Data Science

Association Rules

Themis Palpanas
University of Paris

Thanks for slides to:

- Jiawei Han
- Jeff Ullman

Roadmap

- Frequent Patterns
  - Frequent Pattern Analysis
  - Applications
  - Market-Basket Model
  - Association Rules
- A-Priori Algorithm
- Improvements to A-Priori

What Is Frequent Pattern Analysis?

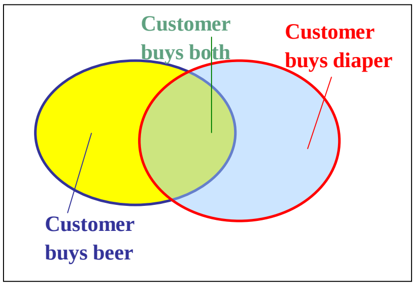
- Frequent pattern: a pattern (a set of items, subsequences, substructures, etc.) that occurs frequently in a data set
- First proposed by Agrawal, Imielinski, and Swami [AIS93] in the context of frequent itemsets and association rule mining
- Motivation: Finding inherent regularities in data
  - What products were often purchased together?— Beer and diapers?!
  - What are the subsequent purchases after buying a PC?
  - What kinds of DNA are sensitive to this new drug?
  - Can we automatically classify web documents?
- Applications
  - Basket data analysis, cross-marketing, catalog design, sale campaign analysis, Web log (click stream) analysis, and DNA sequence analysis.

Why Is Freq. Pattern Mining Important?

- Discloses an intrinsic and important property of data sets
- Forms the foundation for many essential data mining tasks
  - Association, correlation, and causality analysis
  - Sequential, structural (e.g., sub-graph) patterns
  - Pattern analysis in spatiotemporal, multimedia, time-series, and stream data
  - Classification: associative classification
  - Cluster analysis: frequent pattern-based clustering
  - Data warehousing: iceberg cube and cube-gradient
  - Semantic data compression: fascicles
  - Broad applications

Basic Concepts: Frequent Patterns and Association Rules

Transaction-id	Items bought
10	A, B, D
20	A, C, D
30	A, D, E
40	B, E, F
50	B, C, D, E, F



- Itemset X = $\{x_1, \ldots, x_k\}$
- Find all the rules $X \rightarrow Y$ with minimum support and confidence
  - support, s, probability that a transaction contains $X \cup Y$
  - confidence, c, conditional probability that a transaction having X also contains Y

Let sup_{min} = 50%, $conf_{min}$ = 50%
Freq. Pat.: {A:3, B:3, D:4, E:3, AD:3}
Association rules:
$A \rightarrow D$ (60%, 100%)
$D \rightarrow A$ (60%, 75%)

The Market-Basket Model

- A large set of *items*, e.g., things sold in a supermarket.
- A large set of *baskets*, each of which is a small set of the items, e.g., the things one customer buys on one day.

Support

- Simplest question: find sets of items that appear "frequently" in the baskets.
- *Support* for itemset I = the number of baskets containing all items in I.
- Given a support *threshold* s, sets of items that appear in $\geq s$ baskets are called *frequent itemsets*.

Example

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

  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}

- Frequent itemsets?

Example

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

  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}

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

Applications --- (1)

- Real market baskets: chain stores keep terabytes of information about what customers buy together.
  - Tells how typical customers navigate stores, lets them position tempting items.
  - Suggests tie-in "tricks," e.g., run sale on diapers and raise the price of beer.
- High support needed, or no $$'s .

Applications --- (2)

- "Baskets" = documents; "items" = words in those documents.
  - Lets us find words that appear together unusually frequently, i.e., linked concepts.
- "Baskets" = sentences, "items" = documents containing those sentences.
  - Items that appear together too often could represent plagiarism.

Applications --- (3)

- "Baskets" = Web pages; "items" = linked pages.
  - Pairs of pages with many common references may be about the same topic.
- "Baskets" = Web pages *p* ; "items" = pages that link to *p* .
  - Pages with many of the same links may be mirrors or about the same topic.

