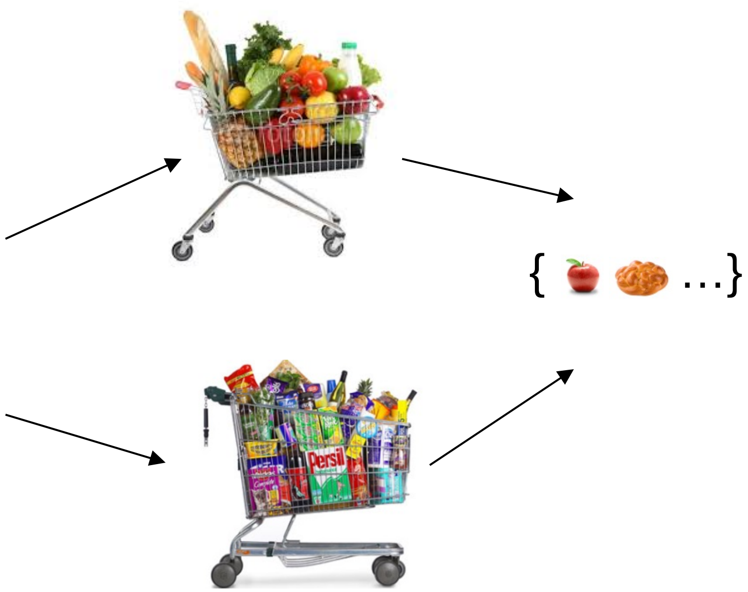
CS5344

Finding Frequent Itemsets



Acknowledgement: Slides adapted from http://www.mmds.org

Motivation

- **Association rule discovery**
  - Find associations between items in a dataset
- **Given a set of transactions, find rules that will predict the occurrence of an item based on the occurrence of other items in the transaction**

Market-Basket Transactions

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

Example Association Rules

{Milk} --> {Coke}
{Diaper, Milk} --> {Beer}

Implication means co-occurrence, not causality!

Market-Basket Model

- **Large set of items**
  - e.g., things sold in a supermarket
- **Large set of baskets, each basket is a small subset of items**
  - e.g., the things a customer buys in a shopping trip
- **Want to discover association rules**
  - People who bought Diaper tend to buy Beer

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

Application

- **A chain store keeps TBs of data about what customers buy together**
  - Sales data collected with barcode scanners
- **What can you do with such data?**
  - Reveals how customers typically navigate stores
    - Shelf management: position items strategically
  - Marketing and sales promotion
    - Suggests tie-in “tricks”, e.g., run sale on diapers but raise the price of beer

Frequent Itemsets

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

- **Itemset – collection of one or more items**
  - **e.g. {Milk, Bread}**
- **k-itemset: itemset that contains k items**
- **Support for itemset *I:* number of baskets containing all items in *I***
  - Often expressed as a fraction of the total number of baskets
  - e.g. support of {Milk, Bread} = 2/5
- **Frequent itemset: an itemset whose support is ≥ *minimum support threshold (minsup)***

Example

B1	**m, c, b**
B2	**m, p, j**
B3	**m, b**
B4	**c, j**
B5	**m, p, b**
B6	**m, c, b, j**
B7	**c, b, j**
B8	**b, c**

- **Items = {milk, coke, pepsi, beer, juice}**
- **Minimum support threshold = 3 baskets**
- **Frequent itemsets:**
  - {m}, {c}, {b}, {j} *1-itemsets*
  - {m, b}, {b, c}, {c, j} *2-itemsets*
- **{m, c} is not a frequent itemset**
  - only appear in two baskets
- ***Note: All items appearing in frequent 2-itemsets also appear in frequent in 1-itemsets***

Association Rules

- **IF-THEN rules about the contents of baskets**
  - ***$\{ i_1, i_2, \ldots, i_k \} \to j$*** means: "if a basket contains all of ***$i_1, \ldots, i_k$*** then it is ***likely*** to contain ***j***"
- **Rule evaluation metrics:**
  - **Support of a rule** is the fraction of baskets that contain the itemset ***$\{ i_1, \ldots, i_k, j \}$***
  - **Confidence** of this association rule is the probability of ***j*** given ***$I = \{i_1, \ldots, i_k\}$***

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

Measures how often item j occurs in baskets that contain I

Example

- **Association rule:** ***{m, b}*** $\rightarrow$ ***c***
- **Support = 2/8 = 0.25**
  - Fraction of baskets that contain the itemset {m, b, c}
- **Confidence = 2/4 = 0.5**

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

B1	**m, c, b**
B2	**m, p, j**
B3	**m, b**
B4	**c, j**
B5	**m, p, b**
B6	**m, c, b, j**
B7	**c, b, j**
B8	**b, c**

Task: Find Association Rules

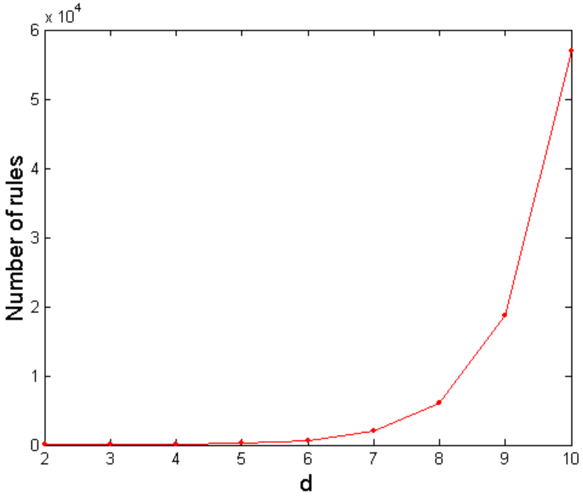
- **Find all association rules with support ≥ *minsup* and confidence ≥ *minconf***
- **Brute-force approach**
  - List all possible association rules

