Association Rules

Juliana Freire
CS92223 – Fall 2013

Modified from Jeff Ullman, Jure Lescovek and Bing Liu
Association Rules: Some History

- Bar code technology allowed retailers to collect massive volumes of sales data
- Basket data: transaction date, set of items bought
- Leverage information for marketing
  - How to design coupons?
  - How to organize shelves?
Association Rules: Some History

- Data is very large and stored in tertiary storage
- Current (as of 1993) “database systems do not provide necessary functionality for a user interested in taking advantage of this information”
- Any feeling of deja vu?
Association Rules: Some History

• The birth of data mining!

• Agrawal et al. (SIGMOD 1993) introduced the problem
  • Mining a large collection of basket data to discover association rules

• Many papers followed…
Association Rules: Impact

[PDF] Fast algorithms for mining association rules
R Agrawal, R Srikant - Proc. 20th Int. Conf. Very Large Data ..., 1994 - www-cgi.cs.cmu.edu
This is a very long and complicated paper about taking a set of transactions (what the paper calls basket data) and finding association rules in them. For example, a marketing firm might want to ask "What percentage of people who bought X also bought Y?" Another question ...
Cited by 13400  Related articles  BL Direct  All 312 versions  Cite  More▼

Mining association rules between sets of items in large databases
R Agrawal, T Imieliński, A Swami - ACM SIGMOD Record, 1993 - dl.acm.org
Abstract We are given a large database of customer transactions. Each transaction consists of items purchased by a customer in a visit. We present an efficient algorithm that generates all significant association rules between items in the database. The algorithm incorporates ...
Cited by 11894  Related articles  BL Direct  All 99 versions  Cite

[PDF] Fast discovery of association rules
R Agrawal, H Mannila, R Srikant, H Toivonen - ... discovery and data ..., 1996 - cs.helsinki.fi
Abstract Association rules are statements of the form "98% of customers that purchase tires and automobile accessories also get automotive services." We consider the problem of discovering association rules between items in large databases. We present two new ...
Cited by 2458  Related articles  All 9 versions  Cite

Mining quantitative association rules in large relational tables
R Srikant, R Agrawal - ACM SIGMOD Record, 1996 - dl.acm.org
Abstract We introduce the problem of mining association rules in large relational tables.
Rakesh Agrawal
Technical Fellow, Microsoft Research
Data Mining - Web Search - Education - Privacy
Verified email at microsoft.com

Citation indices

<table>
<thead>
<tr>
<th></th>
<th>All</th>
<th>Since 2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citations</td>
<td>67728</td>
<td>33881</td>
</tr>
<tr>
<td>h-index</td>
<td>81</td>
<td>57</td>
</tr>
<tr>
<td>i10-index</td>
<td>186</td>
<td>144</td>
</tr>
</tbody>
</table>

Citations to my articles

Select: All, None

<table>
<thead>
<tr>
<th>Title / Author</th>
<th>Cited by</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast algorithms for mining association rules</td>
<td>13400</td>
<td>1994</td>
</tr>
<tr>
<td>R Agrawal, R Srikant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proc. 20th Int. Conf. Very Large Data Bases, VLDB 1215, 487-499</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mining association rules between sets of items in large databases</td>
<td>11894</td>
<td>1993</td>
</tr>
<tr>
<td>R Agrawal, T Imieliński, A Swami</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACM SIGMOD Record 22 (2), 207-216</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mining sequential patterns</td>
<td>4244</td>
<td>1995</td>
</tr>
<tr>
<td>R Agrawal, R Srikant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast discovery of association rules</td>
<td>2458</td>
<td>1996</td>
</tr>
<tr>
<td>R Agrawal, H Mannila, R Srikant, H Toivonen, Al Verkamo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advances in knowledge discovery and data mining 12, 307-328</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automatic subspace clustering of high dimensional data for data mining applications</td>
<td>2090</td>
<td>1998</td>
</tr>
<tr>
<td>R Agrawal, J Gehrke, D Gunopoulos, P Raghavan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACM SIGMOD Record 27 (2), 94-105</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Association Rule Discovery

Supermarket shelf management:

• **Goal:** Identify items that are bought together by sufficiently many customers

• **Approach:** Process the sales data collected with barcode scanners to find dependencies among items

• **A classic rule:**
  - If one buys diaper and milk, then he is likely to buy beer
  - Don’t be surprised if you find six-packs next to diapers!