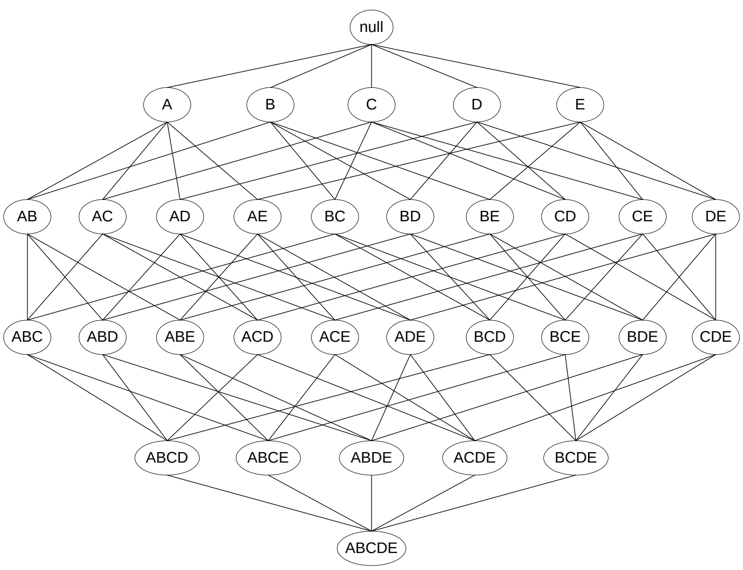
Important Point

- "Market Baskets" is an abstraction that models any many-many relationship between two concepts: "items" and "baskets."
  - Items need not be "contained" in baskets.
- The only difference is that we count co-occurrences of items related to a basket, not vice-versa.

Scale of Problem

- WalMart sells 100,000 items and can store billions of baskets.
- The Web has over 100,000,000 words and billions of pages.

A simple algorithm for finding all frequent itemsets ??



Association Rules

- If-then rules about the contents of baskets.
- $\{i_1, i_2,\ldots,i_k\} \rightarrow j$ means: "if a basket contains all of $i_1,\ldots,i_k$ then it is *likely* to contain j."
- *Confidence* of this association rule is the probability of j given $i_1,\ldots,i_k$.

Example

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}

- An association rule: $\{m, b\} \rightarrow c$.
  - Confidence = ?

Example

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

- An association rule: $\{m, b\} \rightarrow c$.
  - Confidence = 2/4 = 50%.

Interest

- The *interest* of an association rule $X \rightarrow Y$ is the absolute value of the amount by which the confidence differs from the probability of Y.

Example

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

- For association rule $\{m, b\} \rightarrow c$, item *c* appears in 5/8 of the baskets.
- Interest = | 2/4 - 5/8 | = 1/8 --- not very interesting.

Relationships Among Measures

- Rules with high support and confidence may be useful even if they are not "interesting."
  - We don't care if buying bread *causes* people to buy milk, or whether simply a lot of people buy both bread and milk.
- But high interest suggests a cause that might be worth investigating.

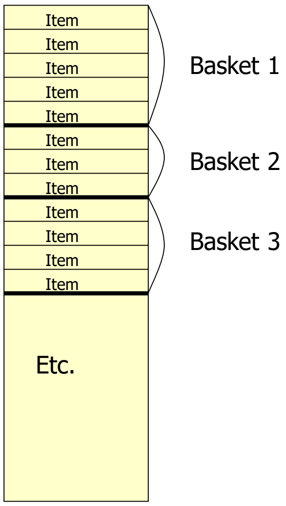
Finding Association Rules

- A typical question: "find all association rules with support $\geq s$ and confidence $\geq c$."
  - Note: "support" of an association rule is the support of the set of items it mentions.
- Hard part: finding the high-support (*frequent*) itemsets.
  - Checking the confidence of association rules involving those sets is relatively easy.

Computation Model

- Typically, data is kept in a "flat file" rather than a database system.
  - Stored on disk.
  - Stored basket-by-basket.
  - Expand baskets into pairs, triples, etc. as you read baskets.

File Organization



Computation Model --- (2)

- The true cost of mining disk-resident data is usually the number of disk I/O's.
- In practice, association-rule algorithms read the data in *passes* --- all baskets read in turn.
- Thus, we measure the cost by the number of passes an algorithm takes.

Main-Memory Bottleneck

- 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.
  - The number of different things we can count is limited by main memory.
  - Swapping counts in/out is a disaster.

Finding Frequent Pairs

- The hardest problem often turns out to be finding the frequent pairs.
- We'll concentrate on how to do that, then discuss extensions to finding frequent triples, etc.

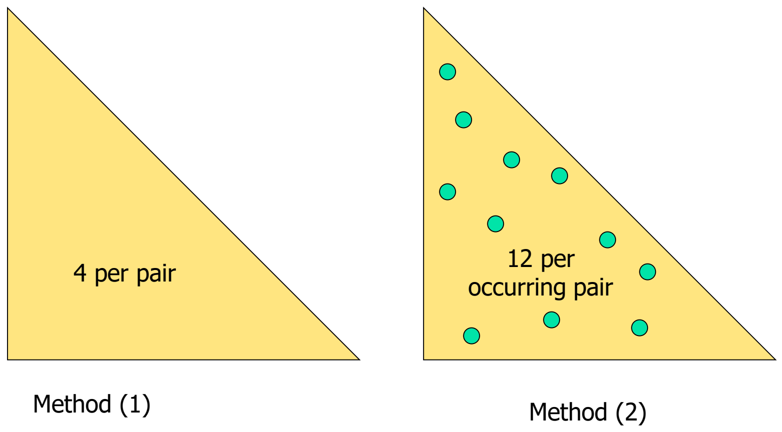
Naïve Algorithm

- Read file once, counting in main memory the occurrences of each pair.
  - Expand each basket of n items into its $n(n-1)/2$ pairs.
- Fails if $(\#\text{items})^2$ exceeds main memory.
  - Remember: #items can be 100K (Wal-Mart) or 10B (Web pages).