    Given d unique items:

    Total number of itemsets = 2^d

    Total number of rules = R

$$R = \sum_{k=1}^{d-1}\left[\binom{d}{k} \times \sum_{j=1}^{d-k}\binom{d-k}{j}\right] = 3^d - 2^{d+1} + 1$$



  - Compute support and confidence for each rule
  - Prune rules that fail the *minsup* and *minconf* thresholds
  - **Computationally prohibitive!**

Task: Find Association Rules

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

Example Rules:

{Milk,Diaper} → {Beer} (s=0.4, c=0.67)
{Milk,Beer} → {Diaper} (s=0.4, c=1.0)
{Diaper,Beer} → {Milk} (s=0.4, c=0.67)
{Beer} → {Milk,Diaper} (s=0.4, c=0.67)
{Diaper} → {Milk,Beer} (s=0.4, c=0.5)
{Milk} → {Diaper,Beer} (s=0.4, c=0.5)

- **Observations:**
  - All the rules are binary partitions of the same itemset: **{Milk, Diaper, Beer}**
  - Rules from the same itemset have identical support but can have different confidence
  - If an itemset does not satisfy minsupp, do not need to generate the rules

Mining Association Rules

- **Step 1: Find all frequent itemsets *I***
  - Generate all itemsets whose support $\geq$ minsup
- **Step 2: Generate rules**
  - For every subset X of I, generate $X \rightarrow I - X$
    - If {A,B,C,D} is a frequent itemset, the candidate rules are:

ABC → D	**ABD → C**	**ACD → B**	**BCD → A**
A → BCD	**B → ACD**	**C → ABD**	**D → ABC**
AB → CD	**AC → BD**	**AD → BC**	**BC → AD**
BD → AC	**CD → AB**		

    - If $|I| = k$, we have $2^k - 2$ candidate association rules that involve all the attributes (ignoring $I \rightarrow \emptyset$ and $\emptyset \rightarrow I$)
    - Output rules above confidence threshold

Example

- *minsup = 3*
- *minconf = 0.75*

(1) Frequent itemsets

{b,m}, {b,c}, {c,m}, {c,j}, {m, c, b}

(2) Generate rules:

~~b → m: conf = 4/6~~

m → b: conf = 4/5

b → c: conf = 4/6

…

~~b, c → m: conf = 3/5~~

b, m → c: conf = 3/4

~~b → c, m: conf = 3/6~~

B1	**m, c, b**
B2	**m, p, j**
B3	**m, b**
B4	**c, j**
B5	**m, p, b**
B6	**m, c, b, j**
B7	**c, b, j**
B8	**b, c**

Rule Generation

- **How to generate rules from frequent itemsets efficiently?**
- **Observations:**
  - Confidence of rules generated from the same itemset has

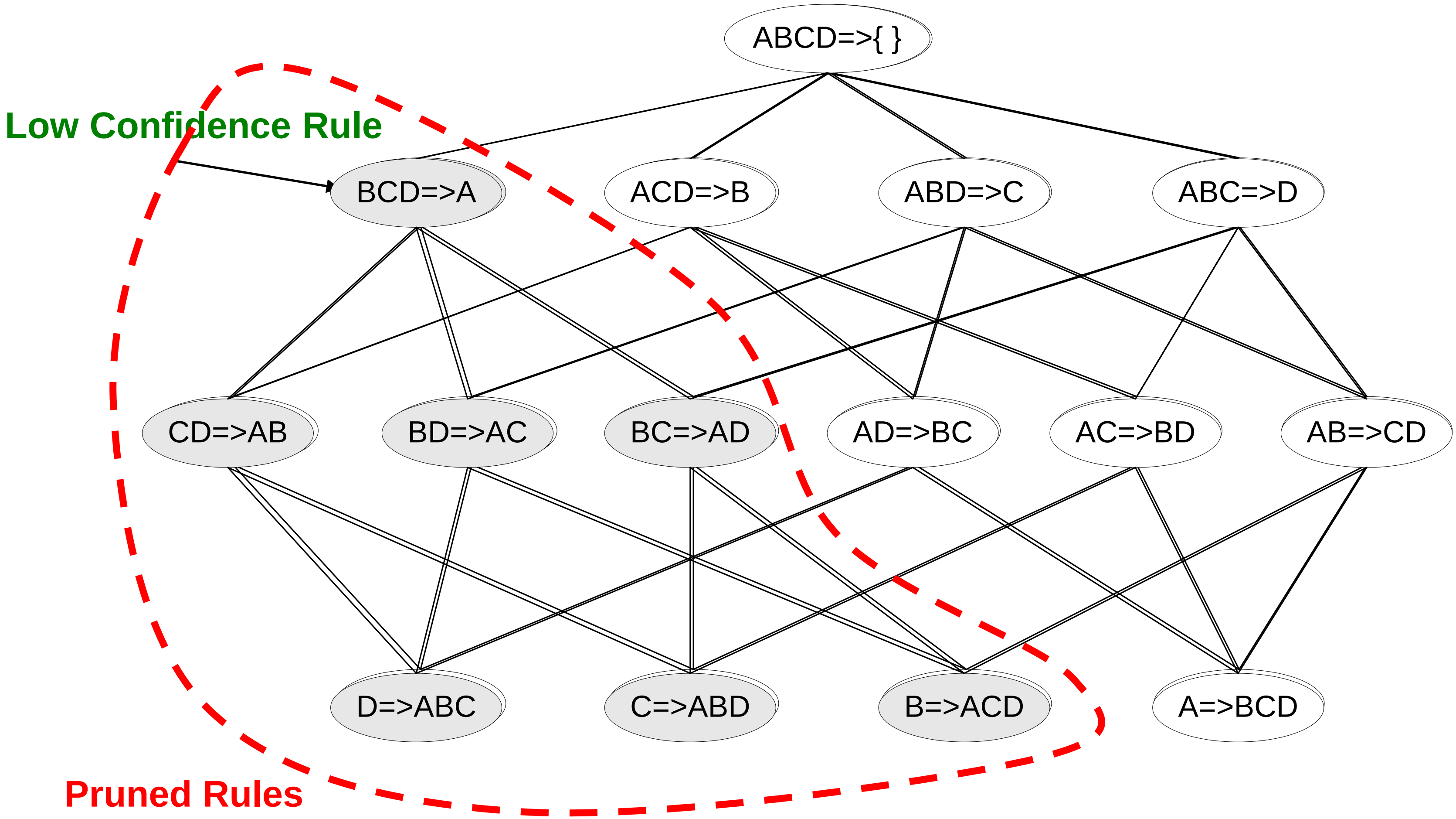
    anti-monotone property: **If $X' \subseteq X \rightarrow f(X') \geq f(X)$**
  - If I = {A,B,C,D}, then

