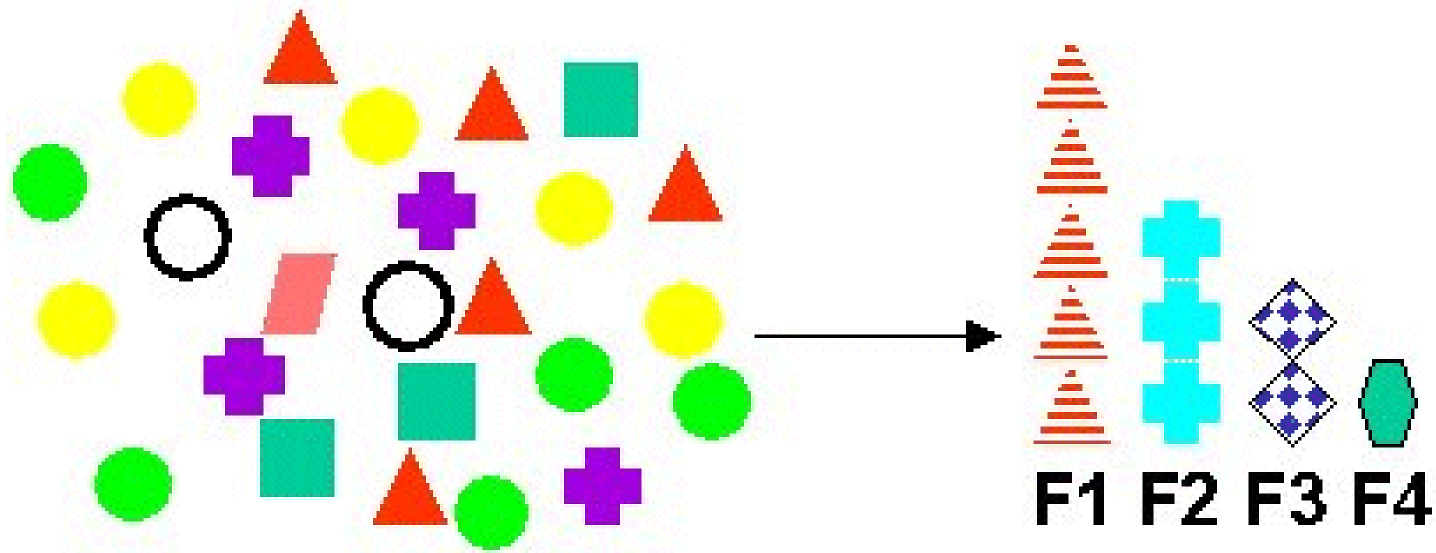
Frequent Item Sets & Association Rules



http://www.csd.uoc.gr/~hy562

University of Crete

Some History

- Barcode technology allowed retailers to collect massive volumes of sales data
  - Basket data: transaction date, set of items bought
  - Data is stored in tertiary storage

- Leverage information for marketing
  - How to design coupons?
  - How to organize shelves?
- The birth of data mining!
  - Agrawal et al. (SIGMOD 1993) introduced the problem of mining a large collection of basket data to discover association rules
  - Many papers followed…

Example: Supermarket Shelf Management

- Goal: Process the sales data to find dependencies among items
  - Given a set of transactions, predict the occurrence of an item based on the occurrences of other items in the transactions *(association rules)*
- Approach: Identify items that are bought together by sufficiently many customers *(frequent itemsets)*
- The famous "diapers-and-beer" example:
  - If one buys diapers, then he is likely to buy beer
  - Don't be surprised if you find six-packs next to diapers!

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

Rules Discovered:

{Milk} --> {Coke}

{Diaper, Milk} --> {Beer}

Example Dataset

- Download it
- https://drive.google.com/file/d/105NsvmCj_yQqkcQNDWaMD_v1et38VwTd/view?usp=drive_link
- Check if you can identify association rules?

The Market-Basket Model

- A large set of items, e.g., things sold in a store
  - $I = \{i_1, i_2, \dots, i_m\}$

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

- A large set of baskets/transactions, e.g., things one customer buys in one visit to the store
  - B_i a set of items, and $B_i \subseteq I$
- Transaction Database T: a set of transactions $B = \{B_1, B_2, \dots, B_n\}$
- Our interest: Identify *associations among "items"*, not "baskets"

Application Examples of Association Rules

- Items = products; Baskets = sets of products someone bought in one transaction
  - Reveals typical buying behaviour of customers
    - Marketing and sales promotion (suggests tie-in "tricks")
      - product p appearing as rule's consequent
        - *"what should be done to boost p sales?"*
      - product p' appearing as rule's antecedent
        - *"which products would be affected if we stop selling p'?"*
    - Shelf management: position certain items strategically
    - Recommendations
      - Amazon customers who bought *X* also bought *Y*
      - Product Bundling (e.g., phone + case + car holder + charger)

- High support needed, or no €€'s
  - Only useful if many customers buy diapers and beer

Market-Baskets and Associations

- A many-many mapping (association) between two kinds of things
  - E.g., 90% of transactions that purchase diaper & milk also purchase beer
- Given a set of baskets, discover association rules
  - The technology focuses on common events, not rare events (“long tail”)
- 2-step approach
  - Find frequent *itemsets*
  - Generate *association rules*

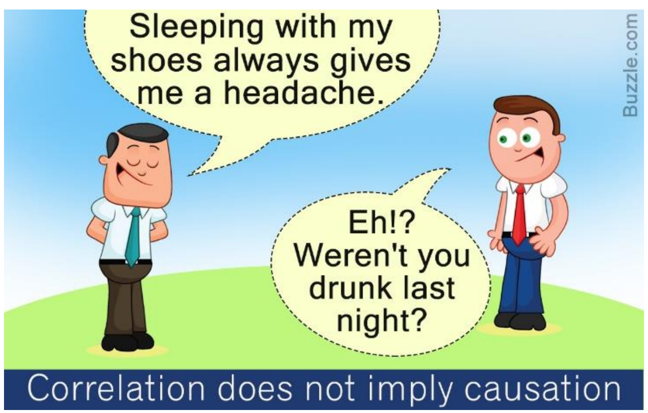
Rules Discovered:

