Big Data Analytics-Selected Topics (60-475)

Frequent Itemset Mining and Association Rules

Lecture 3

School of Computer Science
Faculty of Science
University of Windsor

Dr. Mehdi Kargar
Winter 2017

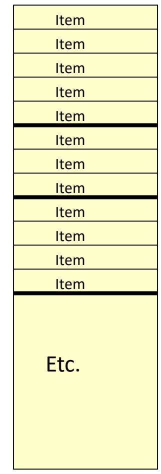
Outline

- In last lecture, we defined frequent itemsets and association rules
  - **{Milk} --> {Coke}**
  - **{Diaper, Milk} --> {Beer}**

- In this lecture, we see algorithms for finding frequent itemsets

Itemsets: Computation Model

- Data is often 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***
      - Can we do it **recursively**?



Items are positive integers

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

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** an algorithm makes over the data

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 of items
  - The number of different things we can count is limited by main memory
  - What happens if we don't have enough space to keep all pairs?
    - **Swapping** between main memory and disk is a **disaster** (**why?**)

Finding Frequent Pairs

- **The hardest problem often turns out to be finding the frequent pairs of items *{i, j}***
  - **Why?** Frequent pairs are **common**, frequent 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**

How to keep the count of pairs?

- First, convert items to integers
  - We can do this using a **hash table**
  - We assume our items are numbered from **1** to **n**
  - Assume Beer is mapped to 5 and Milk is mapped to 71
- Then, next step is how to **keep the counts** of the number of times that each pair ***{i, j}*** is appeared in all baskets
  - {5, 71} has appeared in 56,000 baskets

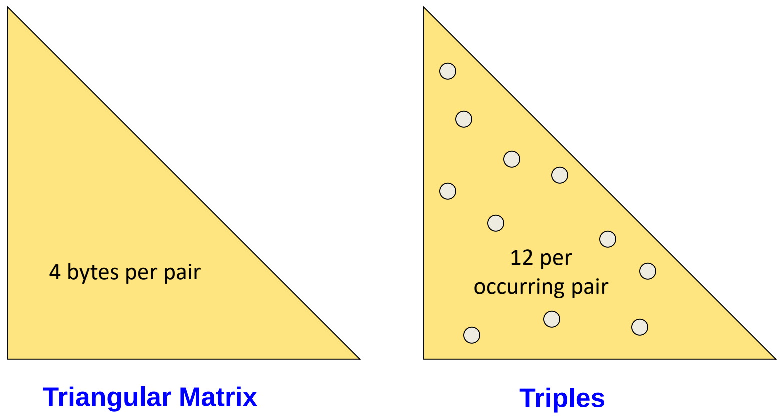
- **Naïve approach**
  - Use a **two dimensional array** ***a[i,j]***, and assume ***i < j***
    - What is wrong with this method?!
    - Remember we may have 100,000 items

Two approaches for counting pairs in memory

- **Approach 1:** Count all pairs using a **one dimensional triangular array**
  - We store in a[k], the count for pair {i,j}, with $1 \le i < j \le n$
  - **$k = (i - 1)(n - i/2) + j - i$**
- **Approach 2:** Keep **triples [*i*, *j*, *c*]** as:
  - The count of the pair of items {*i*, *j*} is *c*. **We only keep count if c > 0**
  - We need a **hash table** with **i** and **j** as search key to **quickly** find **[*i*, *j*, *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 hash table

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

Comparing the 2 Approaches



Comparing the two approaches

- **Approach 1: Triangular Matrix**
  - **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},…
  - Pair {*i*, *j*} is at position $(i-1)(n-i/2)+j-1$
  - Total number of pairs $\binom{n}{2}$; total bytes = $\mathbf{2n^2}$
    - Note: we approximate $\binom{n}{2} = \frac{n^2}{2}$
  - **Triangular Matrix** requires 4 bytes per pair
- **Approach 2** uses **12 *bytes*** per occurring pair
  - *(but only for pairs with count > 0)*
- **Approach 2 beats Approach 1 if less than 1/3 of possible pairs actually occur**

Now, the main challenge is how to **find (count) frequent pairs** efficiently?

Naïve approach for finding frequent pairs

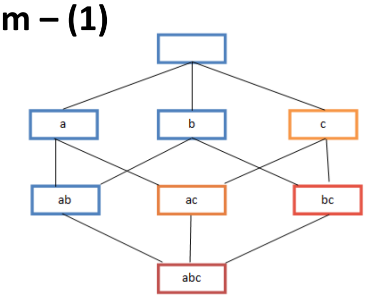
- Read **file once**, counting in main memory the occurrences of each pair:
  - From **each basket** of ***m*** items, generate its ***m(m-1)/2*** pairs by two nested loops
- **Naïve approach fails if (#items)^2 exceeds main memory**
  - Even if we use the two efficient approaches for keeping the counts
  - **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!
      - For only 100,000 items

Can we do better?!

