DSA5101 Introduction to Big Data for Industry

AY2025/26 Sem1 By Zhao Peiduo

Lecture 1

Frequent Itemsets and Association Rules

Association Rule Discovery Market-basket model:
- **Goal:** Identify items that are bought together by many supermarket customers.
- **Approach:** Process sales data of each customer to find dependencies among items.

The Market-Basket Model
- A large set of **items** — e.g., things sold in a supermarket.
- A large set of **baskets** — each basket is a small subset of items (what a customer buys).
- Want to discover **association rules** — e.g., people who bought $\{x, y, z\}$ tend to buy $\{v, w\}$.
- A many-to-many mapping between two kinds of things — connections among **items**, not baskets.
- Example:

Input:

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Output: Rules Discovered: {Milk → Coke}, {Diaper, Milk → Beer}

Frequent Itemsets
- **Simplest question:** Find sets of items that appear together *frequently* in baskets.
- **Support** for itemset I: Number of baskets containing all items in I (often as a fraction of total baskets).
- Given a **support threshold** s: sets of items appearing in at least s baskets are **frequent itemsets**.
- Example:Support of {Beer, Bread} = 2 baskets.

Association Rules
- $\{i_1, \ldots, i_k\} \to j$ means: "if a basket contains all i's, it's *likely* to contain j".
- **Confidence:** Probability of j given $I = \{i_1, \ldots, i_k\}$

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

Interesting Association Rules
- Not all high-confidence rules are interesting — e.g., $X \to$ milk might be high just because milk is common.
- **Interest:** Difference between confidence and the fraction of baskets containing j

$$\mathrm{Interest}(I \to j) = |\mathrm{conf}(I \to j) - \Pr[j]|$$

- Interesting rules have high positive or negative interest values (usually above 0.5).

Finding Association Rules Problem: Find all rules with support $\geq s$ and confidence $\geq c$.

Mining Association Rules
- Find all frequent itemsets I.
- **Rule generation:** For every subset A of I, generate a rule $A \to I \setminus A$ with:

$$\mathrm{conf}(A \to I \setminus A) = \frac{\mathrm{support}(I)}{\mathrm{support}(A)}$$

- **Observation 1:** Single pass over subsets of I to compute confidence.
- **Observation 2:** Monotonicity — if $B \subset A \subset I$, then

$$\mathrm{conf}(B \to I \setminus B) \leq \mathrm{conf}(A \to I \setminus A)$$

- Use monotonicity to prune rules below confidence threshold.

Itemsets: Computation Model
- Data is typically kept in flat files:
  - Stored on disk, too large to fit in main memory.
  - Stored basket-by-basket (e.g., $20, 52, 38, -1, 40, 22, -1, 20, 22, -1, \ldots$)
- **Major cost:** Time taken to read baskets from disk to memory.
- Assume baskets are small — a block of baskets can be expanded in main memory to generate all subsets of size k via k nested loops.
- Large subsets can often be ruled out using **monotonicity**.

Communication Cost is Key
- Running time $\propto$ #passes through data $\times$ data size.
- A pass = reading all baskets sequentially.
- Since data size is fixed, measure speed by number of passes.

Main-Memory Bottleneck
- For many algorithms, main memory is the limiting factor.
- While reading baskets, we need to count occurrences (e.g., pairs).
- The number of distinct items we can count is limited by memory.

Finding Frequent Pairs
- Goal: Find frequent pairs $\{i_1, i_2\}$.
- Frequent pairs are common; frequent triples are rare.
- Probability of being frequent drops exponentially with set size.
- **Approach:**
  - First focus on pairs, then extend to larger sets.
  - Generate all itemsets, but keep only those likely to be frequent.

Counting Pairs in Memory
- **Approach 1:** Use a matrix to count all pairs.
- **Approach 2:** Store triples $[i, j, c]$ meaning count of pair $\{i, j\}$ is c.
- Memory usage:
  - Approach 1: 4 bytes per pair.
  - Approach 2: 12 bytes per pair with count > 0.

Comparing the Two Approaches
- Triangular matrix: Count only if $i < j$, needs $2n^2$ bytes total.
- Approach 2 wins if less than 1/3 of possible pairs occur.

If Memory Fits All Pairs:
1. For each basket, double loop to generate all pairs.
2. Increment count for each generated pair.

A-Priori Algorithm
- A two-pass algorithm that limits memory usage.
- **Key idea: Monotonicity** — If I is frequent, every subset $J \subset I$ is also frequent.
- Contrapositive for pairs: If i is infrequent, no pair containing i can be frequent.
- Steps:
  - **Pass 1:** Count each item; items with count $\geq s$ are frequent.
  - **Pass 2:** Count only pairs where both items are frequent.