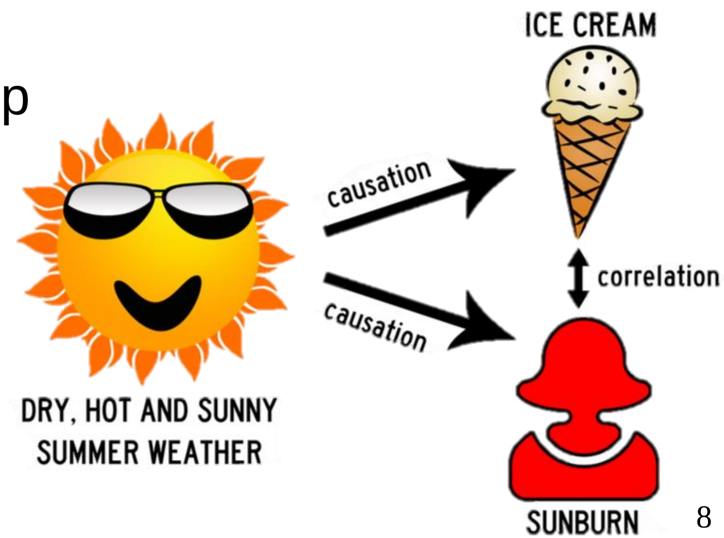
{Milk} --> {Coke}

{Diaper, Milk} --> {Beer}

Causation vs. Association

- In machine learning, $X \rightarrow Y$ usually implies a causal relationship
  - "a change in X (seen as cause) forces a change in Y *(seen as effect)*"
  - *causation* is complex and difficult to prove
- In rule mining, $X \rightarrow Y$ is an association relationship
  - "X is associated with Y "
  - Much easier to calculate and prove
    - of less interest for medical research than for market research
- Association rules indicate only the *existence* of a statistical relationship (correlation) between X and Y
  - They do not specify the *nature* of the relationship





Frequent Itemsets

- Find sets of items, called itemsets, that appear "frequently" in the baskets
  - k-itemset: a set of k items
  - B_1 = {b, c, m} is a 3-itemset

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

- A transaction B_i contains an itemset $A = \{i_1, i_2, \ldots, i_k\}$, if $A \subseteq B_i$
  - B_3= {b, c, d, m} contains the 3-itemset {b, c, m}

- *Support* of itemset *A*: the number (or fraction) of baskets containing all items in *A*
  - Support of {Milk} = 4
  - Support of {Milk, Diaper, Beer} = 2

- *Frequent itemsets*: sets of items that appear in at least *s* baskets
  - *s* is a given support threshold

Example: Frequent Itemsets

- Items = {b, c, d, j, m}
- Support threshold *s* = 3 baskets

B_1 = {m, c, b} B_2 = {m, d, j}

B_3 = {m, b} B_4 = {c, j}

B_5 = {m, d, b} B_6 = {m, c, b, j}

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

- Frequent itemsets: {b}, {c}, {j}, {m},

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

Association Rules

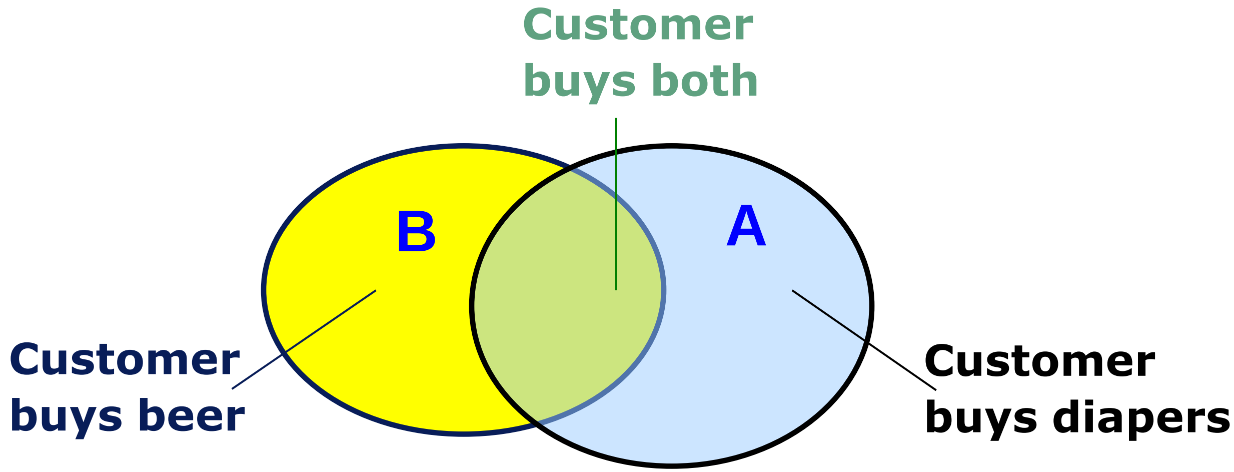
- An association rule is an implication of the form:
  $\{i_1, i_2, \ldots, i_k\} \rightarrow \{j_1, j_2, \ldots, j_l\}$, where

  $\{i_1, i_2, \ldots, i_k\}, \{j_1, j_2, \ldots, j_l\}, \subset I$, *and*
  $\{i_1, i_2, \ldots, i_k\} \cap \{j_1, j_2, \ldots, j_l\} = \varnothing$

- If-then rules about the contents of baskets
  - $\{i_1, i_2, \ldots, i_k\} \rightarrow j$ means:
    "if a basket contains all of $i_1, \ldots, i_k$ then it is *likely* to contain j"

- A general form of an association rule is `Body→Head[Support,Confidence]`
  - *Antecedent*, left-hand side (LHS), body
  - *Consequent*, right-hand side (RHS), head
  - *Support*, frequency
  - *Confidence*, strength

Support and Confidence



- *Support* of the rule A → B: the frequency of the rule within all transactions in the database *T*, i.e., the probability that a transaction contains the union of A and B
  - support(A → B) = p(A B) = support({A,B})