A-Priori Algorithm

A-Priori Algorithm – (1)

- A **two-pass** approach called ***A-Priori*** limits the need for main memory

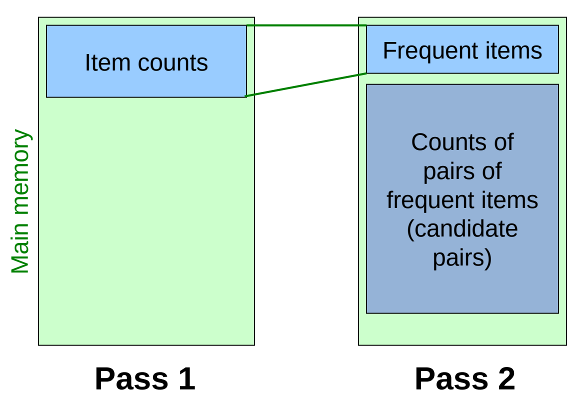


- **Key idea: *monotonicity***
  - If a set of items I appears at least s times, so does every **subset** J of I
- **Contrapositive for pairs:**
  If item i does not appear in s baskets, then no pair including i can appear in s baskets
- **So, how does A-Priori find frequent pairs?**

A-Priori Algorithm – (2)

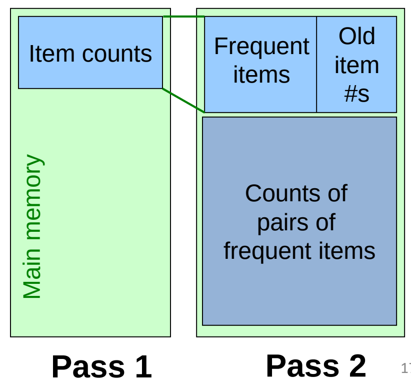
- **Pass 1:** Read baskets and count in main memory the occurrences of each **individual item**
  - Requires only memory proportional to **#items** not **(#items)^2**
- **Items that appear $\geq s$ times are the frequent items**
- **Pass 2:** Read baskets again and count in main memory **only** those 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)

Main-Memory: Picture of A-Priori



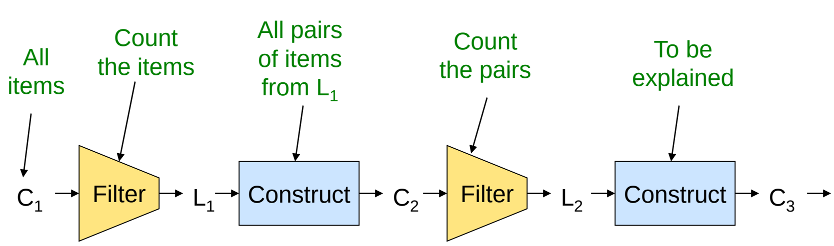
Detail for A-Priori

- You can use the triangular matrix method with:
  - *m* = 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



Frequent Triples, Etc.

- **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,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} }

****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

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:**
  - Association rules with intervals:
    - For example: Men over 65 have 2 cars
  - Association rules when items are in a taxonomy
    - Bread, Butter → FruitJam
    - BakedGoods, MilkProduct → PreservedGoods
  - Lower the support ***s*** as itemset gets bigger

Big Data Analytics

Quiz I

- Suppose there are 100,000 items, and 10,000,000 baskets of 10 items each
- Assume that we have the extreme case in which every pair of items appeared only once
- Which method is better to store the pairs of items in main memory?
  - one dimensional triangular array **OR**
  - triples [i, j, c]
- Assume all items are stored as integers

21

Big Data Analytics

Answer to Quiz I

- **One dimensional triangular array**:
  - We need to store $\binom{100{,}000}{2}$ = **5×10^9** integers
    - Note: we approximate $\binom{n}{2} = \frac{n^2}{2}$
- Total number of pairs among all the baskets:
  - $10^7\binom{10}{2} = 4.5 \times 10^8$
  - Thus, the maximum number of nonzero pairs (in the extreme case in which every pair of items appeared only once) is 4.5×10^8
- **Triples [i, j, c]:**
  - Maximum number of non-zero pairs is 4.5×10^8
  - We need three times this number for the triples method: **1.35×10^9**
- **The winner is Triples!**

22

Quiz II

- Compute frequent **pairs** for the baskets below with Apriori Algorithm
- Assume threshold *s = 3*
  a) {1, 2, 4, 5, 8, 9}
  b) {1, 4, 7, 8, 9}
  c) {1, 2, 5, 9}
  d) {1, 2, 5, 7, 8}
  e) {1, 2, 8, 10}

Answer to Quiz II

- Compute frequent pair for the baskets below with Apriori Algorithm. Assume threshold s = 3.
  a) {1, 2, 4, 5, 8, 9}
  b) {1, 4, 7, 8, 9}
  c) {2, 5, 6, 9}
  d) {1, 2, 3, 7, 8}
  e) {1, 2, 8, 10}
- Answer:
  - **Pass 1**:
    - Frequent items with count greater or equal 3 are: 1, 2, 8, 9
  - **Pass 2**:
    - Frequent pairs among frequent items are: {1,2}, {1,8}, {2, 8}