Memory Requirement:
- Pass 1: Memory $\propto$ number of items.
- Pass 2: Memory $\propto$ square of number of frequent items.
- Often, frequent items $\ll$ total items for a good threshold s.

Detail for A-Priori
- Can use the triangular matrix method with n = number of frequent items.
- May save space compared with storing triples.

Trick:
- Re-number frequent items $1, 2, \ldots$ and keep a table relating new numbers to original item numbers.

Frequent Triples, Etc.
- For each k, construct two sets of *k-tuples*:
  - C_k = **candidate** *k-tuples*: sets that might be frequent (support $\geq s$) based on info from pass for $k-1$.
  - L_k = the set of truly frequent *k-tuples*.

Candidate Itemset Generation (details) To form C_k from L_{k-1}: For every two itemsets in L_{k-1} with exactly $k-2$ common items, take their union to generate an itemset of size k. Prune itemsets where any subset of size $k-1$ is not in L_{k-1}.

A-Priori for All Frequent Itemsets
- One pass for each k (itemset size).
- Needs memory to count each candidate *k-tuple*.
- For typical market-basket data and reasonable support (e.g., 1%), $k = 2$ requires the most memory.

Possible extension: Lower the support s as itemset size increases.

Hash Functions
- **Definition:** A hash function h takes a *hash-key value* and produces a *bucket number* $\in [0, B-1]$.
- Should randomize hash-keys roughly uniformly into buckets.

Indexing using Hash Functions
- Used for indexing to enable fast search/retrieval.
- **Example:** Hash the name to the ordinal position of the first letter, use as bucket index.

PCY (Park-Chen-Yu) Algorithm
- **Observation:** In pass 1 of A-Priori, most memory is idle — only individual item counts are stored.
- Use spare memory in pass 1 to prune candidate pairs for pass 2.

Pass 1 of PCY:
- Maintain a hash table with as many buckets as memory allows.
- Count number of pairs hashed into each bucket (store counts, not the pairs).

PCY Algorithm — First Pass
- For each basket:
  - Count each item's frequency.
  - For each pair: hash to a bucket and increment that bucket's count.
- Only store bucket counts.

Using hash buckets to prune candidate pairs
- If a bucket count $< s$, none of its pairs can be frequent — prune them.
- If a bucket contains a frequent pair, it is frequent.

PCY Algorithm — Between Passes
- Replace buckets with a bit vector: 1 if count $\geq s$, else 0.
- Bit vector uses 1/32 memory of integer counts.
- Also decide frequent items for 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 whose bit in the bit vector is 1 (i.e., a frequent bucket)
- Both conditions are necessary for the pair to have a chance of being frequent

Main-Memory Details
- Note: We do not need to count a bucket past s counts.
- On second pass, a table of (item, item, count) triples is essential (cannot use triangular matrix approach).
- Hash table must eliminate approximately 2/3 of the candidate pairs for PCY to beat A-Priori.

Refinement: Multistage Algorithm
- Using an additional pass (3 passes), we further prune our set of candidate pairs.
- Key idea: After Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY (satisfying both conditions):
  - i and j are frequent
  - $\{i, j\}$ hashes to a frequent bucket from Pass 1

Multistage – Pass 2

- Using a new independent hash function, only hash pairs $\{i, j\}$ if:
  1. Both i and j are frequent
  2. $\{i, j\}$ hashed to a frequent bucket with the first hash function
- Remark: The second hash table is slightly smaller ($\frac{31}{32}$ of first), but fewer frequent buckets are expected due to stricter conditions.

Multistage – Pass 3

- Count only pairs $\{i, j\}$ that satisfy:
  1. Both i and j are frequent
  2. Using first hash function, $\{i, j\}$ hashes to a frequent bucket in first bit-vector
  3. Using second hash function, $\{i, j\}$ hashes to a frequent bucket in second bit-vector
- Remark: Second bitmap is slightly smaller; combined bitmaps occupy about $\frac{1}{16}$ of memory. Fewer candidate pairs than PCY.

Important Points

1. The two hash functions have to be independent
2. We need to check both hashes on the third pass

- Remark: More passes possible for pruning, but each requires an extra bitmap; eventually memory runs out.

Refinement: Multihash

- Key idea: Use several independent hash tables on the first pass.
- Risk: Halving the number of buckets doubles average count — must ensure most buckets stay below s.
- Benefit similar to multistage, but in 2 passes.

Multihash – Pass 2

- Same as Pass 3 of multistage:
  1. Both i and j frequent
  2. $\{i, j\}$ hashes to frequent bucket for both hash functions

Adding More Hash Functions

- Either multistage or multihash can use more than two hash functions.
- In multistage, diminishing returns as bit-vectors consume memory.
- In multihash, bit-vectors use same space as one PCY bitmap; too many hash functions cause most buckets to become frequent.