Mining Massive Datasets

Lecture 09

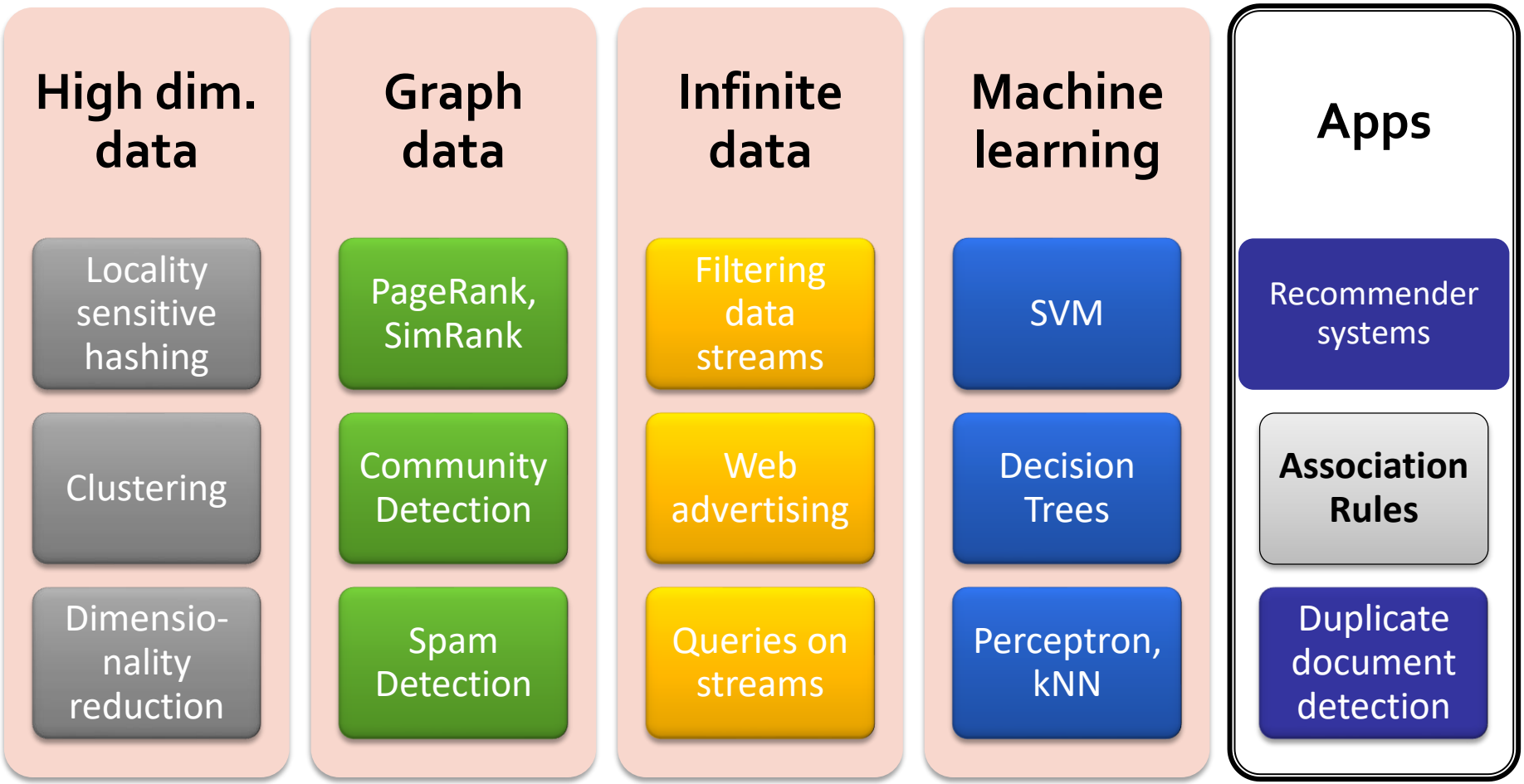
Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Note on Slides

A substantial part of these slides come (either verbatim or in a modified form) from the book Mining of Massive Datasets
by Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University).
For more information, see the website accompanying the book: http://www.mmds.org.

Current Topic



Association Rule Discovery

Supermarket shelf management: Market-basket model:

- **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 someone buys diaper and milk, then he/she is likely to buy beer

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
  - E.g., the things one customer buys on one day
- Goal: discover **association rules**
  - People who bought {x,y,z} tend to buy {v,w}
    - Amazon!