- *Confidence* of the rule A → B: denotes the percentage of transactions that contain B, among those that contain A, i.e., the conditional probability that a transaction containing A also contains B
  - confidence(A → B) = p(B|A) = p(A B) / p(A)
    = support({A,B}) / support({A})

Example: Confidence

B_1 = {m, c, b} B_2 = {m, d, j}
B_3 = {m, b} B_4 = {c, j}
B_5 = {m, d, b} B_6 = {m, c, b, j}
B_7 = {c, b, j} B_8 = {b, c}

- An association rule: {m, b} → {c}
  - Support ({m, b}) = 4, Support ({m, b, c}) = 2
  - Confidence ({m, b} → c) = 2/4 = 50%

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

Interesting Association Rules

- Not all high-confidence rules are interesting
  - The rule $\{i_1, i_2, \dots, i_k\} \rightarrow$ milk may have high confidence for many itemsets $\{i_1, i_2, \dots, i_k\}$, because milk is purchased very often (independent of the itemset) and the confidence will be very high

- Lift (originally called interest) of an association rule A→B is the difference between its confidence and the fraction of baskets that contain B

  Lift (A→B) = | conf(A→B) – Pr[B] |

  - Interesting rules are those with high positive or negative lift values thus we take the absolute value
  - For uninteresting rules, the fraction of baskets containing itemset B will be the same as the fraction of the subset baskets including $A \cup B$
    - So confidence may be high, but interest low

Example: Confidence and Lift

B_1 = {m, c, b}
B_2 = {m, d, j}
B_3 = {m, b}
B_4 = {c, j}
B_5 = {m, d, b}
B_6 = {m, c, b, j}
B_7 = {c, b, j}
B_8 = {b, c}

- An association rule: {m, b} → c
  - Confidence ({m, b} → c) = 2/4 = 50%
  - Lift ({m, b} → c) = |0.5 – 5/8| = 1/8
    - Item c appears in 5/8 of the baskets
  - Rule is not very interesting!

Lift (A→B) = |conf(A→B) – Pr[B]|

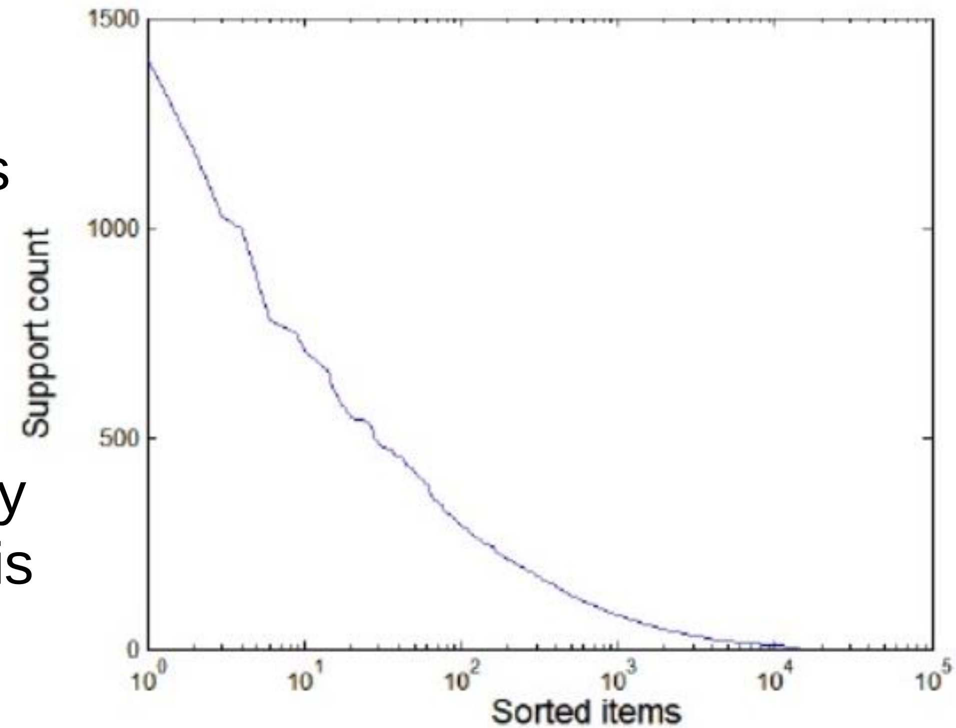
Finding Association Rules

- Goal: Find all rules that satisfy the user-specified *minimum support* (**minsup**) and *minimum confidence* (**minconf**)
  - *support* >= *s* AND *confidence* >= *c*
- Key Features
  - Completeness: find all rules
  - Mining with data on disk (not in memory)
- Hard part: Finding the frequent itemsets
  - If A → B has high support and confidence, then both A and B will be frequent

How to Set the Appropriate MinSup?

- Many real data sets have skewed support distribution

- If minsup is too high, we could miss itemsets involving interesting rare items (e.g., expensive products)



- If minsup is too low, it is computationally expensive and the number of itemsets is very large

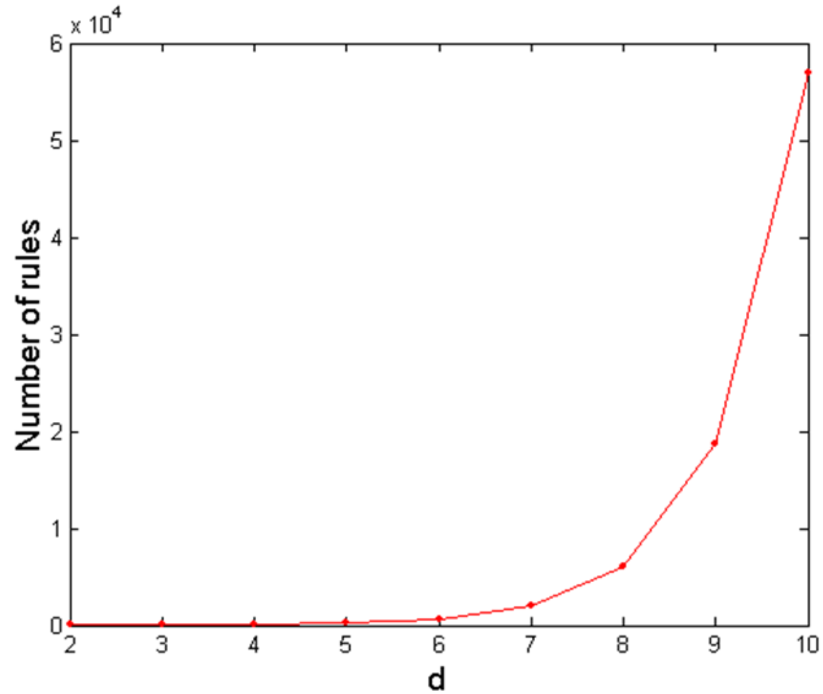
- A single minsup threshold may not be always effective