    $\text{conf}(ABC \rightarrow D) \geq \text{conf}(AB \rightarrow CD) \geq \text{conf}(A \rightarrow BCD)$

Recall: $\text{conf}(X \rightarrow Y) = \text{support}(X \cup Y) / \text{support}(X)$

Rule Generation

- **Lattice of rules**



Task: Find Frequent Itemsets

- **To find frequent itemsets, we need to count**
- **To count, we need to generate them**
- **Turns out that finding frequent pairs of items is the hardest**
  - Frequent pairs are common, frequent triples are fewer, quadruples are rare.
  - Probability of being frequent drops exponentially with size

Finding Frequent Item Pairs

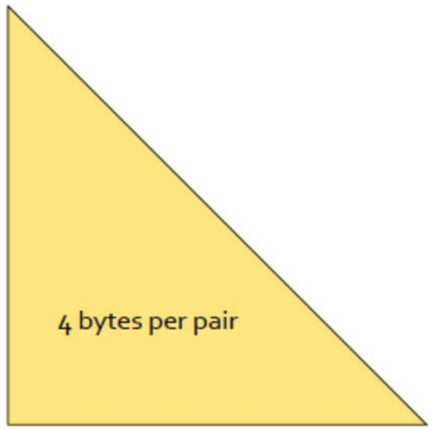
- **Naïve method: Read baskets once, count in main memory the occurrences of each pair**
  - From each basket of ***n*** items, generate its ***n(n-1)/2*** pairs by two nested loops.
  - If *I* = {*A, B, C, D*}, we have *AB, AC, AD, BC, BD, CD*
- **What if $(\#items)^2$ exceeds main memory?**
  - E.g. Walmart has over 100K items
  - Assume counts are 4-byte integers
  - Number of pairs of items: 100,000(100000 - 1)/2 = $5*10^9$
  - Therefore, 20 GB of memory needed

For machine with 2 GB memory, $n < 2^{15}$ or 33,000

Counting Pairs in Memory

- **How to store the $n(n-1)/2$ counts so that we can quickly find the count for a pair of items?**
- **Represent items by integers 1 to n**
- **Count pairs of items *{i, j}* only if *i < j***
- **Use a 2-dimensional array M where entry M[i, j] gives the count of item pair {i, j} with $1 \le i < j \le n$.**
  - Half of the array is wasted
- **Use Triangular Matrix or Triples Method**

Counting Pairs in Memory

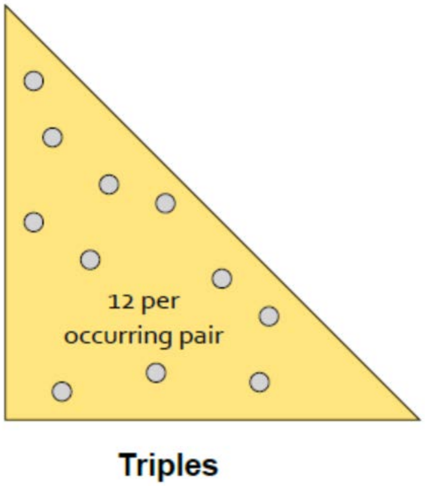


Triangular Matrix

- **Triangular Matrix Method**
  - Use a 1-dimensional triangular array T
  - Keep pair counts in lexicographical order:
    - {1,2}, {1, 3}, …{1, n}, {2, 3}, {2, 4}, …{2, n}, {3, 4}… {n-2, n-1}, {n-2, n}, {n-1, n}
  - Store count of pair *{i, j}* at entry T[k] with $1 \leq i < j \leq n$