Input:

TID	*Items in a basket*
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}

Applications

- **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
- **Amazon's people who bought *X* also bought *Y***

Outline

First: Define

- **Frequent itemsets**
- **Association rules:** Confidence, Support, Interestingness

Then: Algorithms for finding frequent itemsets

- **Finding frequent pairs**
- **A-Priori algorithm**
- **PCY algorithm + refinements**

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 expressed as a fraction of the total number of baskets)
- Given a ***support threshold s***, ***frequent itemsets*** are sets of items that appear in at least ***s*** baskets

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

Support of {Beer, Bread} = 2

Example: Frequent Itemsets

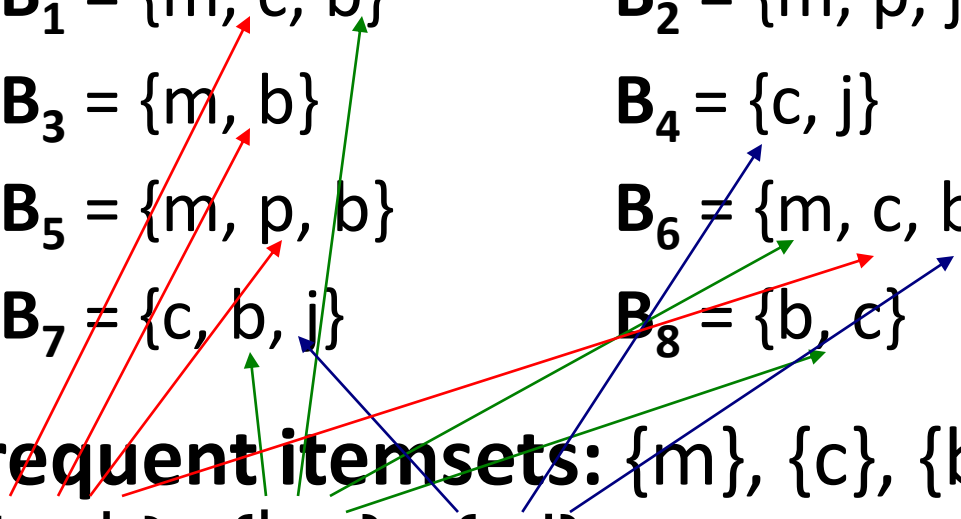
- **Items** = {milk, coke, pepsi, beer, juice}
- **Support threshold** = 3 baskets

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}.

Association Rules

- An **association rule R** is an "if-then" rule about the contents of baskets
- A-rule has form $\{i_1, i_2, \ldots, i_k\} \rightarrow \{j_1, j_2, \ldots, j_m\}$
- R applies ⇔ "if a basket contains all in $\{i_1, i_2, \ldots, i_k\}$ then it also contains $\{j_1, j_2, \ldots, j_m\}$"
- For $\boldsymbol{B} = \{m, c, b\}$, which a-rules apply?
- $\{m\} \rightarrow \{c, b\}$, $\{m, c\} \rightarrow \{b\}$
- In general, for every subset $\boldsymbol{A}$ of $\boldsymbol{I}$ we can generate an ass. rule $\boldsymbol{A \rightarrow I \backslash A}$
- But most rules are not significant – why?

Association Rules

- Rule $I \rightarrow J$ applies ⇔ "if a basket contains all in $I = \{i_1, i_2, \ldots, i_k\}$ then it also contains all in $J = \{j_1, j_2, \ldots, j_m\}$"
- Intuitively, "good" rules should have (B=basket) ..
  - High confidence: If $I \subseteq B$, then $J \subseteq B$ (= rule applies)
  - High rule support: Rule applies for *many* baskets
- Support of an ass-rule is the number of baskets containing both I and J, i.e. **support(I ∪ J)** (link)
  - Note: for each such basket rule applies!
- Confidence of an ass-rule is the probability that it applies if $I \subseteq B$:

$$\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup J)}{\text{support}(I)}$$

Interesting Association Rules

- Not all high-confidence rules are interesting
  - The rule $X \rightarrow milk$ may have high confidence for many itemsets X, because milk is just purchased very often (independent of X) and the confidence is high
- **Interest** of an association rule $I \rightarrow J$: difference between its confidence and the fraction of baskets that contain $J = \{j_1, .., j_p\}$

$$\text{Interest}(I \rightarrow J) = \text{conf}(I \rightarrow J) - \Pr[J]$$

  - Interesting rules are those with high positive or negative interest values (usually above 0.5)

Example: Confidence and Interest

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}

Set	m,b	m,b,c	c
B1	1	1	1
B2			
B3	1		
B4			1
B5	1		
B6	1	1	1
B7			1
B8			1

- **Association rule: {m, b} → c**
  - **Confidence** = 2/4 = 0.5
  - **Interest** = |0.5 – 5/8| = 1/8
    - Item ***c*** appears in 5/8 of the baskets
    - Rule is not very interesting!