Association Rule Mining Task

Brute-force approach:

- List all possible association rules
  - Given d unique items:
    - Total number of itemsets = 2^d
    - Total number of ARs = R

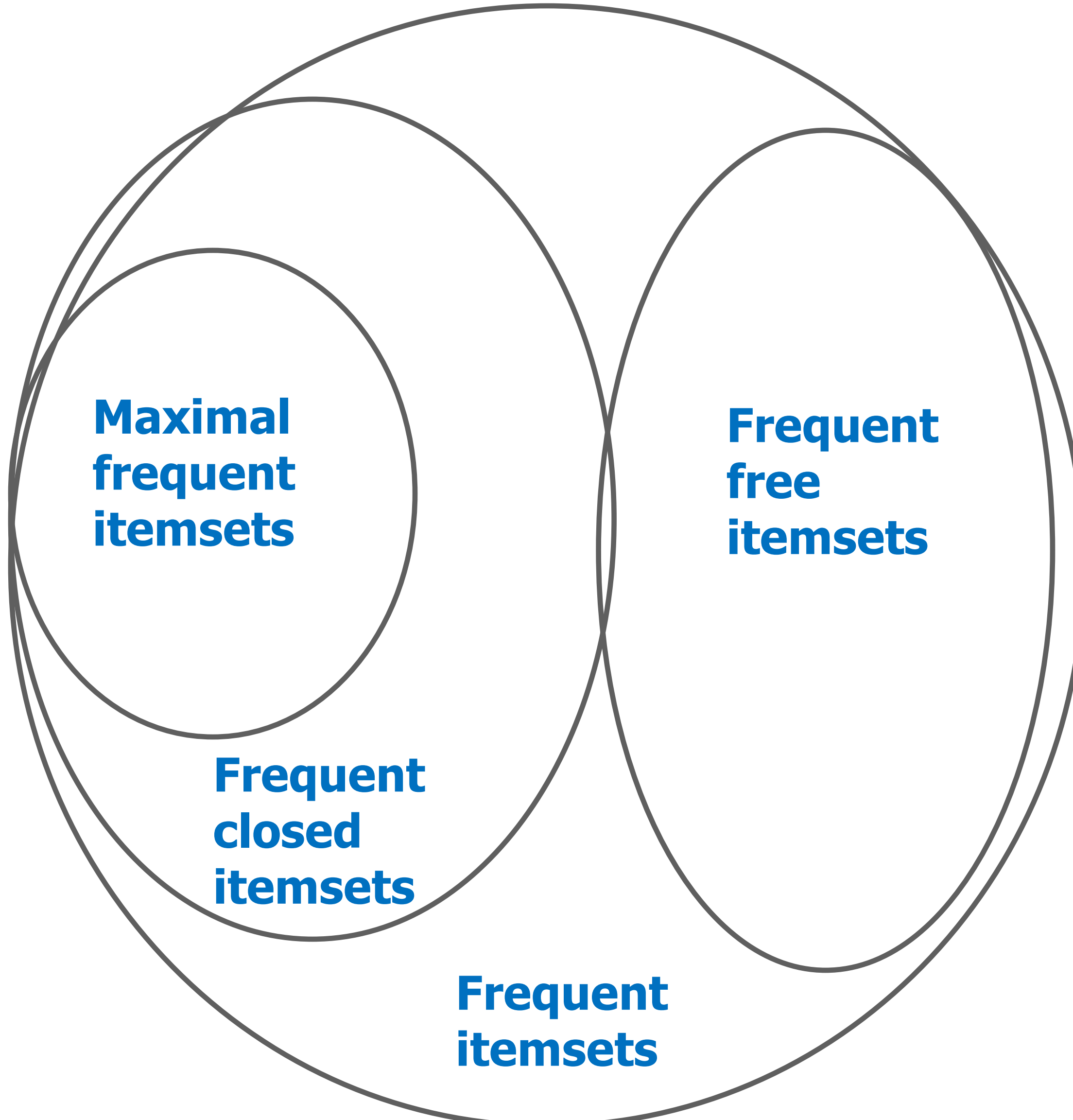
$$R = \sum_{k=1}^{d-1}\left[\binom{d}{k} \times \sum_{j=1}^{d-k}\binom{d-k}{j}\right] = 3^d - 2^{d+1} + 1$$



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

Compacting Output Rules: Classes of Itemsets

- To reduce the number of rules we can post-process and only output:
  - Maximal Frequent itemsets: no *immediate superset is frequent*
    - Can generate all frequent itemsets (without support)
  - Closed itemsets: no *immediate superset has the same count* (>0)
    - Can generate all frequent itemsets and their support
- Alternately:
  - Free itemset: no *immediate subset has the same count* (>0)



Example: Maximal/Closed

	Count	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

Apriori Algorithm

Reducing the Number of Candidates: The Apriori algorithm

- Rules from the same itemset have equal support but can have different confidence
  - Thus, we may decouple the support and confidence