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bread, Coke, Milk</td>
</tr>
<tr>
<td>2</td>
<td>Beer, Bread</td>
</tr>
<tr>
<td>3</td>
<td>Beer, Coke, Diaper, Milk</td>
</tr>
<tr>
<td>4</td>
<td>Beer, Bread, Diaper, Milk</td>
</tr>
<tr>
<td>5</td>
<td>Coke, Diaper, Milk</td>
</tr>
</tbody>
</table>

Rules Discovered:
- \{Milk\} --\> \{Coke\}
- \{Diaper, Milk\} --\> \{Beer\}
The Market-Basket Model

- A large set of *items*, e.g., things sold in a supermarket
  - \( I = \{i_1, i_2, \ldots, i_m\} \)

- A large set of *baskets/transactions*, e.g., the things one customer buys on one day
  - \( t \) a set of items, and \( t \subseteq I \).

- Transaction Database \( T \): a set of transactions \( T = \{t_1, t_2, \ldots, t_n\} \).

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bread, Coke, Milk</td>
</tr>
<tr>
<td>2</td>
<td>Beer, Bread</td>
</tr>
<tr>
<td>3</td>
<td>Beer, Coke, Diaper, Milk</td>
</tr>
<tr>
<td>4</td>
<td>Beer, Bread, Diaper, Milk</td>
</tr>
<tr>
<td>5</td>
<td>Coke, Diaper, Milk</td>
</tr>
</tbody>
</table>
Market-Baskets and Associations

- A many-many mapping (association) between two kinds of things.
  - Identify *connections among “items,”* not “baskets.”
  - E.g., 90% of transactions that purchase bread and butter also purchase milk

- The technology focuses on common events, not rare events ( “long tail” ).

**Rules Discovered:**
- \{Milk\} \rightarrow \{Coke\}
- \{Diaper, Milk\} \rightarrow \{Beer\}
Association Rules: Approach

• Given a set of baskets, discover association rules
  • People who bought \{a,b,c\} tend to buy \{d,e\}

• 2-step approach
  • Find frequent \textit{itemsets}
  • Generate \textit{association rules}

\begin{tabular}{|c|l|}
\hline
\textit{TID} & \textit{Items} \\
\hline
1 & Bread, Coke, Milk \\
2 & Beer, Bread \\
3 & Beer, Coke, Diaper, Milk \\
4 & Beer, Bread, Diaper, Milk \\
5 & Coke, Diaper, Milk \\
\hline
\end{tabular}

Rules Discovered:
\{\text{Milk}\} \rightarrow \{\text{Coke}\}
\{\text{Diaper, Milk}\} \rightarrow \{\text{Beer}\}
Applications – (1)

• **Items** = products; **baskets** = sets of products someone bought in one trip to the store.

• **Real market baskets:** Chain stores keep TBs of data 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

  • Only useful if many buy diapers & beer.

• **Amazon’s people who bought X also bought Y**
Applications – (2)

- **Baskets** = sentences; **items** = documents containing those sentences.
  - Items that appear together too often could represent plagiarism.
    - “I love NYC” – {d1, d3, d5}
    - “The subway is slow” – {d1, d3}
  - Notice items do not have to be “in” baskets.

- **Baskets** = patients; **Items** = drugs & side-effects
  - Has been used to detect combinations of drugs that result in particular side-effects
  - **But requires extension:** Absence of an item needs to be observed as well as presence
Applications – (3)

- **Baskets** = Web pages; **items** = words.

- Unusual words appearing together in a large number of documents, e.g., “Brad” and “Angelina,” may indicate an interesting relationship.
Aside: Words on the Web

- Many Web-mining applications involve words.
  1. Cluster pages by their topic, e.g., sports.
  2. Find useful blogs, versus nonsense.
  3. Determine the sentiment (positive or negative) of comments.
  4. Partition pages retrieved from an ambiguous query, e.g., “jaguar.”
Words – (2)

Some well-known facts from information retrieval:

1. Very common words are *stop words*.
   - They rarely help determine meaning, and they block from view interesting events, so ignore them.

2. The TF/IDF measure distinguishes “important” words from those that are usually not meaningful.
Words – (3)

\[ \text{TF/IDF} = \text{“term frequency, inverse document frequency”} \]

-relates the number of times a word appears to the number of documents in which it appears.

- Low values are words like “also” that appear at random.
- High values are words like “computer” that may be the topic of documents in which it appears at all.
Scale of the Problem

• WalMart sells 100,000 items and can store billions of baskets.

• The Web has billions of words and many billions of pages.
Frequent Itemsets

- Simplest question: find sets of items that appear “frequently” in the baskets.