Details of Main-Memory Counting

- Two approaches:
  1. Count all item pairs, using a triangular matrix.
  2. Keep a table of triples [i, j, c] = the count of the pair of items {i,j} is c.
- (1) requires only (say) 4 bytes/pair.
- (2) requires 12 bytes, but only for those pairs with count > 0.

Details of Main-Memory Counting



Details of Approach #1

- Number items 1, 2,…
- Keep pairs in the order {1,2}, {1,3},…, {1,n}, {2,3}, {2,4},…,{2,n}, {3,4},…, {3,n},…{n-1,n}.
- Find pair {i, j} at the position:
  - $(i-1)(n-i/2) + j - i$
- Total number of pairs $n(n-1)/2$; total bytes about $2n^2$.

Details of Approach #2

- You need a hash table, with *i* and *j* as the key, to locate (*i, j, c*) triples efficiently.
  - Typically, the cost of the hash structure can be neglected.
- Total bytes used is about 12*p*, where *p* is the number of pairs that actually occur.
  - Beats triangular matrix if at most 1/3 of possible pairs actually occur.

Roadmap

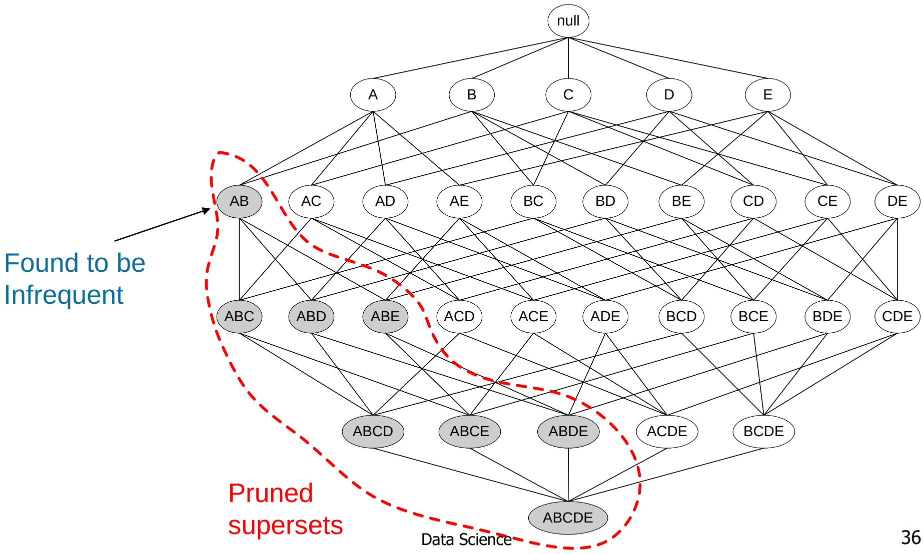
- Frequent Patterns
- A-Priori Algorithm
  - Monotonicity Property
  - Algorithm Description
- Improvements to A-Priori

A-Priori Algorithm --- (1)

- A two-pass approach called *a-priori* limits the need for main memory.
- Key idea: *monotonicity* : if a set of items appears at least *s* times, so does every subset.
  - Contrapositive for pairs: if item *i* does not appear in *s* baskets, then no pair including *i* can appear in *s* baskets.

(Agrawal & Srikant @VLDB'94, Mannila, et al. @ KDD' 94)

Illustrating the Apriori principle



A-Priori Algorithm --- (1)

A-Priori Algorithm --- (1)

A-Priori Algorithm --- (1)

A-Priori Algorithm --- (1)

A-Priori Algorithm --- (2)

- Pass 1: Read baskets and count in main memory the occurrences of each item.
  - memory requirements?

A-Priori Algorithm --- (2)

- Pass 1: Read baskets and count in main memory the occurrences of each item.
  - Requires only memory proportional to #items.

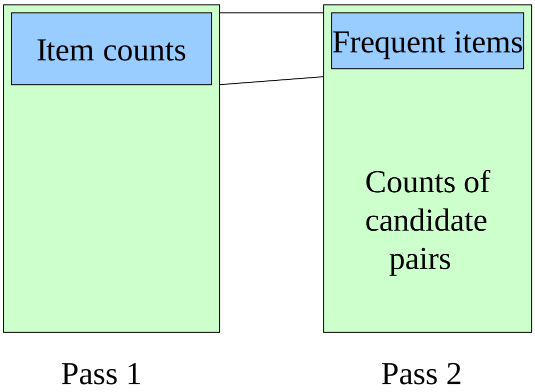
A-Priori Algorithm --- (2)

- Pass 1: Read baskets and count in main memory the occurrences of each item.
  - Requires only memory proportional to #items.
- Pass 2: Read baskets again and count in main memory only those pairs both of which were found in Pass 1 to be frequent.
  - memory requirements?

