3.2.2 Mining Association Rules

華中科技大學 计算机科学与技术学院 School of Computer Science &Technology, HUST

- **Step 1: Find all frequent itemsets I**
  - According to *support threshold s* (We will explain this later)
- **Step 2: Rule generation**
  - For every subset A of I, generate a rule $A \to I \setminus A$
    - Since I is frequent, A is also frequent
    - Single pass to compute rule confidence. confidence(***A,B*→*C,D***) = support(**A,B,C,D**) / support(**A,B**)
    - **Observation for further pruning (剪枝):** If **A,B,C→D** is below confidence, so is **A,B→C,D**
  - **Output the rules above the confidence threshold**
- **Step 3: Find interesting association rules (optional)**
  - According to *interest*

$$conf(I \to j) = \frac{support(I \cup j)}{support(I)}$$

$$Interest(I \to j) = conf(I \to j) - \Pr[j]$$

3.2.2 Example

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

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}

New example. Note: m for milk, c for coke, p for pepsi, b for beer, j fo r juice

- **Support threshold** $s = 3$, **confidence** $c = 0.75$
- **1) Frequent itemsets:**
  - {b,m} {b,c} {c,m} {c,j} {m,c,b}
- **2) Generate rules:**
  - ~~b→m: c=4/6~~ b→c: c=5/6 ~~b,c→m: c=3/5~~
  - m→b: c=4/5 c→b: c=5/6 b,m→c: c=3/4
  - ~~c→m: c=3/6~~ ~~m→c: c=3/5~~ ~~b→c,m: c=3/6~~
  - ~~c→j: c=3/6~~ j→c: c=3/4 ……

$$conf(I \to j) = \frac{support(I \cup j)}{support(I)}$$

3.2.3 Compacting the Output

- **To reduce the number of rules, we can further post-process them and only output in step 1 (Find all frequent itemsets *I*):**
  - **Maximal frequent itemsets(极大频繁项集):** frequent, and no immediate superset(超集) is frequent
    - Gives more pruning

  or

  - **Closed frequent itemsets(闭合频繁项集):** frequent, and no immediate superset has the same count (> 0),
    - Stores not only frequent information, but exact counts

Note: **Superset(超集)** 若一个集合S2中的每一个元素都在集合S1中，且集合S1中可能包含S2中没有的元素，则集合S1就是S2的一个超集。S1是S2的超集，则S2是S1的真子集，反之亦然。

3.2.3 Example: Maximal/Closed

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

Support threshold $s=3$

	Support	frequent	Maximal	Closed
A	4	Yes	No	No
B	5	Yes	No	Yes
C	3	Yes	No	No
AB	4	Yes	Yes	Yes
AC	2	No	No	No
BC	3	Yes	Yes	Yes
ABC	2	No	No	No
		Total:5	Total:2	Total:3

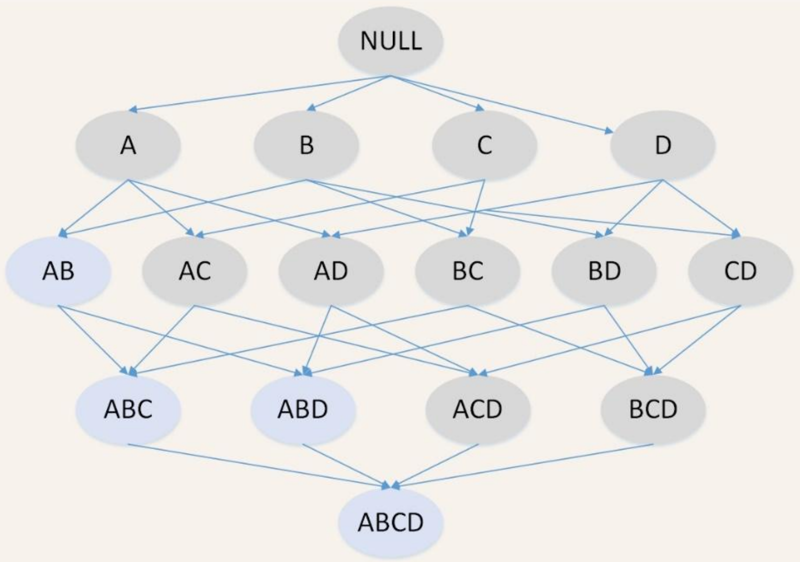
Frequent, but superset AB also frequent.