where $$k = (i-1)\left(n - \frac{i}{2}\right) + j - i$$

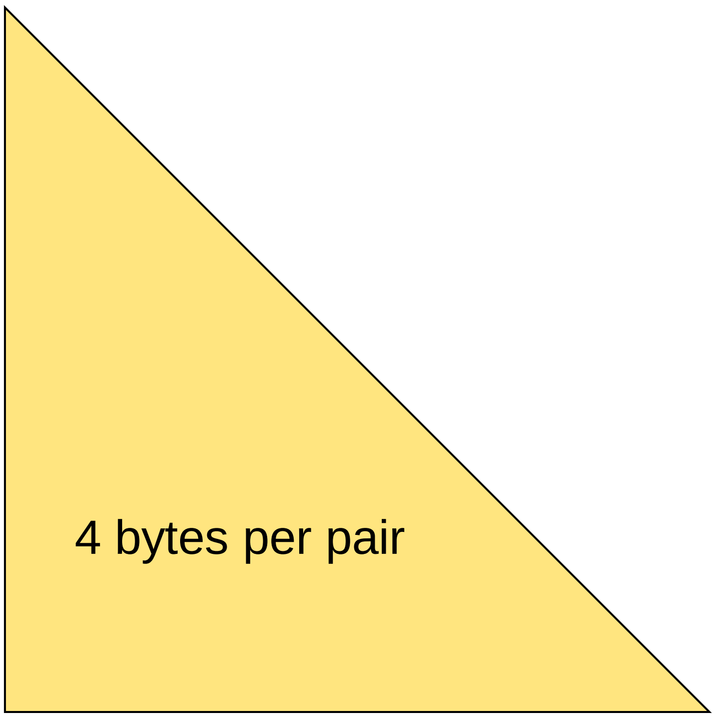
Counting Pairs in Memory



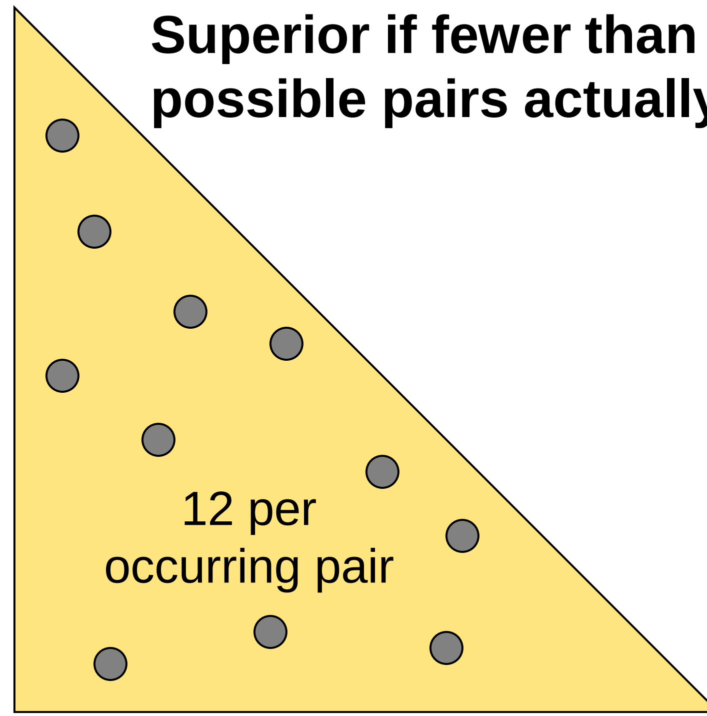
- **Triples Method using Hash Table**
  - Store counts as triples *[i, j, c]* where count of pair *{i, j}* with $i < j$ is c
  - Use hash table with i and j as search key
  - Only keep pairs with **count > 0**
  - Assume ids of items are also 4 bytes, we need 12 bytes for each pair and some overhead for hashtable

Counting Pairs in Memory

Superior if fewer than 1/3 of possible pairs actually occur



Triangular Matrix



Triples

What if we have too many items so the pairs cannot fit in main memory?