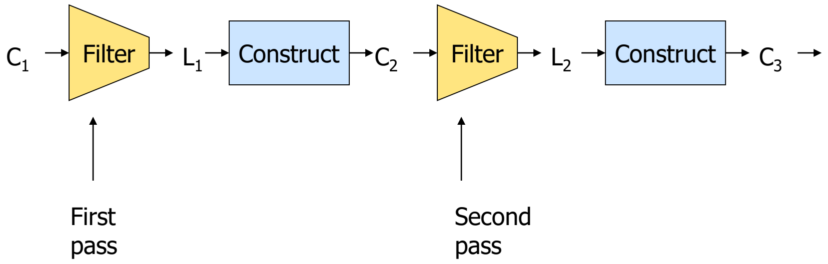
A-Priori Algorithm --- (2)

- Pass 1: Read baskets and count in main memory the occurrences of each item.
  - Requires only memory proportional to #items.
- Pass 2: Read baskets again and count in main memory only those pairs both of which were found in Pass 1 to be frequent.
  - Requires memory proportional to square of frequent items only.

Picture of A-Priori



Frequent Triples, Etc.



A-Priori for All Frequent Itemsets

- One pass for each *k*.
- Needs room in main memory to count each candidate *k* –tuple.
- For typical market-basket data and reasonable support (e.g., 1%), *k* = 2 requires the most memory.

The Apriori Algorithm—An Example

$Sup_{min} = 2$

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

The Apriori Algorithm—An Example

$Sup_{min} = 2$

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

C_1

1^{st} scan

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

The Apriori Algorithm—An Example

$Sup_{min} = 2$

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

C_1

1^{st} scan

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

The Apriori Algorithm—An Example

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

$Sup_{min} = 2$

1st scan → C_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

→ L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

C_2

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

The Apriori Algorithm—An Example

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

$Sup_{min} = 2$

1st scan → C_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

→ L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

C_2

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

2nd scan → C_2

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

The Apriori Algorithm—An Example

$Sup_{min} = 2$

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

1st scan → C_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

C_2

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

2nd scan → C_2

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

L_2

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

The Apriori Algorithm—An Example

$Sup_{min} = 2$

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

1st scan → C_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

C_2

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

2nd scan → C_2

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

L_2

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

C_3

Itemset
{B, C, E}

The Apriori Algorithm—An Example

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

Sup_{min} = 2

1[st] scan → C_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

→ L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

C_2

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

2[nd] scan → C_2

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

→ L_2

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

C_3

Itemset
{B, C, E}

3[rd] scan → L_3

Itemset	sup
{B, C, E}	2

The Apriori Algorithm

- 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 $\cup_k L_k$;

Important Details of Apriori

- How to generate candidates?
  - Step 1: self-joining L_k
  - Step 2: pruning
- How to count supports of candidates?
- Example of Candidate-generation
  - 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*}

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 C_k

  select ***p.item***$_1$, ***p.item***$_2$, ***…***, ***p.item***$_{k-1}$, ***q.item***$_{k-1}$

  from L_{k-1} ***p***, L_{k-1} ***q***

  where ***p.item***$_1$***=q.item***$_1$, ***…***, ***p.item***$_{k-2}$***=q.item***$_{k-2}$, ***p.item***$_{k-1}$ ***<*** ***q.item***$_{k-1}$
- Step 2: pruning

  forall ***itemsets c in*** 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

How to Count Supports of Candidates?

- Why counting supports of candidates 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

Exploiting the Apriori principle

1. Find frequent 1-items and put them to L_k (**k=1**)
2. Use L_k to generate a collection of *candidate* itemsets C_{k+1} with size (**k+1**)
3. Scan the database to find which itemsets in C_{k+1} are frequent and put them into L_{k+1}
4. If L_{k+1} is not empty
   - **k=k+1**
   - Goto step 2

R. Agrawal, R. Srikant: "Fast Algorithms for Mining Association Rules", *Proc. of the 20th Int'l Conference on Very Large Databases*, 1994.