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

Finding Association Rules

- **Problem:** Find all association rules with support $\geq s$ and confidence $\geq c$

- **Hard part:** Finding the frequent itemsets!
  - If $\{i_1, i_2, \ldots, i_k\} \rightarrow J$ has high support and confidence, then both $\{i_1, i_2, \ldots, i_k\}$ and $\{i_1, i_2, \ldots, i_k, j_1, \ldots, j_p\}$ will be "frequent"

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

Mining Association Rules

- **Step 1:** Find all frequent itemsets I
  - (we will explain this next)
- **Step 2:** **Rule generation**
  - For every subset A of I, generate a rule $A \rightarrow I \setminus A$
    - Since I is frequent, A is also frequent
    - **Variant 1:** Single pass to compute the rule confidence
      - confidence(***A,B*→*C,D***) = support(**A,B,C,D**) / support(**A,B**)
    - **Variant 2:**
      - **Observation:** If **A,B,C→D** is below confidence, so is **A,B→C,D**
      - Can generate "bigger" rules from smaller ones!
  - Output the rules above the confidence threshold

Example

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}

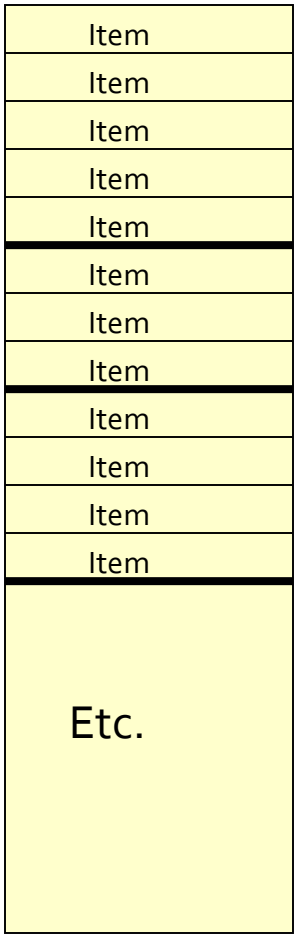
B_7 = {c, b, j} B_8 = {b, c}

- **Thresholds: support $s = 3$, confidence $c = 0.75$**
- **1) Frequent itemsets:**
  - **Singletons and {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 … **b,m→c**: c=3/4
  - ~~**b→c,m**: c=3/6~~

Finding Frequent Itemsets

Itemsets: Computation Model

- **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.

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

Bottleneck: Main-Memory

- 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

Finding Frequent Pairs

- **The hardest problem often turns out to be finding the frequent pairs 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
- First concentrate on pairs, then extend
- 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

Naïve Algorithm

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
- **Fails if (#items)^2 exceeds main memory**
  - **Remember:** #items can be on the order of 1M (Wal-Mart) or 10B (Web pages)
    - Suppose: 10^6 items, pair counts are 4-byte integers
    - Number of pairs of items: $10^6(10^6-1)/2 = 5*10^{11}$
    - Therefore, $2*10^{12}$ (2000 gigabytes) of memory needed