- Two steps:
  - ❶*Frequent Itemsets:* Find all itemsets that have minimum support
    - Key idea: anti-monotonicity of support: $\forall A,B \ A \subseteq B \Rightarrow s(A) \geq s(B)$
  - ❷*Rule generation:* Use frequent itemsets to generate rules
    - For every subset `A` of a frequent itemset `I`, generate rule $A \rightarrow I \setminus A$
    - Variant 1: Perform a single pass to compute the rule confidence
      - `conf(A,B→C,D) = supp(A,B,C,D)/supp(A,B)`
    - Variant 2: Filter out bigger rules from smaller ones
      - If `A,B,C→D` is below confidence, so is `A,B→C,D`
    - Confidence of rules generated from the same itemset has an anti-monotone property
      - e.g., I = {A,B,C,D}: conf(ABC→D)≥conf(AB→CD)≥conf(A→BCD)
      - Confidence is anti-monotone w.r.t. number of items on the RHS of the rule

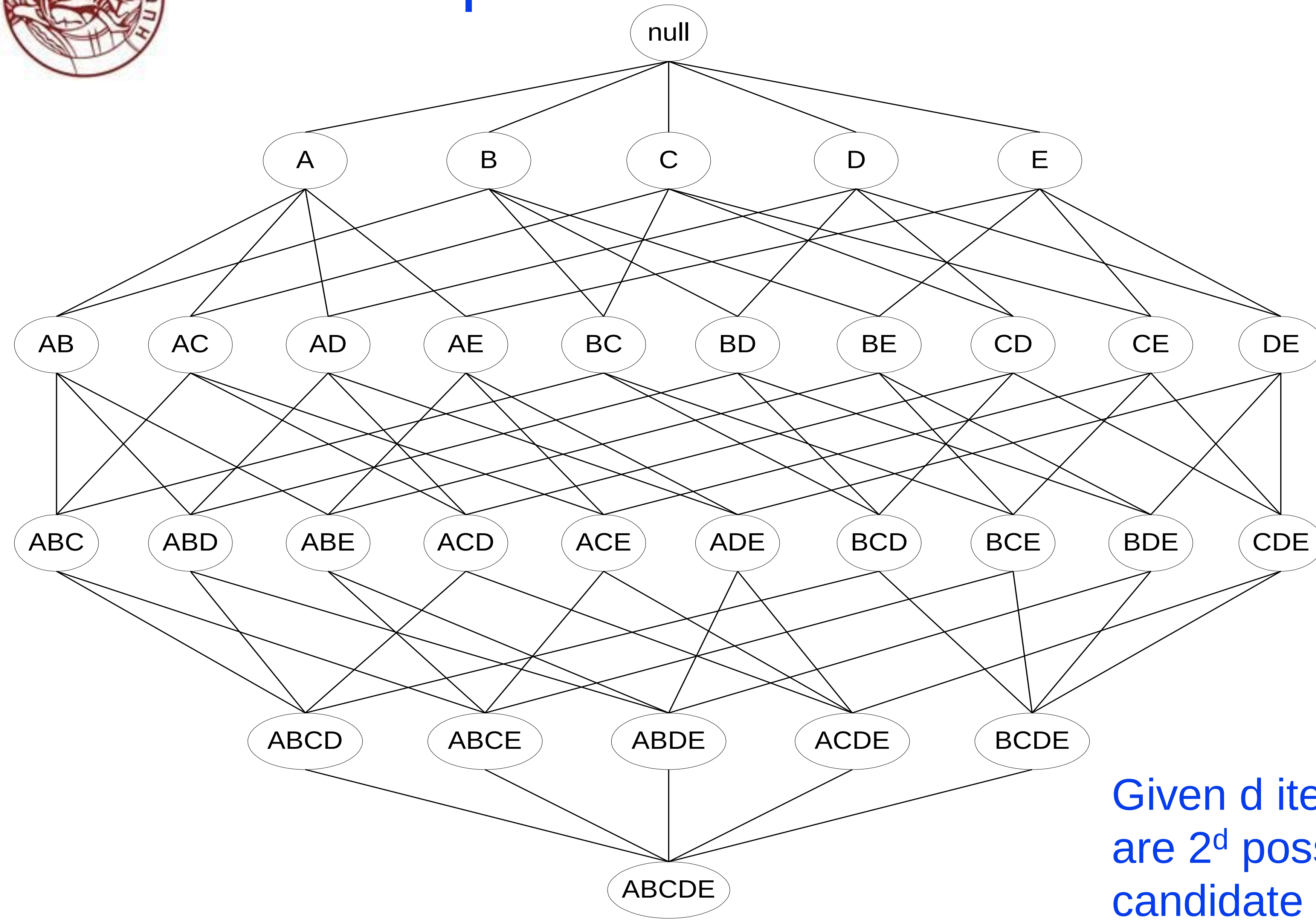
Example

B_1 = {m, c, b} B_2 = {m, d, j}
B_3 = {m, c, b, n} B_4= {c, j}
B_5 = {m, d, b} B_6 = {m, c, b, j}
B_7 = {c, b, j} B_8 = {b, c}

- 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 … b,m→c: *c*=3/4
  - ~~b→c,m: *c*=3/6~~