The Apriori algorithm

C_k: Candidate itemsets of size k

L_k : frequent itemsets of size k

L_1 = {frequent 1-itemsets};

for (k = 2; L_k !=∅; 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 $\cup_k L_k$;

61

GenerateCandidates

- Assume the items in L_k are listed in an order (e.g., alphabetical)
- **Step 1:** ***self-joining*** L_k *(IN SQL)*

  insert into C_{k+1}

  select ***p.item$_1$, p.item$_2$, …, p.item$_k$, q.item$_k$***

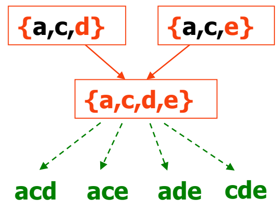
  from L_k ***p***, L_k ***q***

  where ***p.item$_1$=q.item$_1$, …, p.item$_{k-1}$=q.item$_{k-1}$, p.item$_k$ < q.item$_k$***

62

Example of Candidates Generation

- *L_3={abc, abd, acd, ace, bcd}*
- ***Self-joining***: *L_3*L_3*
  - ***abcd*** from ***abc*** and ***abd***
  - ***acde*** from ***acd*** and ***ace***



63

63

GenerateCandidates

- Assume the items in L_k are listed in an order (e.g., alphabetical)
- **Step 1:** ***self-joining*** L_k *(IN SQL)*

  insert into C_{k+1}

  select ***p.item$_1$, p.item$_2$, …, p.item$_k$, q.item$_k$***

  from L_k ***p***, L_k ***q***

  where ***p.item$_1$=q.item$_1$, …, p.item$_{k-1}$=q.item$_{k-1}$, p.item$_k$ < q.item$_k$***
- **Step 2:** ***pruning***

  forall ***itemsets c in*** C_{k+1} do

  forall ***k-subsets s of c*** do

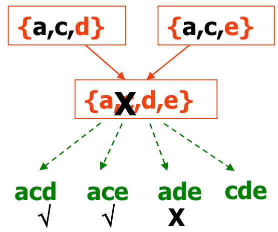
  if *(s is **not** in L_k)* **then delete c from** C_{k+1}

64

64

Example of Candidates Generation

- 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}



65

The Apriori algorithm

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

L_1 = {frequent items};
for (k = 1; L_k !=∅; 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 $\cup_k L_k$;

66

How to Count Supports of Candidates?

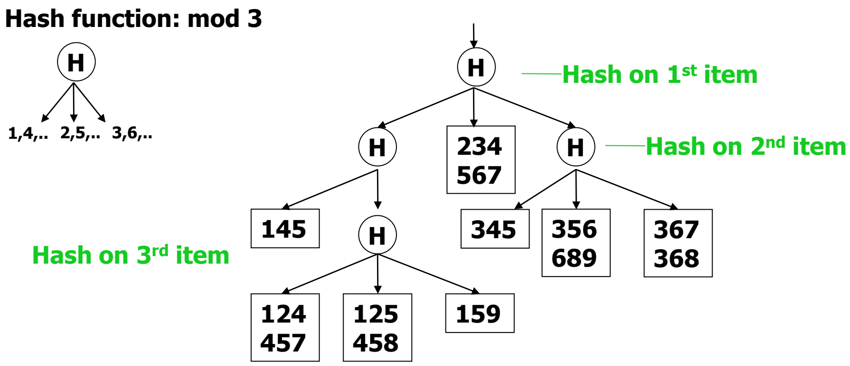
- Naive algorithm?