Counting Pairs in Memory

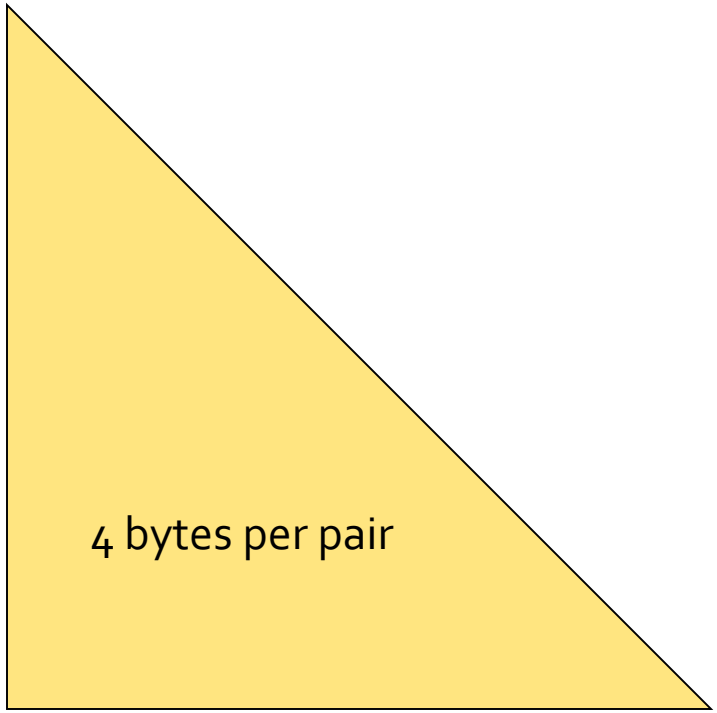
Two approaches:

- **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

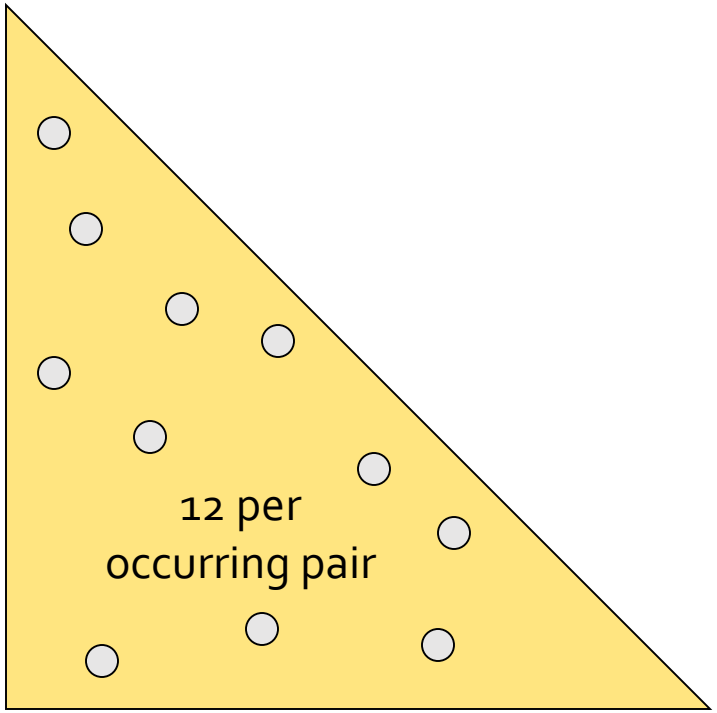
Note:

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

Comparing both Approaches



Triangular Matrix (dense)



Triples (sparse)

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 $\boldsymbol{n(n-1)/2}$; total bytes= $\boldsymbol{2n^2}$
  - **Triangular Matrix** requires 4 bytes per pair
- **Approach 2** 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*

Comparing the two approaches

- Approach 1: Triangular Matrix
  - n = total number items
  - Co
  - K
  - P
  - T
  - T
- Ap
(bu
  - Beats Approach 1 if less than 1/3 of possible pairs actually occur

2n2

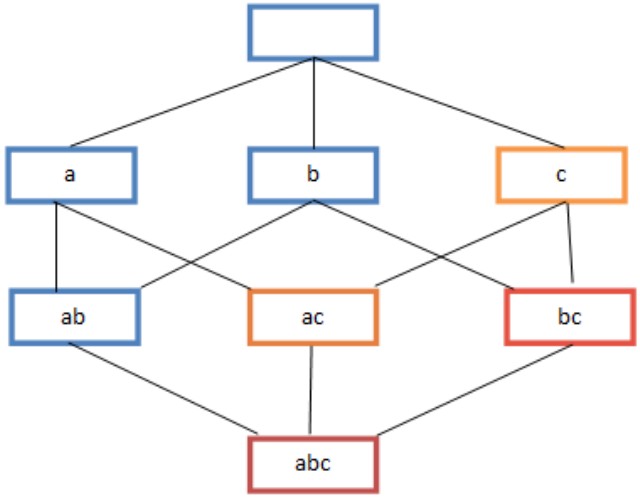
Problem is if we have too many items so the pairs do not fit into memory.