- **Support** for itemset $I = \text{the number of baskets containing all items in } I$.
  - Often expressed as a fraction of the total number of baskets

- Given a support threshold $s$, sets of items that appear in at least $s$ baskets are called **frequent itemsets**.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bread, Coke, Milk</td>
</tr>
<tr>
<td>2</td>
<td>Beer, Bread</td>
</tr>
<tr>
<td>3</td>
<td>Beer, Coke, Diaper, Milk</td>
</tr>
<tr>
<td>4</td>
<td>Beer, Bread, Diaper, Milk</td>
</tr>
<tr>
<td>5</td>
<td>Coke, Diaper, Milk</td>
</tr>
</tbody>
</table>

Support of \{Beer, Bread\} = 2
Example: Frequent Itemsets

- 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: $\{m\}$, $\{c\}$, $\{b\}$, $\{j\}$,
  - $\{m,b\}$, $\{b,c\}$, $\{c,j\}$.
The Market-Basket Model

• A transaction \( t \) contains \( X \), a set of items (itemset) in \( I \), if \( X \subseteq t \).

• An association rule is an implication of the form:
  \[ X \rightarrow Y, \text{ where } X, Y \subseteq I, \text{ and } X \cap Y = \emptyset \]

• An itemset is a set of items.
  • E.g., \( X = \{\text{milk, bread, cereal}\} \) is an itemset.

• A \( k \)-itemset is an itemset with \( k \) items.
  • E.g., \( \{\text{milk, bread, cereal}\} \) is a 3-itemset
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 \( i_1, \ldots, i_k \).

\[
\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}
\]
Example: Confidence

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

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

\[
\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}
\]
Support and Confidence

• Support for rule $X \rightarrow Y$ in a data set $T$ is the number of transactions in $T$ that contain $X$. Assume $T$ has $n$ transactions.

• Then,

$$support = \frac{X.count}{n}$$

$$confidence = \frac{(X \cup Y).count}{X.count}$$
Interesting Association Rules

- Not all high-confidence rules are interesting
  - The rule $X \rightarrow \text{Milk}$ may have high confidence for many itemsets $X$, because milk is purchased very often (independent of $X$) and the confidence will be very high

- **Interest** of an association rule $I \rightarrow j$ is the difference between its confidence and the fraction of baskets that contain $j$
  
  $\text{Interest } (I \rightarrow j) = \text{conf}(I \rightarrow j) - \text{Pr}[j]$  

- Interesting rules are those with high positive or negative interest values

- For uninteresting rules, the fraction of baskets containing $j$ will be the same as the fraction of the subset baskets including $\{I,j\}$. So confidence will be high, but interest low
Example: Confidence and Interest

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

- An association rule: \( \{m, b\} \rightarrow c \).
  - Confidence = \( \frac{2}{4} = 50\% \).
  - Interest = \( |0.5 - \frac{5}{8}| = \frac{1}{8} \)
    - Item c appears in \( \frac{5}{8} \) of the baskets

\[
\text{Interest (I} \rightarrow j\text{)} = \text{conf(I} \rightarrow j\text{)} - \Pr[j]
\]
Finding Association Rules

• **Goal:** Find all rules that satisfy the user-specified *minimum support* (minsup) and *minimum confidence* (minconf).
  
  - Support $\geq s$ and confidence $\geq c$

• **Key Features**
  - Completeness: find all rules.
  - No target item(s) on the right-hand-side
  - Mining with data on hard disk (not in memory)

• **Hard part: Finding the frequent itemsets**
  - If $I \rightarrow j$ has high support and confidence, then both $I$ and $j$ will be frequent
The Apriori algorithm

- Probably the best known algorithm

- Two steps:
  - 1) Find all itemsets I that have minimum support (frequent itemsets, also called large itemsets).
  - 2) Rule generation: Use frequent itemsets to generate rules.
    - For every subset A of I, generate rule $A \rightarrow I \setminus A$
    - Perform a single pass to compute the rule confidence
      - $Conf(A,B \rightarrow C,D) = \frac{supp(A,B,C,D)}{supp(A,B)}$
      - If $A,B,C \rightarrow D$ is below confidence, so is $A,B \rightarrow C,D$
      - Can generate bigger rules from smaller ones
    - Output rules above confidence threshold
Example

\[ \text{conf}(I \rightarrow j) = \frac{\text{supp}(I,j)}{\text{supp}(I)} \]

\[ B_1 = \{m, c, b\} \quad B_2 = \{m, p, j\} \]
\[ B_3 = \{m, c, b, n\} \quad B_4 = \{c, j\} \]
\[ B_5 = \{m, p, b\} \quad B_6 = \{m, c, b, j\} \]
\[ B_7 = \{c, b, j\} \quad B_8 = \{b, c\} \]

- Min support \( s = 3 \), confidence \( c = 0.75 \)
Example

\[ \text{conf}(I \rightarrow j) = \frac{\text{supp}(I,j)}{\text{supp}(I)} \]

\[ B_1 = \{m, c, b\} \quad B_2 = \{m, p, j\} \]
\[ B_3 = \{m, c, b, n\} \quad B_4 = \{c, j\} \]
\[ B_5 = \{m, p, b\} \quad B_6 = \{m, c, b, j\} \]
\[ B_7 = \{c, b, j\} \quad B_8 = \{b, c\} \]