– 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

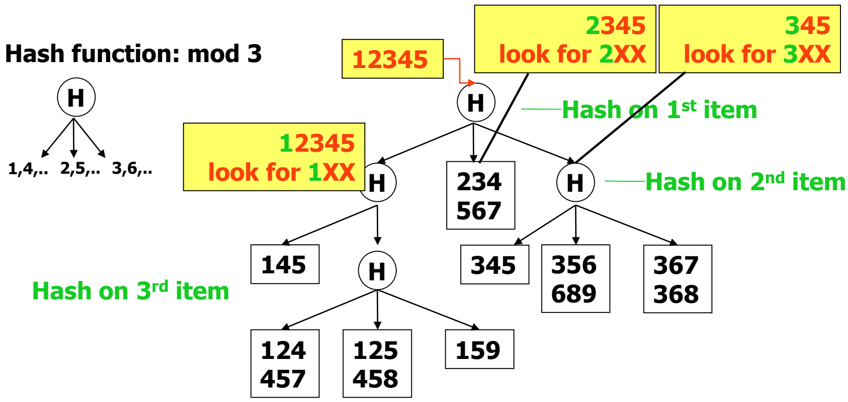
67

Example of the hash-tree for C_3



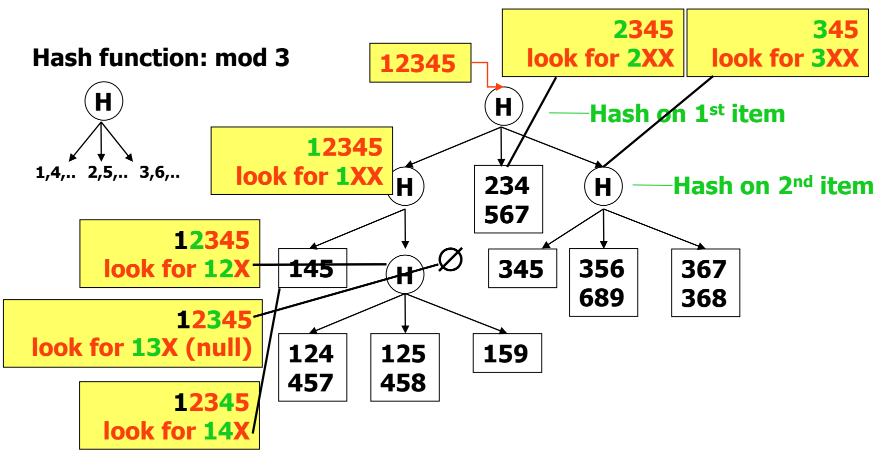
68

Example of the hash-tree for C_3



69

Example of the hash-tree for C_3



The subset function finds all the candidates contained in a transaction:
- At the root level it hashes on all items in the transaction
- At level i it hashes on all items in the transaction that come after item the i-th item

70

Where are the Association Rules?

- so far we have seen how A-priori efficiently computes all the frequent itemsets
- but how are the association rules generated from the frequent itemsets?

Association Rule Generation

- given the frequent itemsets, generate association rules as follows
  - for each frequent itemset I
    - generate all non-empty subsets of I
  - for each non-empty subset s of I
    - output association rule: s -> (I-s)

Association Rule Generation

- given the frequent itemsets, generate association rules as follows
  - for each frequent itemset I
    - generate all non-empty subsets of I
  - for each non-empty subset s of I
    - output association rule: s -> (I-s), if supp(I)/supp(s)>=c

Association Rule Generation

- given the frequent itemsets, generate association rules as follows
  - for each frequent itemset I
    - generate all non-empty subsets of I
  - for each non-empty subset s of I
    - output association rule: s -> (I-s), if supp(I)/supp(s)>=c

- we know supp(rule)>=s
  - generated from frequent itemsets

Roadmap

- Frequent Patterns
- A-Priori Algorithm
- Improvements to A-Priori
  - Park-Chen-Yu Algorithm
  - Multistage Algorithm
  - Approximate Algorithms
  - Compacting Results

PCY Algorithm

- Hash-based improvement to A-Priori.
- During Pass 1 of A-priori, most memory is idle.
- Use that memory to keep counts of buckets into which pairs of items are hashed.
  - Just the count, not the pairs themselves.
- Gives extra condition that candidate pairs must satisfy on Pass 2.

- J. Park, M. Chen, and P. Yu. An effective hash-based algorithm for mining association rules. In *SIGMOD'95*

PCY Algorithm --- Before Pass 1 Organize Main Memory

- Space to count each item.
  - One (typically) 4-byte integer per item.
- Use the rest of the space for as many integers, representing buckets, as we can.

PCY Algorithm --- Pass 1

```
FOR (each basket) {
    FOR (each item)
      add 1 to item's count;
    FOR (each pair of items) {
      hash the pair to a bucket;
      add 1 to the count for that
    bucket
    }
}
```

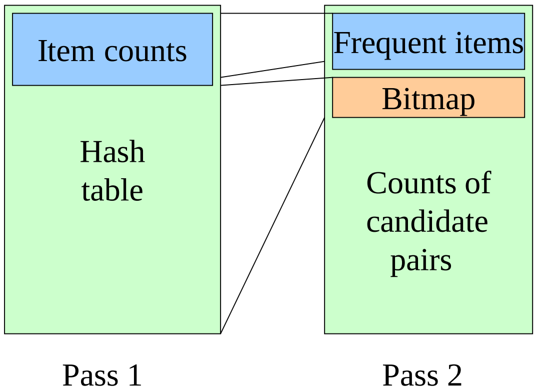
Observations About Buckets

1. If a bucket contains a frequent pair, then the bucket is surely frequent.
   - We cannot use the hash table to eliminate any member of this bucket.
2. Even without any frequent pair, a bucket can be frequent.
   - Again, nothing in the bucket can be eliminated.
3. But in the best case, the count for a bucket is less than the support s.
   - Now, all pairs that hash to this bucket can be eliminated as candidates, even if the pair consists of two frequent items.

PCY Algorithm --- Between Passes

- Replace the buckets by a bit-vector:
  - 1 means the bucket count exceeds the support *s* (frequent bucket); 0 means it did not.
- Integers are replaced by bits, so the bit-vector requires little second-pass space.
- Also, decide which items are frequent and list them for the second pass.