Frequent, and its only superset ABC, not freq.

Freq, but superset AB has same count.

Its only superset, ABC, has smaller count.

Not freq.

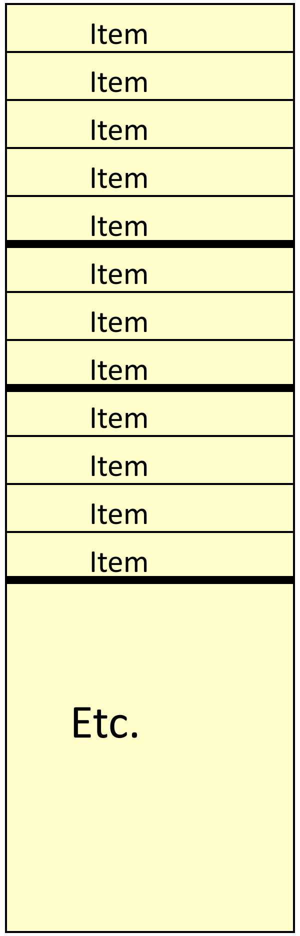


Section 3.3: Finding Frequent Itemsets

Content

3.3.1 Generating 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***



Items are positive integers, and boundaries between baskets are –1.

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

3.3.2 Itemsets Computation Model

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

3.3.2 Itemsets Computation Model

- The true cost of mining disk-resident data is usually the **number of disk I/Os**

- In practice, association-rule algorithms read the data in ***passes*** – all baskets read in turn

- We measure the cost by the **number of passes (扫描次数)** an algorithm makes over the data

3.3.3 Finding Frequent Pairs

- **The hardest problem often turns out to be finding the frequent pairs (频繁项对) of items $\{i_1, i_2\}$**
  - **Why?** Freq. pairs are common, freq. triples are rare
    - **Why?** Probability of being frequent drops exponentially with size; number of sets grows more slowly with size
- **Let’s first concentrate on pairs, then extend to larger sets**
- **The approach:**
  - 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

3.3.3 Finding Frequent Pairs

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

- **Naïve approach to finding frequent pairs**
- Read file once, counting in main memory the occurrences of each pair:
  - From each basket of ***n*** items, generate its ***n(n-1)/2*** pairs by two nested loops
- **However, fails if (#items)^2 exceeds main memory**
  - **Remember:** #items can be 100K (Wal-Mart) or 10B (Web pages)
    - Suppose 10^5 items, counts are 4-byte integers
    - Number of pairs of items: $10^5(10^5\text{-}1)/2 = 5*10^9$
    - Therefore, $2*10^{10}$ (20 gigabytes) of memory needed