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



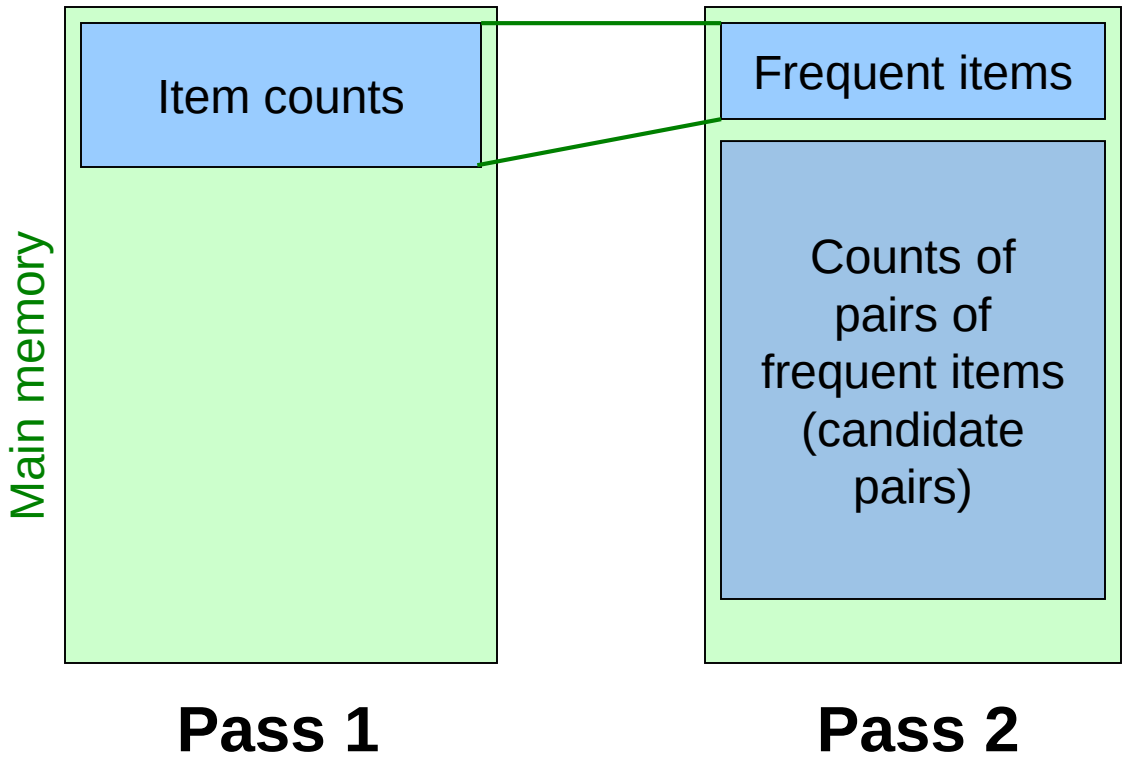
- **Contrapositive for pairs:**
  If item i does not appear in s baskets, then no pair including i can appear in s baskets

- **So, how does A-Priori find frequent pairs?**

A-Priori Algorithm – (2)

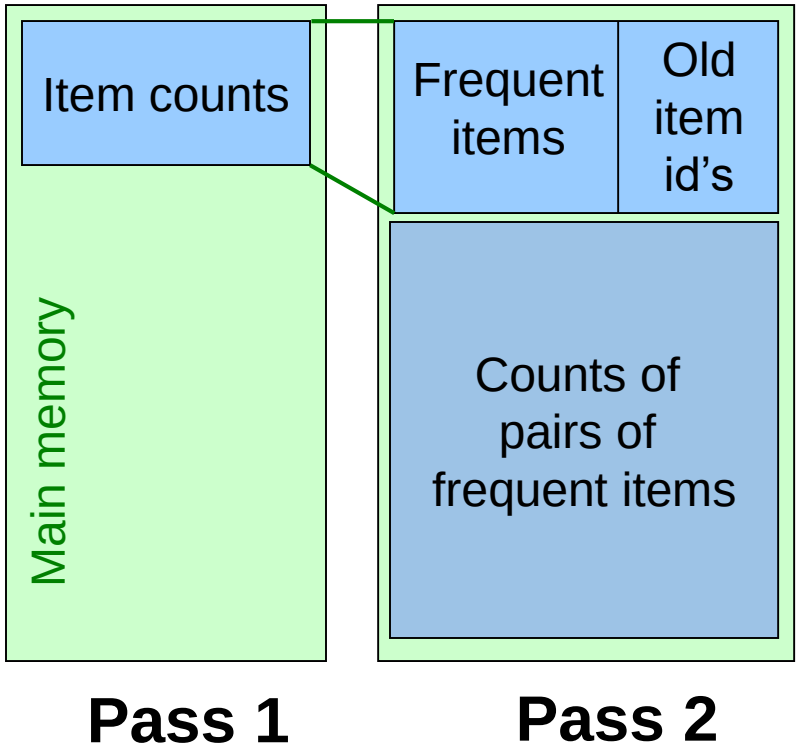
- **Pass 1:** Read baskets and count in main memory the occurrences of each **individual item**
  - Requires only memory proportional to #items
- **Items that appear $\geq s$ times are the frequent items**
- **Pass 2:** Read baskets again and count in main memory only those pairs where both elements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