- **Min support** \( s=3 \), **confidence** \( c=0.75 \)
- **1) Frequent itemsets:**
  - \{b,m\} \{b,c\} \{c,m\} \{c,j\} \{m,c,b\}
Example

\[ \text{conf}(I \rightarrow j) = \frac{\text{supp}(I,j)}{\text{supp}(I)} \]

\[
\begin{align*}
B_1 &= \{m, c, b\} & B_2 &= \{m, p, j\} \\
B_3 &= \{m, c, b, n\} & B_4 &= \{c, j\} \\
B_5 &= \{m, p, b\} & B_6 &= \{m, c, b, j\} \\
B_7 &= \{c, b, j\} & B_8 &= \{b, c\}
\end{align*}
\]

- Min support \( s = 3 \), confidence \( c = 0.75 \)
- **1) Frequent itemsets:**
  - \{b,m\} \{b,c\} \{c,m\} \{c,j\} \{m,c,b\}
- **2) Generate rules:**
  - \( b \rightarrow m: \text{c} = \frac{4}{6} \)
  - \( b \rightarrow c: \text{c} = \frac{5}{6} \)
  - \( b, c \rightarrow m: \text{c} = \frac{3}{5} \)
  - \( m \rightarrow b: \text{c} = \frac{4}{5} \)
  - \( b \rightarrow c, m: \text{c} = \frac{3}{6} \)
  - \( b, m \rightarrow c: \text{c} = \frac{3}{4} \)
Compacting the Output

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

2. **Closed itemsets**: no immediate superset has the same count (> 0).
   - Stores not only frequent information, but exact counts
## Example

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th>Maximal (s=3)</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>AB</td>
<td>4</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AC</td>
<td>2</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>BC</td>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ABC</td>
<td>2</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Frequent, but superset BC also frequent.

Frequent, and its only superset, ABC, not freq.

Superset BC has same count.

Its only superset, ABC, has smaller count.
Computing Itemsets

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

Note: We want to find frequent itemsets. To find them, we have to count them. To count them, we have to generate them.
Computing Itemsets

- Cost of mining is the *number of disk I/Os*
- In practice, association-rule algorithms read data in passes
- We measure the cost by the *number of passes* over the data
- Main memory bottleneck:
  - As we read the baskets, we need to count the pairs, triples, …
  - The number of different things we can count is limited by main memory
  - Swapping counts in/out is a disaster. Why?
Finding Frequent Pairs

• This is the hardest problem!
  • Often, frequent pairs are common, frequent triples are rare
  • The probability of being frequent drops exponentially with size; number of sets grows more slowly with size.

• We always need to generate all the itemsets

• But we would only like to count/keep track of those itemsets that in the end turn out to be frequent
Naïve Algorithm

- Read file one, counting in main memory the occurrences of each pair
  - From each basket of n items, generate its n(n-1)/2 pairs using two nested loops
- Problem: fails if n^2 exceeds main memory
  - 100K (Walmart) 10B (Web pages)
Step 1: Mining all frequent itemsets

- A frequent *itemset* is an itemset whose support is $\geq \text{minsup}$. 
- **Key idea:** The apriori property (downward closure property): any subsets of a frequent itemset are also frequent itemsets.

**Diagram:**

```
  ABC  ABD  ACD  BCD
  /     /     /    /
 AB   AC   AD   BC  BD  CD
  /     /     /    /
 A   B  C   D
```
The Algorithm

- **Iterative algo.** (also called **level-wise search**): Find all 1-item frequent itemsets; then all 2-item frequent itemsets, and so on.
  - In each iteration $k$, only consider itemsets that contain some $k-1$ frequent itemset.

- Find frequent itemsets of size 1: $F_1$

- From $k = 2$
  - $C_k =$ candidates of size $k$: those itemsets of size $k$ that could be frequent, given $F_{k-1}$
  - $F_k =$ those itemsets that are actually frequent, $F_k \subseteq C_k$ (need to scan the database once).
Example – Finding Frequent Itemsets

Dataset T

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>T100</td>
<td>1, 3, 4</td>
</tr>
<tr>
<td>T200</td>
<td>2, 3, 5</td>
</tr>
<tr>
<td>T300</td>
<td>1, 2, 3, 5</td>
</tr>
<tr>
<td>T400</td>
<td>2, 5</td>
</tr>
</tbody>
</table>

itemset:count

1. scan T → $C_1$: {1}:2, {2}:3, {3}:3, {4}:1, {5}:3
   → $F_1$: {1}:2, {2}:3, {3}:3, {5}:3
   → $C_2$: {1,2}, {1,3}, {1,5}, {2,3}, {2,5}, {3,5}

2. scan T → $C_2$: {1,2}:1, {1,3}:2, {1,5}:1, {2,3}:2, {2,5}:3, {3,5}:2
   → $F_2$: {1,3}:2, {2,3}:2, {2,5}:3, {3,5}:2
   → $C_3$: {2, 3,5}

3. scan T → $C_3$: {2, 3, 5}:2 → $F_3$: {2, 3, 5}

minsup=0.5
Details: Ordering of Items

- The items in \( I \) are sorted in **lexicographic order** (which is a total order).