Picture of PCY



PCY Algorithm --- Pass 2

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

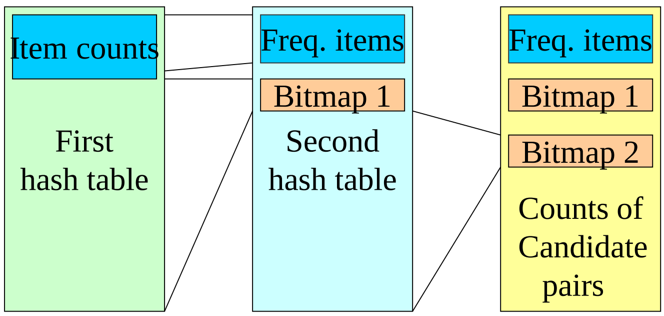
Memory Details

- Hash table requires buckets of 2-4 bytes.
  - Number of buckets thus almost 1/4-1/2 of the number of bytes of main memory.
- On second pass, a table of (item, item, count) triples is essential.
  - Thus, hash table must eliminate 2/3 of the candidate pairs to beat a-priori.

Multistage Algorithm

- Key idea: After Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY.
- On middle pass, fewer pairs contribute to buckets, so fewer *false positives* --- frequent buckets with no frequent pair.

Multistage Picture



Multistage --- Pass 3

- Count only those pairs $\{i,j\}$ that satisfy:
  1. Both i and j are frequent items.
  2. Using the first hash function, the pair hashes to a bucket whose bit in the first bit-vector is 1.
  3. Using the second hash function, the pair hashes to a bucket whose bit in the second bit-vector is 1.

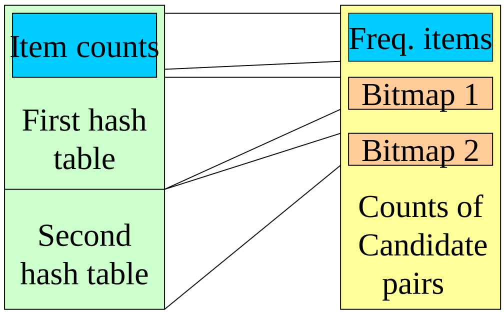
Important Points

1. The two hash functions have to be independent.
2. We need to check both hashes on the third pass.
   - If not, we would wind up counting pairs of frequent items that hashed first to an infrequent bucket but happened to hash second to a frequent bucket.

Multihash

- Key idea: use several independent hash tables on the first pass.
- Risk: halving the number of buckets doubles the average count. We have to be sure most buckets will still not reach count *s*.
- If so, we can get a benefit like multistage, but in only 2 passes.

Multihash Picture



Extensions

- Either multistage or multihash can use more than two hash functions.
- In multistage, there is a point of diminishing returns, since the bit-vectors eventually consume all of main memory.
- For multihash, the bit-vectors total exactly what one PCY bitmap does, but too many hash functions makes all counts $\geq s$.

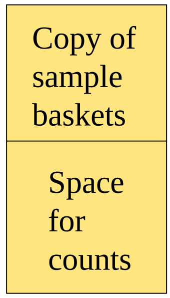
All (Or Most) Frequent Itemsets In $\leq$ 2 Passes

- Simple algorithm.
- SON (Savasere, Omiecinski, and Navathe).
- Toivonen.

Simple Algorithm --- (1)

- Take a main-memory-sized random sample of the market baskets.
- Run a-priori or one of its improvements (for sets of all sizes, not just pairs) in main memory, so you don't pay for disk I/O each time you increase the size of itemsets.
  - Be sure you leave enough space for counts.

The Picture



Simple Algorithm --- (2)

- Use as your support threshold a suitable, scaled-back number.
  - E.g., if your sample is 1/100 of the baskets, use s /100 as your support threshold instead of s .

Simple Algorithm --- Option

- Optionally, verify that your guesses are truly frequent in the entire data set by a second pass.
- But you don't catch sets frequent in the whole but not in the sample.
  - Smaller threshold, e.g., s /125, helps.

Toivonen's Algorithm --- (1)

- Start as in the simple algorithm, 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.
  - Goal is to avoid missing any itemset that is frequent in the full set of baskets.
- H. Toivonen. Sampling large databases for association rules. In *VLDB'96*

Toivonen's Algorithm --- (2)

- Add to the itemsets that are frequent in the sample the *negative border* of these itemsets.
- An itemset is in the negative border if it is not deemed frequent in the sample, but *all* its immediate subsets are.

Example: Negative Border

- *ABCD* is in the negative border if and only if it is not frequent, but all of *ABC*, *BCD*, *ACD*, and *ABD* are.

Toivonen's Algorithm --- (3)

- In a second pass, count all candidate frequent itemsets from the first pass, and also count the negative border.
- If no itemset from the negative border turns out to be frequent, then the candidates found to be frequent in the whole data are *exactly* the frequent itemsets.

Toivonen's Algorithm --- (4)

- What if we find something in the negative border is actually frequent?
- We must start over again!
- 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.

Theorem:

- If there is an itemset frequent in the whole, but not frequent in the sample, then there is a member of the negative border frequent in the whole.

Proof:

- Suppose not; i.e., there is an itemset *S* frequent in the whole, but not frequent or in the negative border in the sample.
- Let *T* be a smallest subset of *S* that is not frequent in the sample.
- *T* is frequent in the whole (monotonicity).
- *T* is in the negative border (else not "smallest").