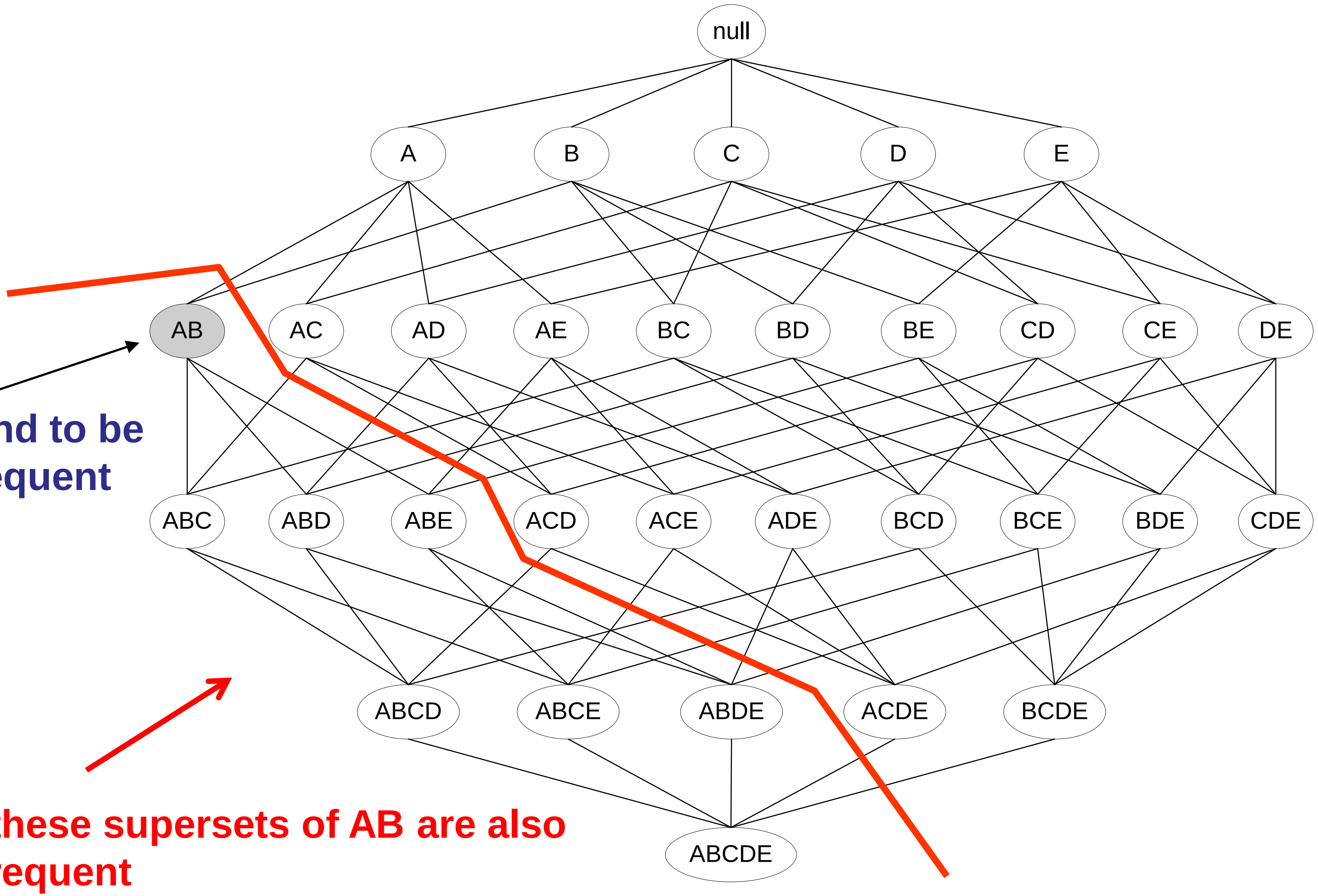
Apriori Algorithm

- **Two-pass approach reduce the need for main memory**
- **Key idea: Apriori Principle**
  - If an itemset is frequent, then all its subsets must be frequent
- **Apriori principle holds due to the anti-monotone property of support**

$$\forall X, Y : (X \subseteq Y) \Rightarrow \text{support}(X) \geq \text{support}(Y)$$

  - If item *i* does not appear in *s* baskets, then no superset containing *i* can appear in *s* baskets

Apriori Principle



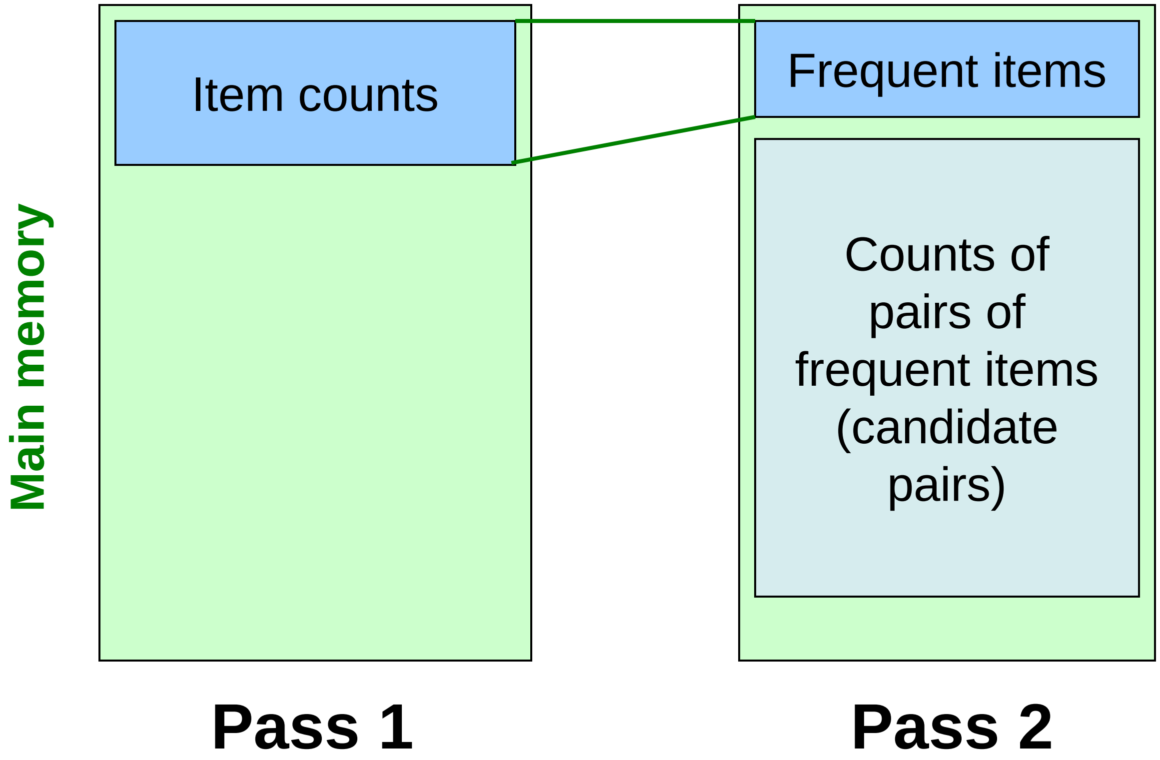
Found to be Infrequent

All these supersets of AB are also infrequent

Apriori Algorithm

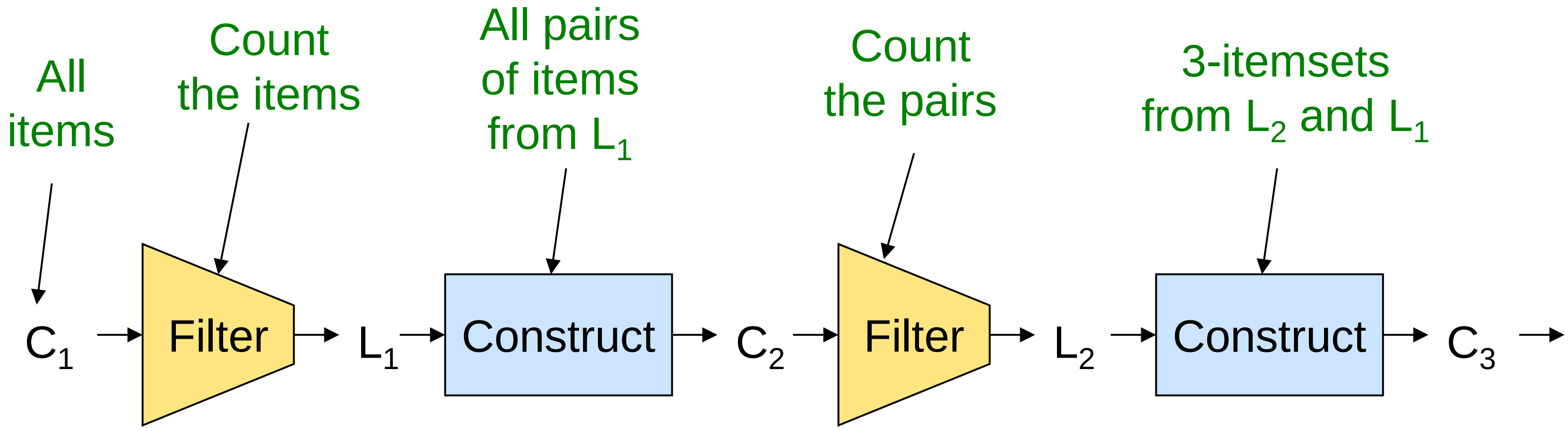
- **Pass 1: Read baskets and count in main memory the occurrences of each individual item**
  - Requires memory proportional to number of items
  - Items that appear **≥ *s*** (minsup) times are **frequent items**
- **Pass 2: Read baskets again and count in main memory <u>only</u> those pairs where both items are frequent (from Pass 1)**
  - Requires memory proportional to square of frequent items only (for counts)
  - Plus a list of the frequent items (so we know what must be counted)