- The order is used throughout the algorithm in each itemset.

- \( \{w[1], w[2], \ldots, w[k]\} \) represents a \( k \)-itemset \( w \) consisting of items \( w[1], w[2], \ldots, w[k] \), where \( w[1] < w[2] < \ldots < w[k] \) according to the total order.
Algorithm Apriori($T$)

$C_1 \leftarrow \text{init-pass}(T)$;

$F_1 \leftarrow \{ f \mid f \in C_1, f.\text{count}/n \geq \text{minsup} \}$;  // n: no. of transactions in $T$

for ($k = 2; F_{k-1} \neq \emptyset; k++$) do

    $C_k \leftarrow \text{candidate-gen}(F_{k-1})$;

    for each transaction $t \in T$ do

        for each candidate $c \in C_k$ do

            if $c$ is contained in $t$ then

                $c.\text{count}++$;

            end

        end

    end

    $F_k \leftarrow \{ c \in C_k \mid c.\text{count}/n \geq \text{minsup} \}$

end

return $F \leftarrow \bigcup_k F_k$;
The candidate-gen function takes $F_{k-1}$ and returns a superset (called the candidates) of the set of all frequent $k$-itemsets. It has two steps:

- **join step**: Generate all possible candidate itemsets $C_k$ of length $k$.
- **prune step**: Remove those candidates in $C_k$ that cannot be frequent.
Candidate-gen function

**Function** candidate-gen($F_{k-1}$)

\[ C_k \leftarrow \emptyset; \]

**for all** $f_1, f_2 \in F_{k-1}$

\[ \text{with } f_1 = \{i_1, \ldots, i_{k-2}, i_{k-1}\} \]

\[ \text{and } f_2 = \{i_1, \ldots, i_{k-2}, i'_{k-1}\} \]

\[ \text{and } i_{k-1} < i'_{k-1} \text{ do} \]

\[ c \leftarrow \{i_1, \ldots, i_{k-1}, i'_{k-1}\}; \]

\[ C_k \leftarrow C_k \cup \{c\}; \]

**for each** $(k-1)$-subset $s$ of $c$ do

\[ \text{if } (s \notin F_{k-1}) \text{ then} \]

\[ \text{delete } c \text{ from } C_k; \]

\[ \text{// prune} \]

end

end

return $C_k$;
An Example

- \( F_3 = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{1, 3, 5\}, \{2, 3, 4\}\} \)

- After join
  - \( C_4 = \{\{1, 2, 3, 4\}, \{1, 3, 4, 5\}\} \)

- After pruning:
  - \( C_4 = \{\{1, 2, 3, 4\}\} \)
    - because \(\{1, 4, 5\}\) is not in \( F_3 \) (\(\{1, 3, 4, 5\}\) is removed)
Step 2: Generating rules from frequent itemsets

- Frequent itemsets ≠ association rules
- One more step is needed to generate association rules
- For each frequent itemset $X$,

  For each proper nonempty subset $A$ of $X$,
  - Let $B = X - A$
  - $A \rightarrow B$ is an association rule if
    - $\text{Confidence}(A \rightarrow B) \geq \text{minconf}$,
    - $\text{support}(A \rightarrow B) = \text{support}(A \cup B) = \text{support}(X)$
    - $\text{confidence}(A \rightarrow B) = \frac{\text{support}(A \cup B)}{\text{support}(A)}$

CS583, Bing Liu, UIC
Generating Rules: An Example

• Suppose \{2,3,4\} is frequent, with sup=50%
  • Proper nonempty subsets: \{2,3\}, \{2,4\}, \{3,4\}, \{2\}, \{3\}, \{4\}, with sup=50%, 50%, 75%, 75%, 75%, 75% respectively
  • These generate these association rules:
    • 2,3 \rightarrow 4, confidence=100%
    • 2,4 \rightarrow 3, confidence=100%
    • 3,4 \rightarrow 2, confidence=67%
    • 2 \rightarrow 3,4, confidence=67%
    • 3 \rightarrow 2,4, confidence=67%
    • 4 \rightarrow 2,3, confidence=67%
  • All rules have support = 50%
Generating Rules: Summary

• To recap, in order to obtain $A \rightarrow B$, we need to have $\text{support}(A \cup B)$ and $\text{support}(A)$

• All the required information for confidence computation has already been recorded in itemset generation. No need to see the data $T$ any more.

