is NOT the rule algorithm, but rule analysis

Real Data Results

		BMS-POS		BMS-WebView-1		BMS-WebView-2	
Algorithm	Support	Time	Rules	Time	Rules	Time	Rules
Apriori	0.01	186m	214,300,568	Failed	Failed	Failed	Failed
FP-Growth		120m		Failed		13m 12s	
Apriori	0.04	16m 9 s	5,061,105	Failed	Failed	58s	1,096,720
FP-Growth		10m 41s		Failed		29s	
Apriori	0.06	8m 35s	1,837,824	1m 50s	3,011,836	28s	510,233
FP-Growth		6m 7s		52s		16s	
Apriori	0.10	3m 58s	530,353	1.2s	10,360	9.1s	119,335
FP-Growth		3m 12s		1.2s		5.9s	
Apriori	0.20	1m 14s	103,449	0.4s	1,516	2.4s	12,665
FP-Growth		1m 35s		0.7s		2.3s	

Study Results – Artificial Data

- At support < 0.40%, FP-Growth performs MUCH faster than Apriori
- At support >= 0.40%, FP-Growth and Apriori are comparable

	Support	Time	Rules	Support	Time	Rules	Support	Time	Rules
Apriori	0.01	4m 4s	1,376,684	0.04	1m 1s	56,962	0.06	44s	41,215
FP-Growth		20s			9.2s			8.2s	
Apriori	0.10	34s	26,962	0.20	20s	13,151	0.40	5.7s	1.997
FP-Growth		7.1			5.8s			4.3s	

Real-World Study Conclusions

- FP-Growth (and other non-Apriori) perform better on artificial data
- On all data sets, Apriori performs sufficiently well in reasonable time periods for reasonable result sets
- FP-Growth may be suitable when low support, large result count, fast generation are needed
- Future research may best be directed toward analyzing association rules

Research Conclusions

- FP-Growth does not have a better complexity than Apriori
  - Common sense indicates that it will run faster
- FP-Growth does not always have a better running time than Apriori
  - Support, dataset appears more influential
- FP-Trees are very complex structures (Apriori is simple)
- Location of data (memory vs. disk) is non-factor in comparison of algorithms

To Use or Not To Use?

- Question: Should the FP-Growth be used in favor of Apriori?
  - Difficulty to code
  - High performance at extreme cases
  - Personal preference
- More relevant questions
  - What kind of data is it?
  - What kind of results do I want?
  - How will I analyze the resulting rules?

References

- Han, Jiawei, Pei, Jian, and Yin, Yiwen, "Mining Frequent Patterns without Candidate Generation". In Proc. of the ACM SIGMOD Int. Conf. on Management of Data, pages 1-12, Dallas, Texas, USA, 2000.
- Orlando, Salvatore, "High Performance Mining of Short and Long Patterns". 2001.
- Pei, Jian, Han, Jiawei, and Mao, Runying, "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets". In SIGMOD Int'l Workshop on Data Mining and Knowledge Discovery, May 2000.
- Webb, Geoffrey L., "Efficient search for association rules". In Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 99--107, 2000.
- Zaiane, Osmar, El-Hajj, Mohammad, and Lu, Paul, "Fast Parallel Association Rule Mining Without Candidacy Generation". In Proc. of the IEEE 2001 International Conference on Data Mining (ICDM'2001), San Jose, CA, USA, November 29-December 2, 2001
- Zaki, Mohammed J., "Generating Non-Redundant Association Rules". In Proceedings of the International Conference on Knowledge Discovery and Data Mining, 2000.
- Zheng, Zijian, Kohavi, Ron, and Mason, Llew, "Real World Performance of Association Rule Algorithms". In proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, California, August 2001.