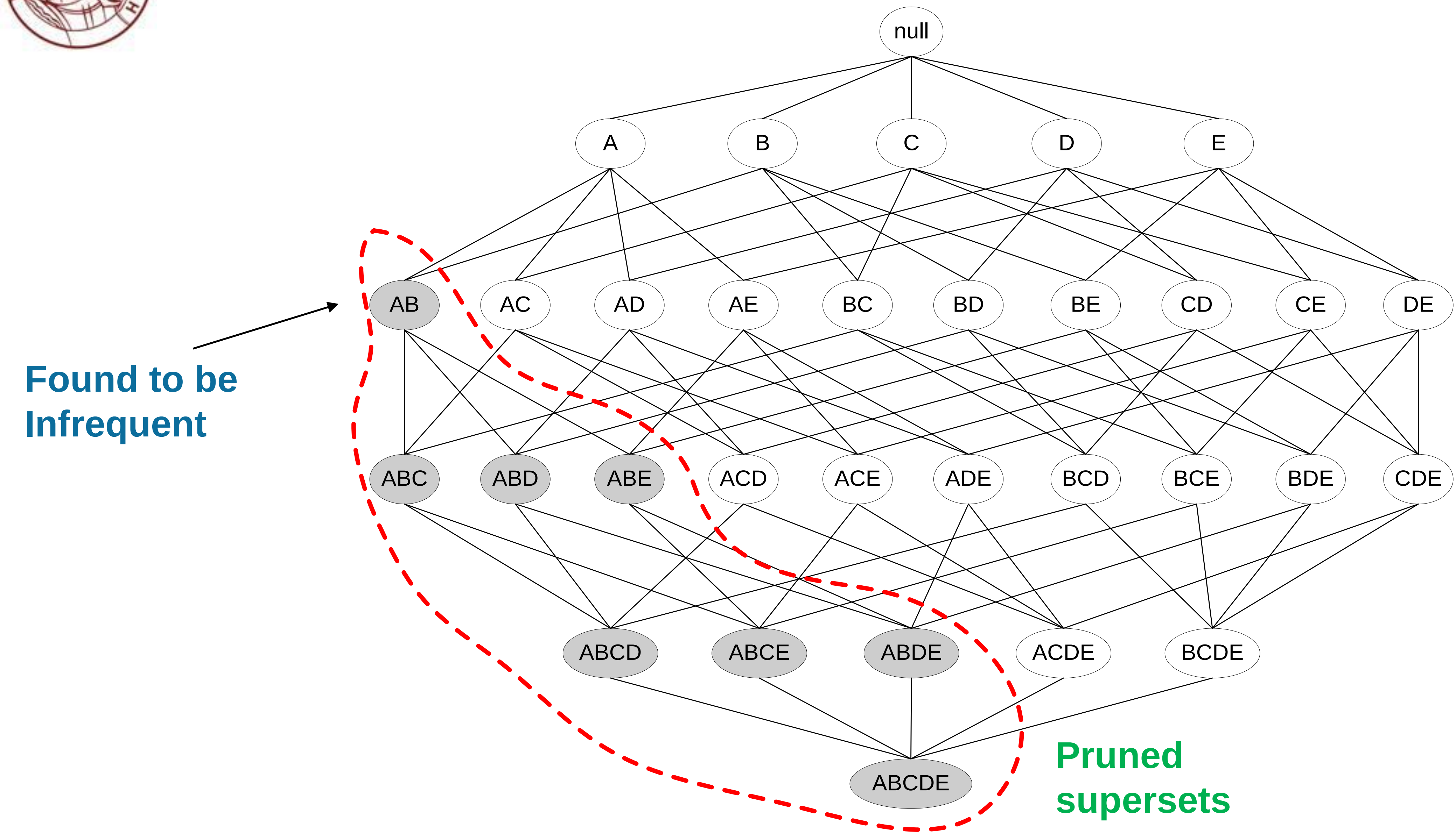
conf(A→B) = supp(A∪B)/supp(A)

Frequent Itemset Generation

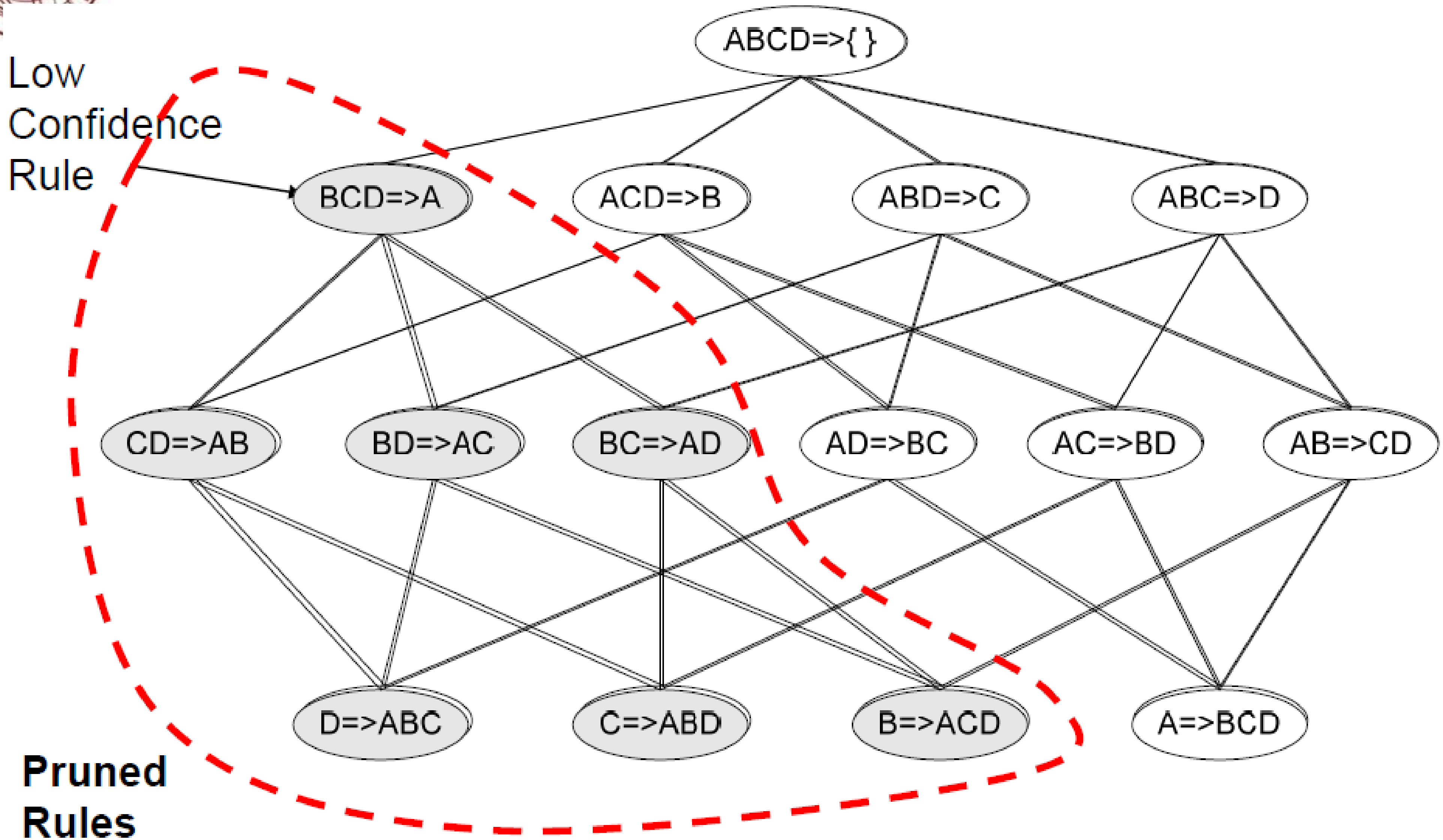


Given d items, there are 2^d possible candidate itemsets

Illustrating the Apriori Principle



Rule Generation Example



Example

Market-Basket transactions

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

Items (1-itemsets)