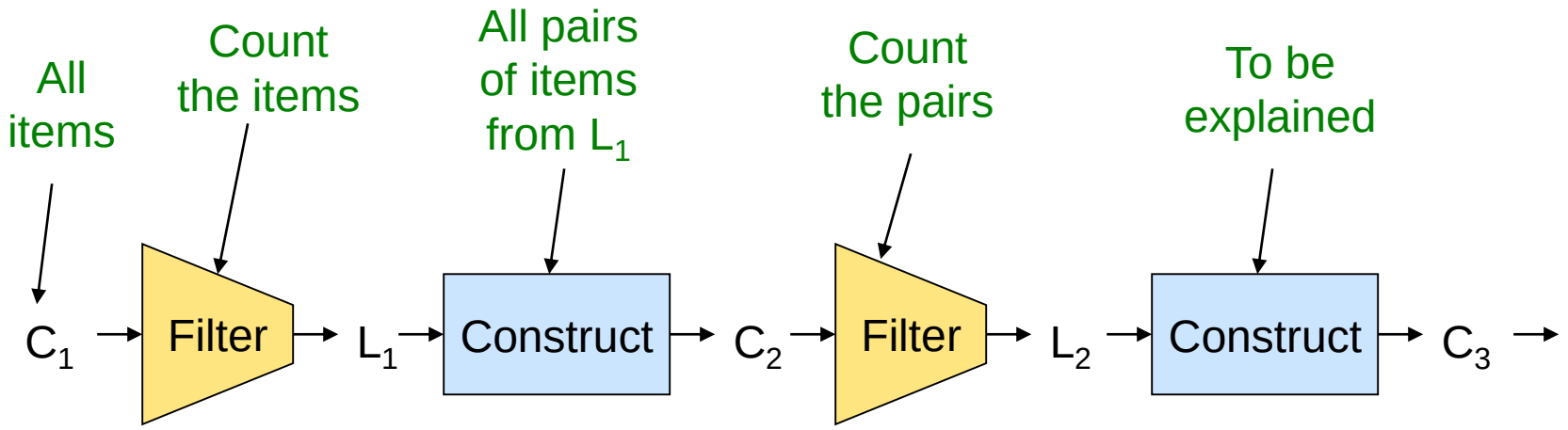
Detail for A-Priori

- You can use the triangular matrix method with ***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