• *This step is not as time-consuming as frequent itemsets generation.*
Example: Counting Pairs

- Suppose $10^5$ items.
- Suppose counts are 4-byte integers.
- Number of pairs of items: $10^5(10^5-1)/2 = 5 \times 10^9$ (approximately).
- Therefore, $2 \times 10^{10}$ (20 gigabytes) of main memory needed.
Details of Main-Memory Counting

- **Two approaches:**
  1. Count all 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 4 bytes/pair.
  - **Note:** always assume integers are 4 bytes.

- (2) requires 12 bytes, but only for those pairs with count > 0.
Comparing Approaches

Method (1) 4 per pair

Method (2) 12 per occurring pair
Triangular-Matrix Approach

- \( n \) = total number of items
- Requires table of size \( O(n) \) to convert item names to consecutive integers.
- Count \( \{i, j\} \) only if \( i < j \).
- Keep pairs in the order \( \{1,2\}, \{1,3\},..., \{1,n\}, \{2,3\}, \{2,4\},...,\{2,n\}, \{3,4\},..., \{3,n\},...,\{n-1,n\} \).
- Pair \( \{i,j\} \) is at position \((i-1)(n-i/2) + j – 1\)
- Total number of pairs \( n(n-1)/2 \); total bytes \( 2n^2 \)
Details of Approach #2

- Total bytes used is about $12p$, where $p$ is the number of pairs that actually occur.
  - Beats triangular matrix if at most $1/3$ of possible pairs actually occur.

- May require extra space for retrieval structure, e.g., a hash table.
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.
A-Priori Algorithm – (2)

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

• Items that appear at least \( s \) times are the *frequent 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 (for counts)
  • Plus a list of the frequent items (so you know what must be counted).
Main-Memory Picture of A-Priori

Pass 1

Item counts

Pass 2

Frequent items

Counts of pairs of frequent items
Detail for A-Priori

- You can use the triangular matrix method with \( n = \) number of frequent items.
  - May save space compared with storing triples.
- **Trick**: re-number frequent items 1,2,… and keep a table relating new numbers to original item numbers.
For each $k$, we construct two sets of $k$–tuples (sets of size $k$):

- $C_k =$ candidate $k$–sets = those that might be frequent sets (support $\geq s$) based on information from the pass for $k–1$.
- $L_k =$ the set of truly frequent $k$–sets.
Frequent Triples, Etc.

All items \( \rightarrow C_1 \rightarrow \text{Filter} \rightarrow L_1 \rightarrow \text{Construct} \rightarrow C_2 \rightarrow \text{Filter} \rightarrow L_2 \rightarrow \text{Construct} \rightarrow C_3 \rightarrow \) All pairs of items from \( L_1 \rightarrow \) Count the pairs

First pass

Frequent items

To be explained

Second pass

Frequent pairs

Count the items

Count the pairs

Filter
A-Priori for All Frequent Itemsets

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

- \( C_1 = \{ \{b\} \{c\} \{j\} \{m\} \{n\} \{p\} \} \)
- Count the support of itemsets in \( C_1 \)
- Prune non-frequent: \( L_1 = \{ b, c, j, m \} \)
- Generate \( C_2 = \{ \{b,c\} \{b,j\} \{b,m\} \{c,j\} \{c,m\} \{j,m\} \} \)
- Count the support of itemsets in \( C_2 \)
- Prune non-frequent: \( L_2 = \{ \{b,m\} \{b,c\} \{c,m\} \{c,j\} \} \)
- Generate \( C_3 = \{ \{b,c,m\} \{b,c,j\} \{b,m,j\} \{c,m,j\} \} \)
- Count the support of itemsets in \( C_3 \)
- Prune non-frequent: \( L_3 = \{ \{b,c,m\} \} \)
A-Priori for All Frequent Itemsets

- One pass for each $k$ (itemset size)
- 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

**Many possible extensions:**
- Lower the support $s$ as itemset gets bigger
- Association rules with intervals:
  - For example: Men over 65 have 2 cars
- Association rules when items are in a taxonomy
  - Bread, Butter $\rightarrow$ FruitJam
  - BakedGoods, MilkProduct $\rightarrow$ PreservedGoods
Frequent Itemsets – (2)

- $C_1 =$ all items
- In general, $L_k =$ members of $C_k$ with support $\geq s$.
- $C_{k+1} =$ $(k + 1)$ -sets, each $k$ of which is in $L_k$. 
Improvements to A-Priori
Association Rules: Not enough memory

- Counting for candidates C2 requires a lot of memory -- $O(n^2)$

- Can we do better?