3.3.4 Counting Candidate Pairs

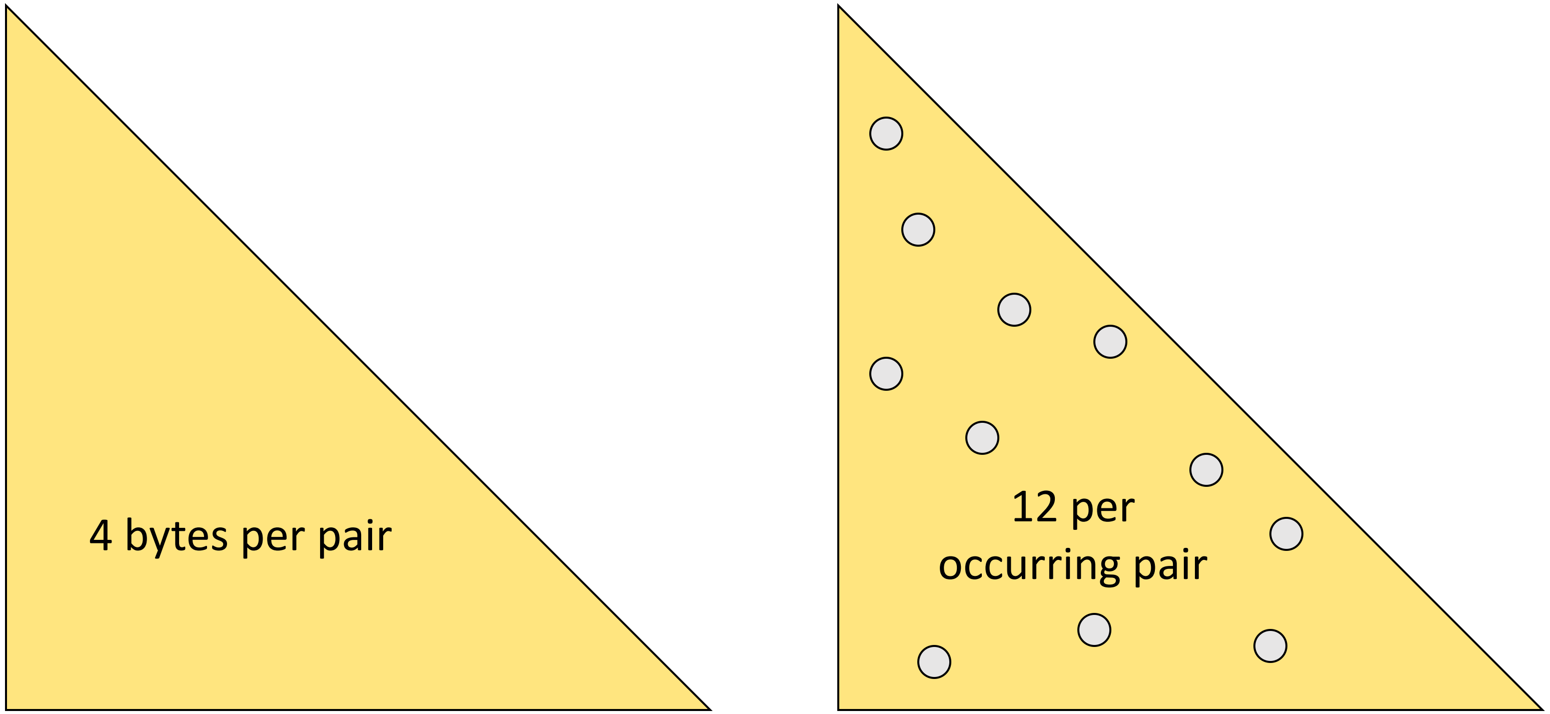
Two approaches for counting candidate pairs in memory:

- **Approach 1:** Count all pairs using a matrix
- **Approach 2:** Keep a table of triples $[i, j, c]$ = "the count of the pair of items $\{i, j\}$ is c."
  - If integers and item ids are 4 bytes, we need approximately 12 bytes for pairs with count > 0
  - Plus some additional overhead for the hashtable

Note:

- **Approach 1** only requires 4 bytes per pair
- **Approach 2** uses 12 bytes per pair (but only for pairs with count > 0)

3.3.4 Counting Candidate Pairs



Approach 1: Triangular Matrix (三角矩阵方法)

Approach 2: Triples (三元组方法)

3.3.4 Counting Candidate Pairs

- **Approach 1: Triangular Matrix (三角矩阵方法)**
  - $\boldsymbol{n}$ = total number items
  - Count pair of items $\{i, j\}$ only if $i<j$
  - Keep pair counts in lexicographic order, {1,2}, {1,3},…, {1,n}, {2,3}, {2,4},…,{2,n}, {3,4},….Then, pair $\{i, j\}$ is at position $(i-1)(n-i/2)+j-1$
  - Triangular matrix requires 4 bytes per pair
  - Total number of pairs $\boldsymbol{n(n-1)/2}$; total bytes= $\boldsymbol{2n^2}$
- **Approach 2: Triples(三元组方法)** uses **12 *bytes*** per occurring pair **(but only for pairs with count > 0)**
  - Beats Approach 1 if less than **1/3** of possible pairs actually occur

3.3.4 Counting Candidate Pairs

❑ Approach 1: Triangular Matrix (三角矩阵方法)