Generalization: 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

** 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

- **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} }

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

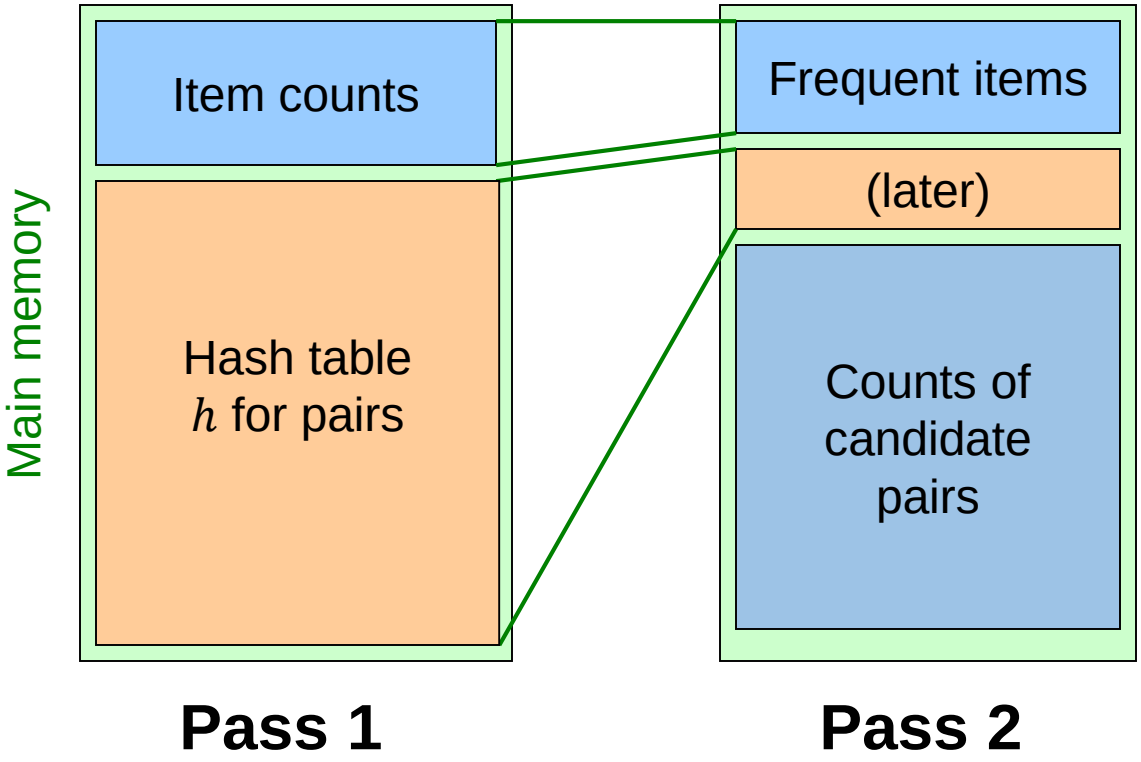
- **Many possible extensions:**
  - Association rules when items are in a taxonomy
    - Bread, Butter → FruitJam
    - BakedGoods, MilkProduct → PreservedGoods
  - Lower the support ***s*** as itemset gets bigger

PCY (Park-Chen-Yu) Algorithm

PCY (Park-Chen-Yu) Algorithm

- **Observation:**
  In pass 1 of A-Priori, most memory is idle
  - We store only individual item counts
- => Use this idle RAM to reduce RAM used in pass 2!
- **Pass 1 of PCY:** In addition to item counts, maintain a **hash table h** with as many buckets as fit in memory
  - Keep a **count** for each bucket into which **pairs** of items are hashed (*for each bucket just keep the count, not the actual pairs!)*
- *Why? If a pair p is frequent, "its" bucket $\boldsymbol{h}(p)$ will receive count above a threshold s for frequent pairs!*
- **Pass 2 of PCY:**
  Count only those pairs that hash to frequent buckets