Item	Count
Bread	**4**
Coke	**2**
Milk	**4**
Beer	**3**
Diaper	**4**
Eggs	**1**

Minimum Support = 3

Pairs (2-itemsets)

Itemset	Count
{Bread,Milk}	**3**
{Bread,Beer}	**2**
{Bread,Diaper}	**3**
{Milk,Beer}	**2**
{Milk,Diaper}	**3**
{Beer,Diaper}	**3**

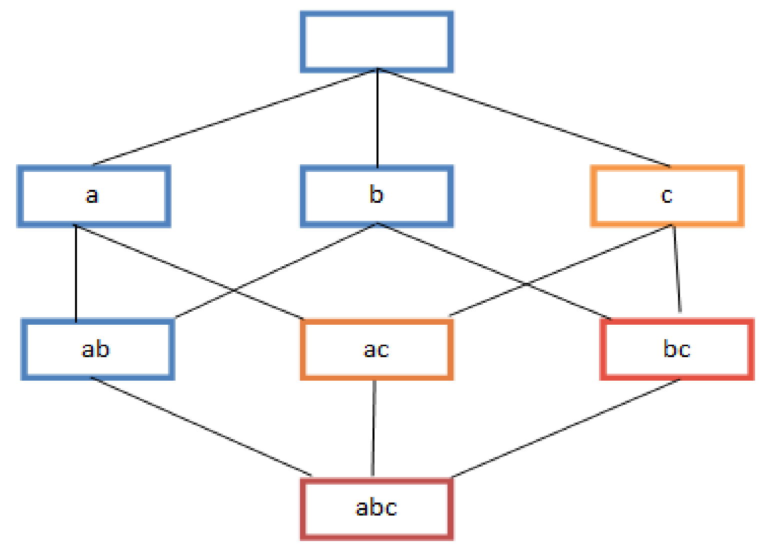
(no need to generate candidates involving Coke or Eggs)

Triplets (3-itemsets)

Itemset	Count
{Bread,Milk,Diaper}	**2**

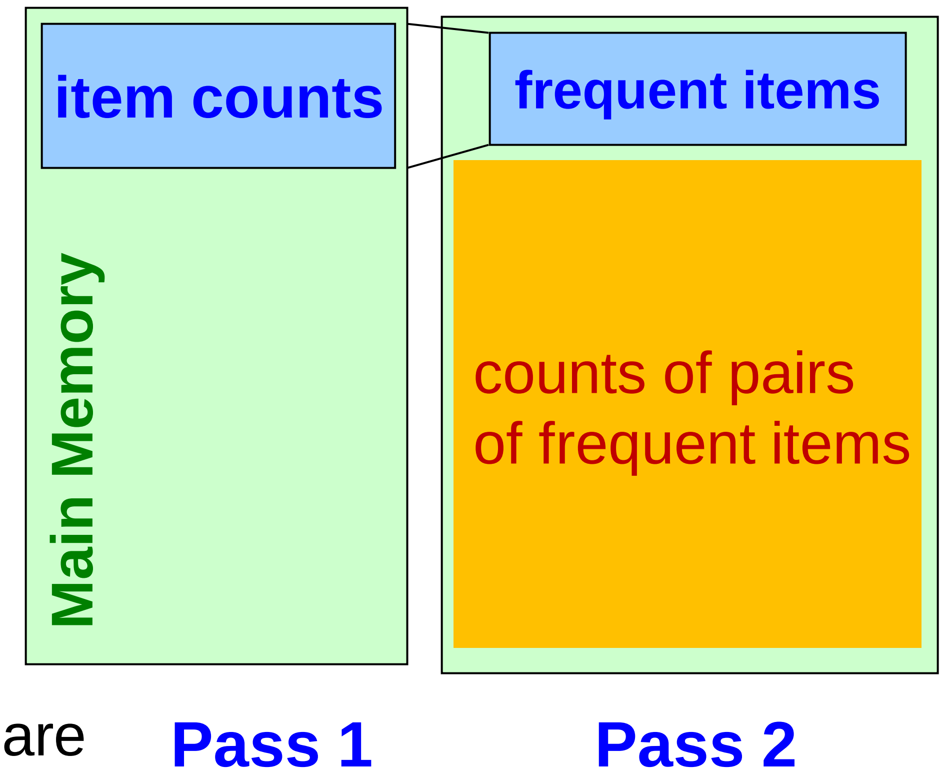
Candidate Generation

- Contrapositive for pairs: if item *i* does not appear in *s* baskets, then no pair including *i* can appear in *s* baskets
- Basic principle (Apriori):
  - An itemset of size k+1 is candidate to be frequent only if all of its subsets of size k are known to be frequent
- Main idea:
  - Construct a candidate of size k+1 by combining two frequent itemsets of size k
  - Prune the generated k+1-itemsets that do not have all k-subsets to be frequent
- So, how does Apriori find frequent pairs?
  - A two-pass approach limiting the need for main memory counts