- \textit{PCY}: In pass 1, there is a lot of memory left, leverage that to help with pass 2
  - Maintain a has table with as many buckets as fit in memory

- \textit{Multistage} improves PCY
Aside: Hash-Based Filtering

- **Simple problem**: I have a set $S$ of one billion strings of length 10.
- I want to scan a larger file $F$ of strings and output those that are in $S$.
- I have 1GB of main memory.
  - So I can't afford to store $S$ in memory.
Solution – (1)

- Create a bit array of 8 billion bits initially all 0’s.
- Choose a hash function $h$ with range $[0, 8\times10^9)$, and hash each member of $S$ to one of the bits, which is then set to 1.
- Filter the file $F$ by hashing each string and outputting only those that hash to a 1.
Solution – (2)

\[ h(s_1), h(s_2), \ldots, h(s_n) = 0010001011000 \]

File $F$ to output; may be in $S$.

$\text{Drop; surely not in } S.$
Solution – (3)

- As at most $1/8$ of the bit array is 1, only $1/8^{th}$ of the strings not in $S$ get through to the output.
- If a string is in $S$, it surely hashes to a 1, so it always gets through.
- Can repeat with another hash function and bit array to reduce the *false positives* by another factor of 8.
Solution – Summary

• Each filter step costs one pass through the remaining file $F$ and reduces the fraction of false positives by a factor of 8.

• Repeat passes until few false positives.

• Either accept some errors, or check the remaining strings.
  • e.g., divide surviving $F$ into chunks that fit in memory and make a pass though $S$ for each.
Aside: Throwing Darts

- A number of times we are going to need to deal with the problem: If we throw $k$ darts into $n$ equally likely targets, what is the probability that a target gets at least one dart?
- **Example**: targets = bits, darts = hash values of elements.
Throwing Darts – (2)

\[ 1 - (1 - \frac{1}{n})^{n(k/n)} \]

Equals 1/e as \( n \to \infty \)

Equivalent

\[ 1 - e^{-k/n} \]

Probability at least one dart hits target

Probability target not hit by one dart

\( k \) darts into \( n \) equally likely targets
Throwing Darts – (3)

• If $k << n$, then $e^{-k/n}$ can be approximated by the first two terms of its Taylor expansion: $1 – k/n$.

• Example: $10^9$ darts, $8*10^9$ targets.
  • True value: $1 – e^{-1/8} = .1175$.
  • Approximation: $1 – (1 – 1/8) = .125$. 
Improvement: Superimposed Codes (Bloom Filters)

- We could use two hash functions, and hash each member of $S$ to two bits of the bit array.
- Now, around $\frac{1}{4}$ of the array is 1’s.
- But we transmit a string in $F$ to the output only if both its bits are 1, i.e., only $\frac{1}{16}$th are false positives.
  - Actually $(1-e^{-1/4})^2 = 0.0493$. 
Superimposed Codes – (2)

- Generalizes to any number of hash functions.
- The more hash functions, the smaller the probability of a false positive.
- **Limiting Factor**: Eventually, the bit vector becomes almost all 1’s.
  - Almost anything hashes to only 1’s.
Aside: History

• The idea is attributed to Bloom (1970).

• “superimposed codes,” were invented at Bell Labs
  • Technically, the original paper on superimposed codes (Kautz and Singleton, 1964) required *uniqueness*: no two small sets have the same bitmap.
PCY Algorithm – An Application of Hash-Filtering

- 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.
Needed Extensions to Hash-Filtering

1. Pairs of items need to be generated from the input file; they are not present in the file.

2. We are not just interested in the presence of a pair, but we need to see whether it is present at least \( s \) (support) times.
PCY Algorithm – (2)

- A bucket contains a *frequent pair* if its count is at least the support threshold.
- If a bucket is not frequent, no pair that hashes to that bucket could possibly be a frequent pair.
- On Pass 2, we only count pairs that hash to frequent buckets.
Picture of PCY
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 in the basket)
        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. A bucket that a frequent pair hashes to 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 is frequent; 0 means it is not.

• 4-byte integers are replaced by bits, so the bit-vector requires 1/32 of memory.

• Also, decide which items are frequent and list them for the second pass.
PCY Algorithm – Pass 2

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

- Both conditions are necessary for the pair to have a chance of being frequent.
Main-Memory: PCY

- **Pass 1**
  - Item counts
  - Hash table for pairs

- **Pass 2**
  - Frequent items
  - Bitmap
  - Counts of candidate pairs
Memory Details

- Buckets require a few bytes each.
  - **Note:** we don’t have to count past s.
  - # buckets is $O$(main-memory size).