Main-Memory: Picture of PCY



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;
```

New in PCY

- **Few things to note:**
  - Pairs of items need to be generated from the input file; they are not present in the file
  - Hash function should have many buckets, i.e. must be likely to hash *different pairs to different buckets*

Observations about Buckets

- **Observation:** If a bucket contains a frequent pair, then the bucket is surely frequent
- However, even without any frequent pair, a bucket can still be frequent ☹
  - So, we cannot use the hash to eliminate any member (pair) of a "frequent" bucket
- But, for a bucket with total count less than s, none of its pairs can be frequent ☺
  - All pairs that hash to this "infrequent" bucket can be eliminated as candidates

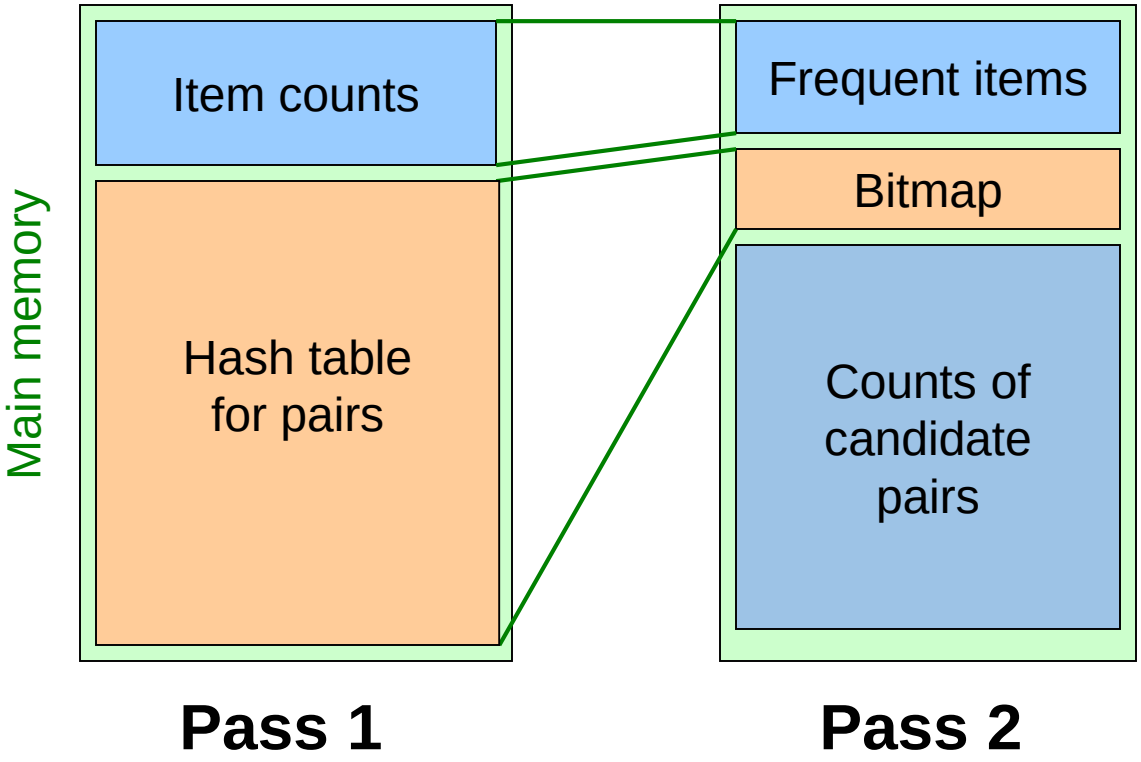
PCY Algorithm – Between Passes

- Replace the buckets by a bit-vector:
  - **1** means the bucket count exceeded the support s (call it a frequent bucket); **0** means it did not
- 4-byte integer counts 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 $\boldsymbol{i}$ and $\boldsymbol{j}$ 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: Picture of PCY



Main-Memory Details

- Buckets require a few bytes each:
  - **Note:** we do not have to count past s
  - #buckets is *O(main-memory size)*
- On second pass, a table of (item, item, count) triples is essential (we cannot use triangular matrix approach)
  - Thus, hash table must eliminate approx. 2/3 of the candidate pairs for PCY to beat A-Priori

Thank you.

Questions?

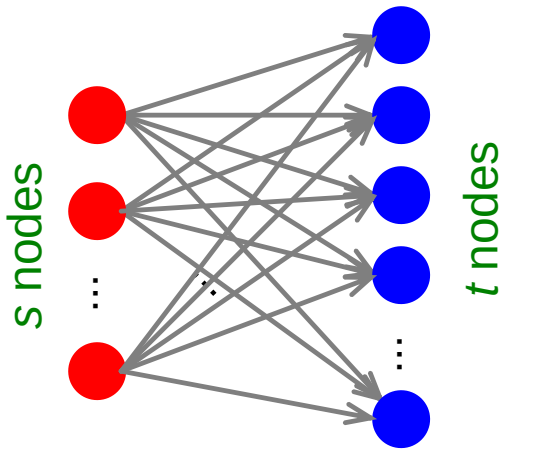
Additional Slides

Associations - More generally

- **A general many-to-many mapping (association) between two kinds of things**
  - But we ask about connections among "items", not "baskets"
- **For example:**
  - Finding communities in graphs (e.g., Twitter)

Example:

- **Finding communities in graphs (e.g., Twitter)**
- **Baskets** = nodes; **Items** = outgoing neighbors
  - Searching for complete bipartite subgraphs $K_{s,t}$ of a big graph



A dense 2-layer graph

- **How?**
  - View each node i as a basket B_i of nodes i it points to
  - $K_{s,t}$ = a set Y of size t that occurs in s buckets B_i
  - Looking for $K_{s,t}$ → set of support s and look at layer t – all frequent sets of size t

Compacting the Output

- To reduce the number of rules we can post-process them and only output:
  - **Maximal frequent itemsets:**
    No immediate superset is frequent
    - Gives more pruning

  or

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

Example: Maximal/Closed

	Support	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

Frequent, but superset BC also frequent.

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

Superset BC has same count.

Its only super-set, ABC, has smaller count.