Apriori Algorithm – Main Memory



Frequent k-Itemsets (k > 2)

- **For each *k*, we have**
  - ***C_k*** = ***candidate k-itemsets*** = those that might be frequent (support $\geq$ **s**) based on ***L_1*** and ***L_{k-1}***
  - ***L_k*** = set of truly frequent ***k-itemsets***



Example

- Suppose minsup = 3
- C_1 = { {b} {c} {j} {m} {p} }
  - Count support of itemsets in C_1
  - Prune non-frequent: L_1 = {{b}, {c}, {j}, {m}}
- C_2 = { {b,c} {b,j} {b,m} {c,j} {c,m} {j,m} }
  - Count support of itemsets in C_2
  - Prune non-frequent: L_2 = { {b,c} {b,m} {c,j} {c,m} }
- C_3 = { {b,c,j} {b,c,m} {b,m,j} {c,j,m} }
  - Count support of itemsets in C_3
  - Prune non-frequent: L_3 = { {b,c,m} }

B1	**m, c, b**
B2	**m, p, j**
B3	**m, b**
B4	**c, j**
B5	**m, p, b**
B6	**m, c, b, j**
B7	**c, b, j**
B8	**b, c, m**

Note: Generate new candidates C_k from L_{k-1} and L_1.
Can be more careful with candidate generation, e.g., in C_3 we know {b,m,j} cannot be frequent since {m,j} is not frequent.

Example

minsup = 2

Database D

TID	Items
100	1 3 4
200	2 3 5
300	1 2 3 5
400	2 5

Scan D

C_1

itemset	sup.
{1}	2
{2}	3
{3}	3
{4}	1
{5}	3

L_1

itemset	sup.
{1}	2
{2}	3
{3}	3
{5}	3

C_2

itemset
{1 2}
{1 3}
{1 5}
{2 3}
{2 5}
{3 5}

Scan D

C_2

itemset	sup
{1 2}	1
{1 3}	2
{1 5}	1
{2 3}	2
{2 5}	3
{3 5}	2

L_2

itemset	sup
{1 3}	2
{2 3}	2
{2 5}	3
{3 5}	2

C_3

itemset
{2 3 5}

Scan D

L_3

itemset	sup
{2 3 5}	2

Apriori Algorithm

- **Let k=1**
- **Generate frequent itemsets of length 1**
- **Repeat until no new frequent itemsets are identified**
  - Generate length (k+1) candidate itemsets from length k frequent itemsets
  - Prune candidate itemsets containing subsets of length k that are infrequent
    - **for size k+1 to be frequent, then all subsets of size k must be frequent**
    - **e.g., consider extending {1,3} by {5}. Since {1,5} is not frequent, {1,3,5} cannot be frequent**
    - **e.g., {2,3,5} is potentially frequent since all its subsets are frequent**
  - Count the support of each candidate by scanning the baskets
  - Eliminate candidates that are infrequent, leaving only those that are frequent

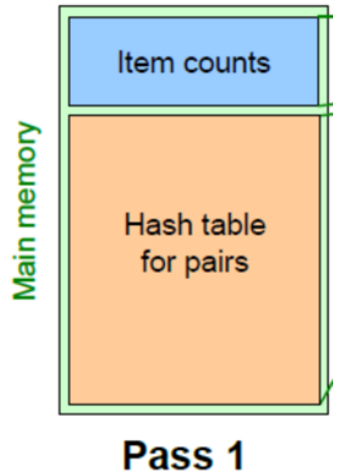
One pass for each *k* (itemset size)

PCY (Park-Chen-Yu) Algorithm

- **Motivation**
  - Pass 1 of Apriori
    - Only individual item counts are stored
    - Remaining memory is not used
  - Pass 2 of Apriori
    - Possible that *(i, j)* is not frequent even though *i* and *j* are frequent
    - But we still must count them (and need to store them in memory)
- **Can we use the idle memory to reduce memory required in pass 2?**

PCY Algorithm – First Pass

- In addition to item counts, maintain a hash table with as many buckets as fit in memory
- Keep a count (do not need the pairs) for each bucket into which pairs of items are hashed
- Number of buckets can be smaller than the number of pairs (collison is possible)