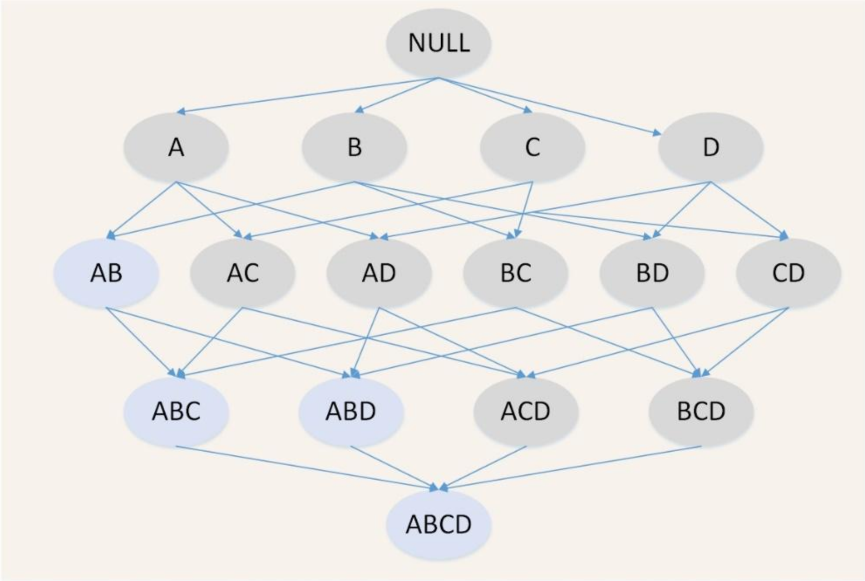
- n = total number items
- Count pair of items $\{i, j\}$ only if $i<j$
- Keep pair … {1,3},..., {1,n}, {2,3}, {2,4},...,{2,n … 1)(n– i/2) + j – 1
- Triangular …
- Total numb…

❑ Approach … *es* per occurring pair **(but o…**

- Beats Appr… actually occur

Problem is if we have too many items so the pairs do not fit into memory.

Can we do better?

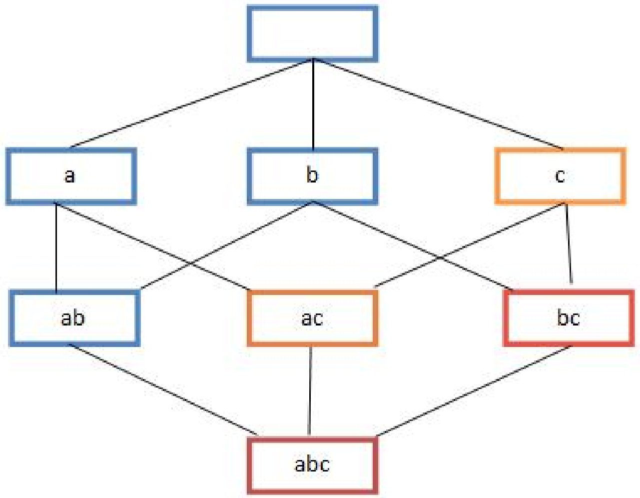


Section 3.4: A-Priori Algorithm

Content

3.4.1 Key idea of A-Priori

- A **two-pass** approach called ***A-Priori* (先验算法)** limits the need for main memory
- **Key idea: *monotonicity*** (单调性)
  - If a set of items I appears at least s times, so does every **subset** J of I