SON Algorithm --- (1)

- Repeatedly read small subsets of the baskets into main memory and perform the first pass of the simple algorithm on each subset.
- An itemset becomes a candidate if it is found to be frequent in *any* one or more subsets of the baskets.
- A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association in large databases. In VLDB'95

SON Algorithm --- (2)

- 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 set of baskets unless it is frequent in at least one subset.

Compacting the Output

- A long pattern contains a combinatorial number of sub-patterns, e.g., $\{a_1, \ldots, a_{100}\}$ contains **?** sub-patterns

Compacting the Output

- A long pattern contains a combinatorial number of sub-patterns, e.g., $\{a_1, \ldots, a_{100}\}$ contains $\binom{100}{1} + \binom{100}{2} + \ldots + \binom{100}{100} = 2^{100} - 1 = 1.27*10^{30}$ sub-patterns!

Compacting the Output

- A long pattern contains a combinatorial number of sub-patterns, e.g., $\{a_1, \ldots, a_{100}\}$ contains $({}^{100}{}_{1}) + ({}^{100}{}_{2}) + \ldots + ({}^{100}{}_{100}) = 2^{100} - 1 = 1.27 * 10^{30}$ sub-patterns!

- Solution: *Mine closed patterns and max-patterns instead*

1. *Maximal Frequent itemsets* : no immediate superset is frequent.

2. *Closed itemsets* : no immediate superset has the same count.
   - Stores not only frequent information, but exact counts.

Closed Patterns and Max-Patterns

- An itemset X is closed if X is *frequent* and there exists *no super-pattern* Y ɔ X, *with the same support* as X (proposed by Pasquier, et al. @ ICDT'99)

- An itemset X is a max-pattern if X is frequent and there exists no frequent super-pattern Y ɔ X (proposed by Bayardo @ SIGMOD'98)

- Closed pattern is a lossless compression of freq. patterns
  - Reducing the # of patterns and rules

Example: Maximal/Closed

	Count	Maximal s=3	Closed
A	4	No	No
B	5	No	Yes
C	3	No	No
AB	4	Yes	Yes
AC	2	No	No
BC	3	Yes	Yes
ABC	2	No	Yes

Closed Patterns and Max-Patterns

- Exercise. DB = {<a_1, …, a_{100}>, < a_1, …, a_{50}>}
  - Min_sup = 1.
- What is the set of closed itemsets?

Closed Patterns and Max-Patterns

- Exercise. DB = $\{<a_1, \ldots, a_{100}>, < a_1, \ldots, a_{50}>\}$
  - Min_sup = 1.
- What is the set of closed itemsets?
  - $<a_1, \ldots, a_{100}>$: 1
  - $< a_1, \ldots, a_{50}>$: 2

Closed Patterns and Max-Patterns

- Exercise. DB = $\{<a_1, \ldots, a_{100}>, < a_1, \ldots, a_{50}>\}$
  - Min_sup = 1.
- What is the set of closed itemsets?
  - $<a_1, \ldots, a_{100}>$: 1
  - $< a_1, \ldots, a_{50}>$: 2
- What is the set of max-patterns?

Closed Patterns and Max-Patterns

- Exercise. DB = $\{<a_1, \ldots, a_{100}>, < a_1, \ldots, a_{50}>\}$
  - Min_sup = 1.
- What is the set of closed itemsets?
  - $<a_1, \ldots, a_{100}>$: 1
  - $< a_1, \ldots, a_{50}>$: 2
- What is the set of max-patterns?
  - $<a_1, \ldots, a_{100}>$: 1

Closed Patterns and Max-Patterns

- Exercise. DB = $\{<a_1, \ldots, a_{100}>, < a_1, \ldots, a_{50}>\}$
  - Min_sup = 1.
- What is the set of closed itemsets?
  - $<a_1, \ldots, a_{100}>$: 1
  - $< a_1, \ldots, a_{50}>$: 2
- What is the set of max-patterns?
  - $<a_1, \ldots, a_{100}>$: 1
- What is the set of all patterns?

Ref: Basic Concepts of Frequent Pattern Mining

- (Association Rules) R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. SIGMOD'93.
- (Max-pattern) R. J. Bayardo. Efficiently mining long patterns from databases. SIGMOD'98.
- (Closed-pattern) N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. ICDT'99.
- (Sequential pattern) R. Agrawal and R. Srikant. Mining sequential patterns. ICDE'95

Ref: Apriori and Its Improvements

- R. Agrawal and R. Srikant. Fast algorithms for mining association rules. VLDB'94.
- H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. KDD'94.
- A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. VLDB'95.
- J. S. Park, M. S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. SIGMOD'95.
- H. Toivonen. Sampling large databases for association rules. VLDB'96.
- S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket analysis. SIGMOD'97.
- S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. SIGMOD'98.