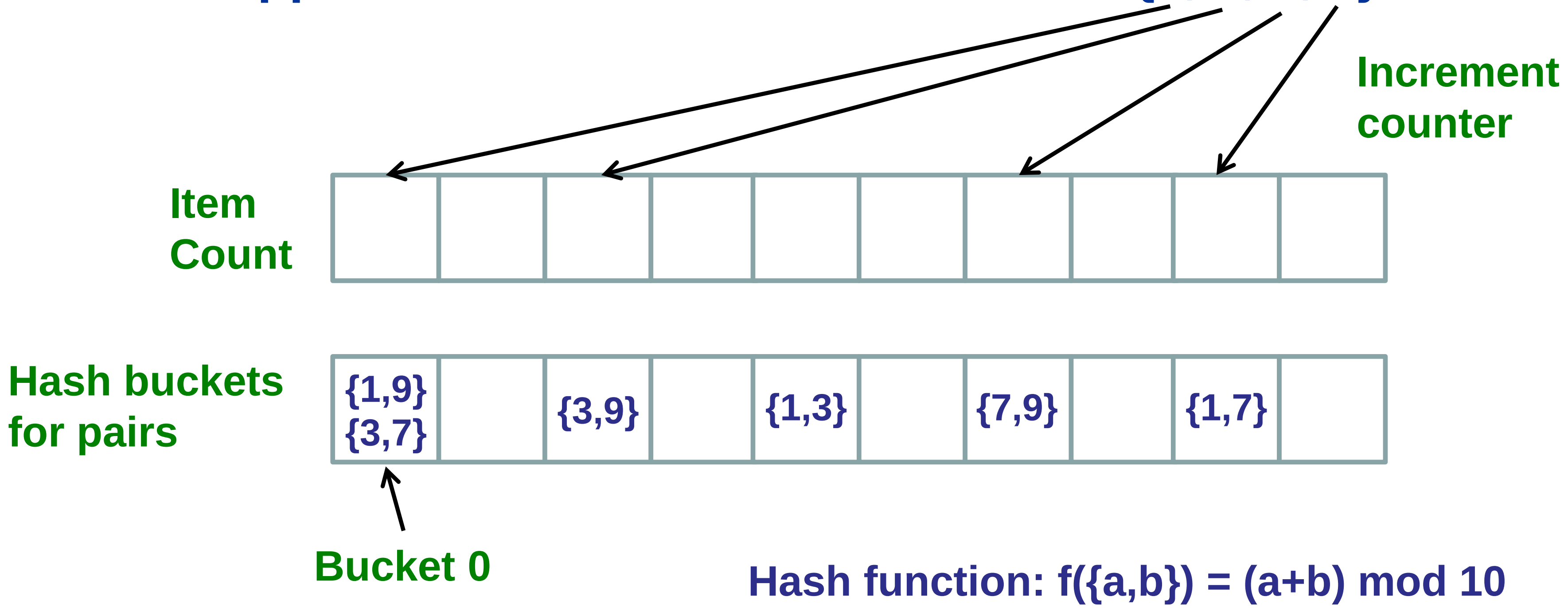



```
FOR (each basket) :
   FOR (each item in the basket) :
        add 1 to item's count;
   FOR (each pair of items) :
        hash the pair to a bucket;
        add 1 to the count for that bucket;
```

New in PCY

Example

- Suppose we have 10 items 1 to 10, then we have 45 pairs
- Suppose memory is only enough to hold 10 item counts, and 10 buckets
- Suppose we have a basket with items {1, 3, 7, 9}



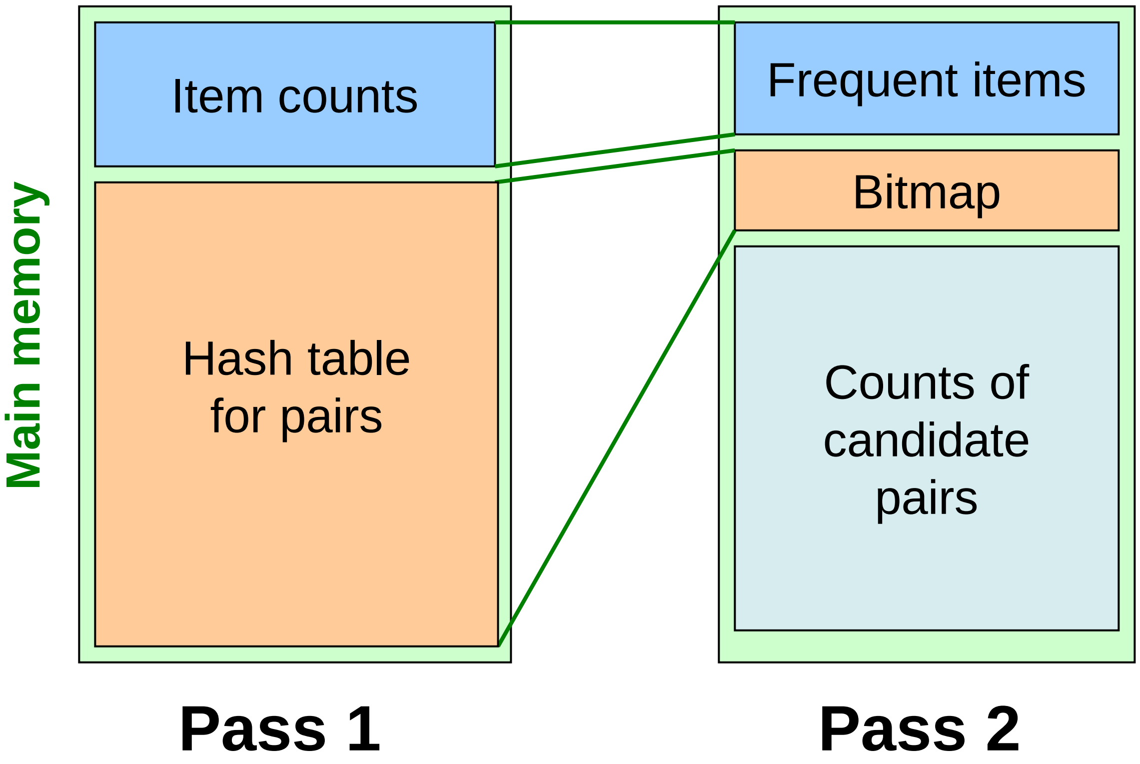
Observations about Buckets

- **If a bucket contains a frequent pair, then the bucket is surely frequent**
- **Note: A bucket can still be frequent even without any frequent pair**
  - There may be collisions, so more than one pair may be hashed to the same bucket, and *total count* > *s*
- **If a bucket has *total count* < *s*, none of its pairs can be frequent**
  - Pairs that hash to this bucket eliminated as candidates (even if the pair consists of 2 frequent items)
- **Pass 2: Only need to count pairs that hash to frequent buckets**