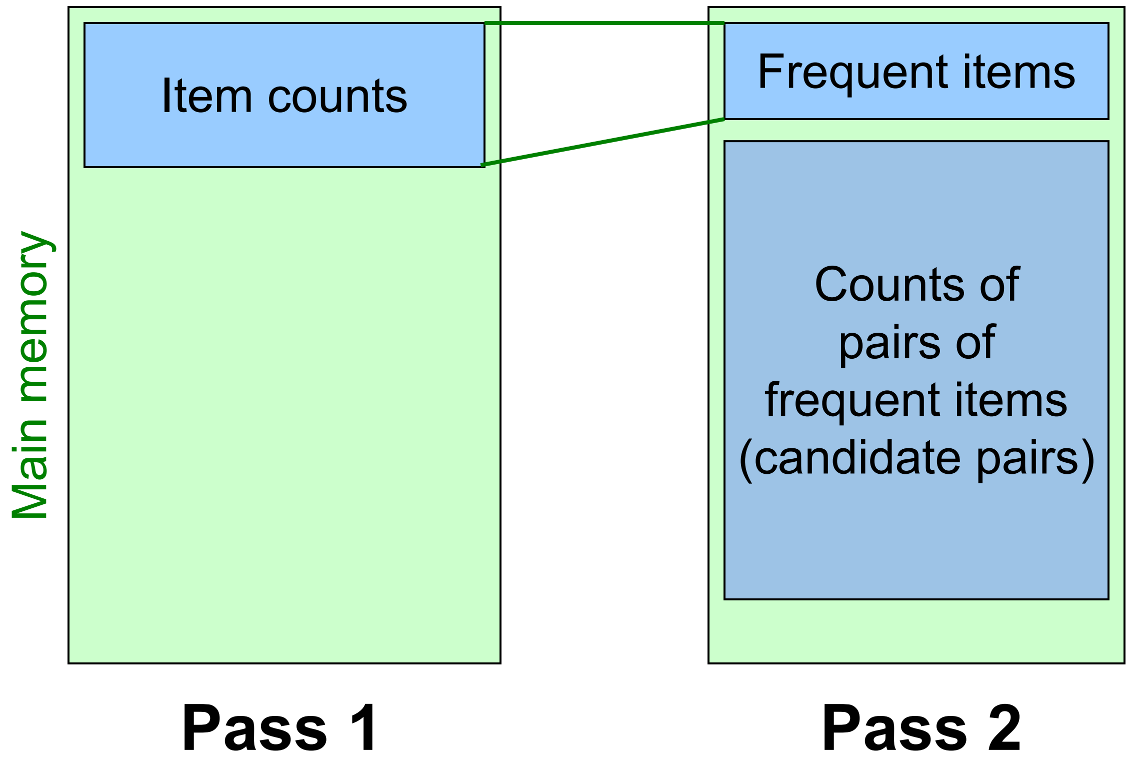
- **Contrapositive for pairs:**
  If item i does not appear in s baskets, then no pair including i can appear in s baskets
- **So, how does A-Priori find freq. pairs?**

3.4.2 A-Priori Algorithm

- **Pass 1:** Read baskets and count in main memory the occurrences of each **individual item**
  - Requires only memory proportional to #items
  - **Items that appear $\geq s$ times are the frequent items**
- **Pass 2:** Read baskets again and count in main memory only those pairs where both elements are frequent (from Pass 1)
  - 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)

3.4.2 A-Priori Algorithm

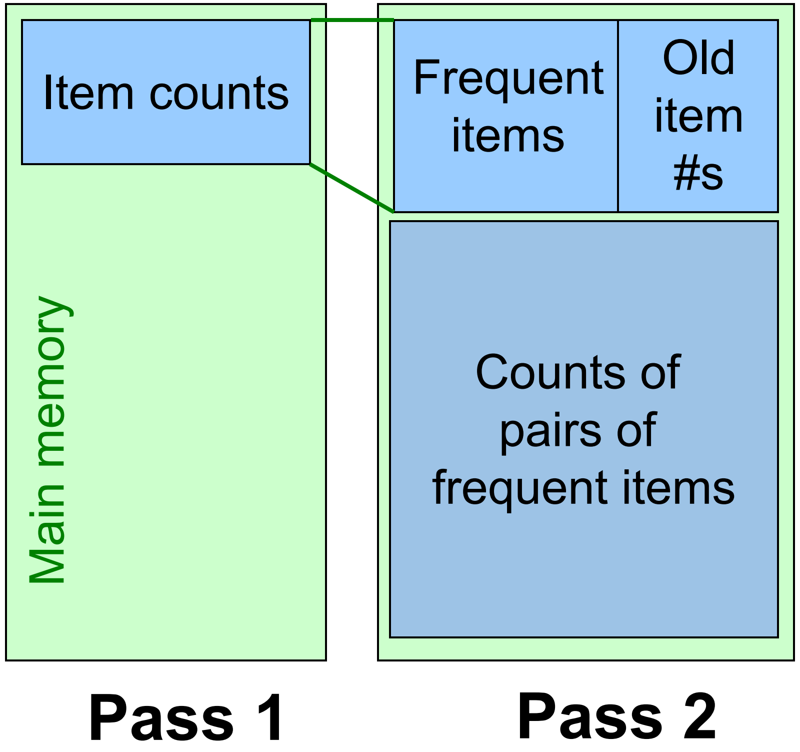
- Main-memory picture of A-Priori:



Note: Green box represents the amount of available main memory. Smaller boxes represent how the memory is used.