- On second pass, a table of \((\text{item, item, count})\) triples is essential
  - Thus, hash table must eliminate 2/3 of the candidate pairs for PCY to beat a-priori.
Example

• 1GB of RAM for hash table in pass 1
• 1 billion baskets, each with 10 items
• How many buckets can we fit in memory? _____
• How many item pairs? ______
• What is the average count for the buckets? _____
• What is a good value for support?
Refinement: Multistage Algorithm

- Limit the number of candidates to be counted
  - Remember: memory is the bottleneck
  - Still need to generate all itemsets
  - Uses several successive hash tables—requires more than two passes

- Key idea: After Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY.
  - i and j are frequent, and
  - \{i,j\} hashes to a frequent bucket from Pass 1

- On middle pass, fewer pairs contribute to buckets, so fewer false positives—frequent buckets with no frequent pair.
Multistage Picture

First hash table

Item counts

Pass 1

Count items
Hash pairs \{i,j\}

Second hash table

Freq. items

Pass 2

Hash pairs \{i,j\} into Hash2 iff:
\{i,j\} hashes to freq. bucket in B1

Counts of candidate pairs

Pass 3

Count pairs \{i,j\} iff:
i,j are frequent,
\{i,j\} hashes to freq. bucket in B1
\{i,j\} hashes to freq. bucket in B2

Main memory

Bitmap 1

Bitmap 2
• Count only those pairs \( \{i, j\} \) that satisfy these candidate pair conditions:

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

Main memory

Pass 1

First hash table

Second hash table

Pass 2

Item counts

Freq. items

Counts of candidate pairs

Bitmap 1

Bitmap 2
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 occupy exactly what one PCY bitmap does, but too many hash functions makes all counts $\geq s$. 
All (Or Most) Frequent Itemsets
In $< 2$ Passes

- A-Priori, PCY, etc., take $k$ passes to find frequent itemsets of size $k$.

- Other techniques use 2 or fewer passes for all sizes:
  - Simple algorithm.
  - SON (Savasere, Omiecinski, and Navathe).
  - Toivonen (see textbook)
Random Sampling – (1)

• Take a 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.

• 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 $s$. 
Main-Memory Picture

<table>
<thead>
<tr>
<th>Copy of sample baskets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space for counts</td>
</tr>
</tbody>
</table>
Random Sampling:— 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 catch more truly frequent itemsets.
    - But requires more space.
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.
  • This is not sampling but processing the entire file in memory-sized chunks

• An itemset becomes a candidate if it is found to be frequent in any one or more subsets of the baskets.
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.
SON Algorithm – Distributed Version

- This idea lends itself to distributed data mining.
- If baskets are distributed among many nodes, compute frequent itemsets at each node, then distribute the candidates from each node.
- Finally, accumulate the counts of all candidates.
SON Mapreduce

- **Map 1**: support = s * p
  - Each mapper gets a fraction p of input
  - Output: (FrequentItemset, 1)

- **Reduce 1**: reducer is assigned a set of itemsets
  - Output: itemsets that appear 1 or more times

- **Map2**: candidates and portion of input file
  - Count occurrences of candidates in file
  - Output: (CandidateSet, support)

- **Reduce 2**: sum support values for CandidateSet
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.
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:
  1. It is not frequent in the sample, but
  2. All of \(ABC, BCD, ACD,\) and \(ABD\) are.

• \(A\) is in the negative border if and only if it is not frequent in the sample.
  ◆ Because the empty set is always frequent.
  ◆ Unless there are fewer baskets than the support threshold (silly case).
Picture of Negative Border

... tripletons
doubletons singletons

Negative Border

Frequent Itemsets from Sample
Toivonen’s Algorithm – (3)

• In a second pass, count all candidate frequent itemsets from the first pass, and also count their 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 that 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.
If Something in the Negative Border is Frequent . . .

... tripletons

doubletons

单身tons

We broke through the negative border. How far does the problem go?

Frequent Itemsets from Sample

Negative Border
Theorem:

• If there is an itemset that is frequent in the whole, but not frequent in the sample, then there is a member of the negative border for the sample that is frequent in the whole.
• **Proof:** Suppose not; i.e.;

1. There is an itemset \( S \) frequent in the whole but not frequent in the sample, and

2. Nothing in the negative border is frequent in the whole.

• Let \( T \) be a **smallest** subset of \( S \) that is not frequent in the sample.

• \( T \) is frequent in the whole (\( S \) is frequent + monotonicity).

• \( T \) is in the negative border (else not “smallest”).
Compacting the Output

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

2. **Closed itemsets**: no immediate superset has the same count (> 0).
   - Stores not only frequent information, but exact counts.
**Example: Maximal/Closed**

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th>Maximal (s=3)</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>AB</td>
<td>4</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AC</td>
<td>2</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>BC</td>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ABC</td>
<td>2</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Frequent, but superset BC also frequent.
Frequent, and its only superset, ABC, not freq.
Superset BC has same count.
Its only superset, ABC, has smaller count.