PCY Algorithm – Pass 2

- **Replace the buckets by a bit vector**
  - *1* means bucket count exceeds support *s* (frequent bucket); *0* means it did not
  - Hash value now corresponds to the bit position
  - 4 byte integer counts replaced by bits, so the bit vector requires 1/32 of memory
- **Count all pairs *{i, j}* that meet the conditions for being a candidate pair**
  - Both ***i*** and ***j*** are frequent items
  - Pair ***{i, j}*** hash to a bucket whose bit in the bit vector is **1** (frequent bucket)
- **Both conditions are necessary for the pair to have a chance of being frequent**

PCY Algorithm – Main Memory

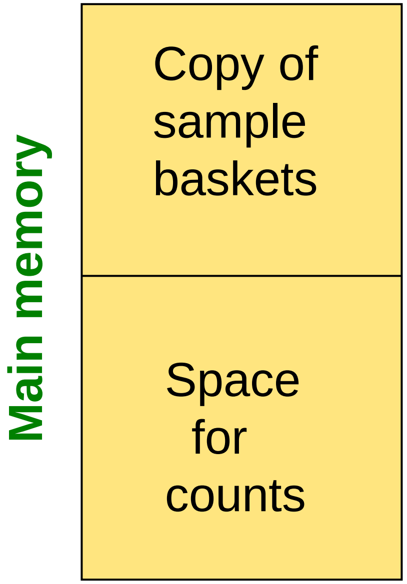


Find Frequent Itemsets in $\leq$ 2 Passes

- **Apriori and PCY take *k* passes to find frequent itemsets of size *k*. Can we use fewer passes?**
- **Random sampling**
  - May miss some frequent itemsets
  - Useful for application where it is not essential to discover every frequent itemset
    - e.g. supermarket, does not run a sale based on every itemset we find
- **SON (Savasere, Omiecinski, and Navathe)**
  - Exact answer in two full passes
  - Implemented by MapReduce

Random Sampling

- **Take a random sample of the market baskets**
- **Load sample into main memory**
- **Run a frequent itemset mining algorithm (e.g., Apriori) in main memory**
- **Reduce support threshold proportionally to match sample size**
  - e.g. if sample 1% of baskets, then we should look for itemsets that appear in at least *s/100* of the baskets



Random Sampling

- **False positives**
  - **Itemset may be frequent in the sample but not in the entire set of baskets** (due to reduced threshold)
  - Run a second pass through all the baskets to verify that the candidate itemsets are truly frequent
    - Can remove false positives totally
- **False negatives**
  - **Itemset is frequent in the original set of baskets but not picked out from the sample**
  - Scanning a second time does not help
    - Using smaller threshold helps catch more truly frequent itemsets, but requires more space

SON Algorithm

- **Repeatedly read small subsets of the baskets into main memory and run an in-memory algorithm to find all frequent itemsets**
  - We are not sampling, but processing the baskets in memory-sized chunks
  - Use *ps* as threshold if each subset is fraction *p* of all the baskets and *s* is the support threshold
  - Store on disk the frequent itemsets found for each chunk
- **An itemset becomes a candidate if it is frequent in *any* one or more subsets of the baskets**
- **Second pass counts all the candidate itemsets and determine which are frequent in the entire set of baskets**

SON Algorithm

- **Key idea: Monotonicity**
  - If an itemset is **not frequent** in any chunk, then it is not frequent in the entire set of baskets
    - Support for itemset ≤ ps in each chunk.
    - Number of chunks is $1/p$
    - Total support for itemset ≤ $(1/p)ps = s$
  - Every itemset that is frequent in the entire set of baskets is frequent in at least one chunk
    - No false negatives
    - We can be sure that all the truly frequent itemsets are among the candidates

SON – Distributed Version

- **SON lends itself to distributed data mining**
  - Each chunk can be processed in parallel
  - Frequent itemsets from each chunk combined to form candidates
- **Distribute candidates to all the nodes**
  - Each node count support for each candidate in a subset of basket
  - Accumulate the counts of all candidates

SON and MapReduce

- **Phase 1: Find candidate itemsets**
  - Map
    - Take assigned subset of baskets and find the frequent itemsets.
    - Lower support threshold from *s* to *ps* if each Map task gets fraction p of the total number of baskets.
    - Output is a set of key-value pairs (F, 1) where F is a frequent itemset, value is always 1 (irrelevant).
  - Reduce
    - Each Reduce task is assigned a set of keys (itemsets)
    - Output keys that appear one or more times (candidate itemsets)

SON and MapReduce

- **Phase 2: Find true frequent itemsets**
  - Map
    - Take output from Reduce function in Phase 1 and a portion of the baskets.
    - Map task counts occurrences of candidate itemsets among the baskets.
    - Output is a set of key-value pairs *(C, v)* where *C* is a candidate itemset, and *v* is its support.
  - Reduce
    - Reduce task take assigned itemsets as keys and sum the associated values (total support)
    - Output itemsets whose sum ≥ s (frequent in dataset)

Summary

- Finding frequent itemsets is expensive
- Finding frequent pairs is hard
  - Memory constraint
- Apriori and PCY Algorithms
  - k passes to find frequent k-itemsets
- Random sampling and SON Algorithms