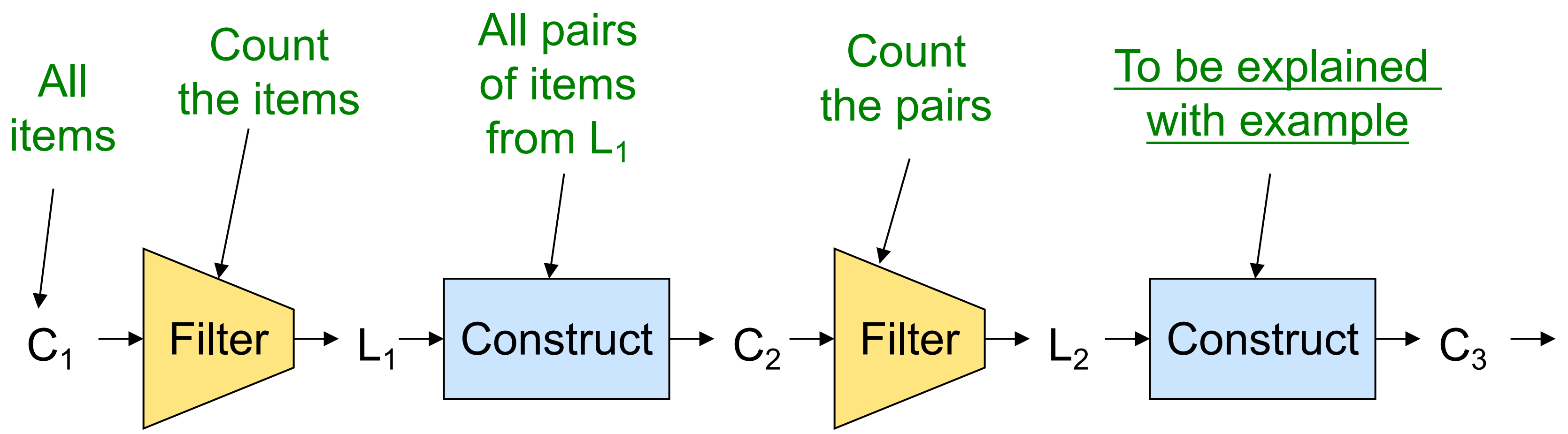
3.4.2 A-Priori Algorithm

- Using triangular matrix (三角矩阵方法) or triples(三元组方法) in Pass 2?
- 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 (频繁项表格)



3.4.3 Frequent K Tuples

- **For each k, we construct two sets of *k-tuples*** (sets of size k):
  - C_k = ***candidate k-tuples*** = 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-tuples



Example: Hypothetical steps of the A-Priori algorithm

- 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,c}, {b,m}, {c,m}, {c,j}}
- Generate C_3 = {~~{b,c,j}~~, {b,c,m}, ~~{b,m,j}~~, ~~{c,m,j}~~}
- Count the support of itemsets in C_3
- Prune non-frequent: L_3 = {{b,c,m}}

** Note here we generate new candidates by **generating C_k from L_{k-1} and L_1**. But that one can be more careful with candidate generation. For example, in C_3 we know {b,m,j} cannot be frequent since {m,j} is not frequent. Also {b,j} is not frequent.

3.4.3 Frequent K Tuples

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

- Example: Using A-Priori algorithm to find frequent itemsets where support s=2.

  B1={1,3,4} **B2={2,3,5}**

  B3={1,2,3,5} **B4={2,5}**

- Ans:
  - C_1={{1},{2},{3},{4},{5}}, L_1 ={{1},{2},{3},{5}}
  - C_2 ={{1,2},{1,3},{1,5},{2,3},{2,5},{3,5}}, L_2 ={{1,3},{2,3},{2,5},{3,5}}
  - C_3={{2,3,5}}, L_3 ={{2,3,5}}
  - Therefore, frequent itemsets: {{1},{2},{3},{5},{1,3},{2,3},{2,5},{3,5},{2,3,5}}

3.4.4 Weaknesses of A-Priori

- 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

- **What if A-Priori runs out of memory for k = 2?**