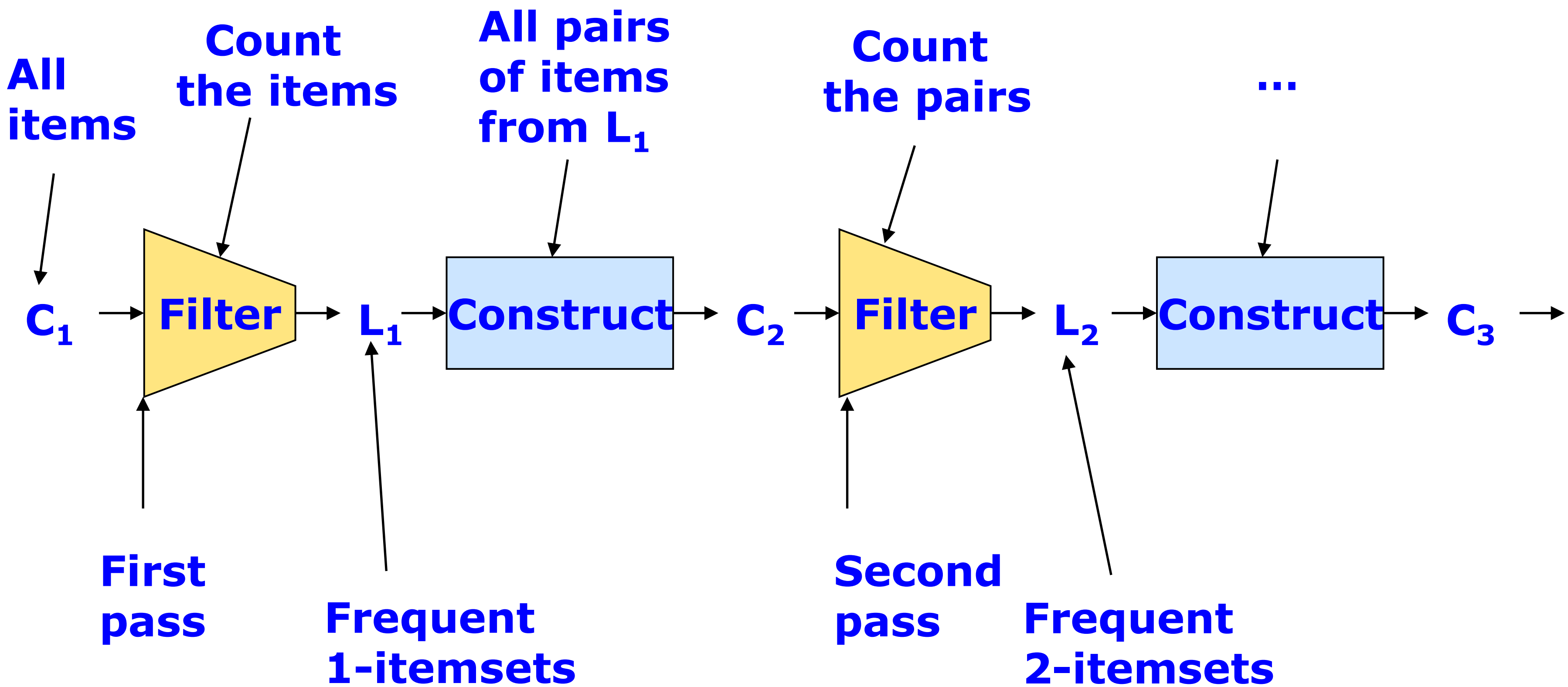
Apriori Algorithm

- Pass 1: Read baskets and count in main memory the occurrences of each item
  - Requires only memory proportional to #items
  - Items that appear at least *s* times (minsup) are the *frequent items*
- Pass 2: Read baskets again and count in main memory only those pairs where both elements were found in Pass 1 to be frequent
  - 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)



Frequent Triples, Etc.

- For each *k*, we construct two sets of *k–itemsets*:
  - C_k = *candidate k*–itemsets: supersets of (k-1)-itemsets with support ≥ *s*
  - L_k = the set of truly frequent *k-itemsets*



The Apriori algorithm

Level-wise approach

C_k = candidate k-itemsets
L_k = frequent k-itemsets

1. k = 1, C_1 = all items
2. While C_k not empty

Frequent itemset generation

3. Scan the database to find which itemsets in C_k are frequent and put them into L_k

Candidate generation

4. Use L_k to generate a collection of candidate (k+1)-itemsets C_{k+1}
5. k = k+1

Recall: Example from Last time

B_1 = {m, c, b} B_2 = {m, d, j}
B_3 = {m, c, b, n} B_4= {c, j}
B_5 = {m, d, b} B_6 = {m, c, b, j}
B_7 = {c, b, j} B_8 = {b, c}

- Frequent itemsets (s = 3):
  - {b}, {c}, {j}, {m}
  - {b,m} {b,c} {c,j} {c,m}
  - {b,c,m}

- How we can compute them with Apriori?

Apriori Execution Example

s = 3

Database TDB

Tid	Items
10	b, c, m
20	d, j, m
30	b, c, m, n
40	c, j
50	b, d, m
60	b, c, j, m
70	b, c, j
80	b, c

1st scan

C_1

Itemset	sup
{b}	6
{c}	6
{d}	2
{j}	4
{m}	5
{n}	1

L_1

Itemset	sup
{b}	6
{c}	6
{j}	4
{m}	5

C_2

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

2nd scan

C_2

Itemset	sup
{b, c}	5
{b, j}	2
{b, m}	4
{c, j}	3
{c, m}	3
{j, m}	2

L_2

Itemset	sup
{b, c}	5
{b, m}	4
{c, j}	3
{c, m}	3

C_3

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

3rd scan

C_3

Itemset	sup
{b, c, m}	3
{b, c, j}	2
{b, m, j}	1
{c, m, j}	1

L_3

Itemset	sup
{b, c, m}	3

How to Improve Apriori Efficiency?

- Dynamic itemset counting
  - Add new candidate itemsets only when *all* of the subsets are estimated to be frequent
- Transaction Reduction
  - A transaction that does not contain *any* frequent k-itemset is useless in subsequent scans
- Hash-based itemset counting
  - A k-itemset whose corresponding *hashing bucket count* is below the threshold cannot be frequent
- Partitioning
  - Any itemset that is potentially frequent in DB must be *frequent in at least one of the partitions* of the DB
- Sampling
  - Mining on a subset of given data, *lower support threshold* and consider a method to determine completeness

Improvements to Apriori

Observations

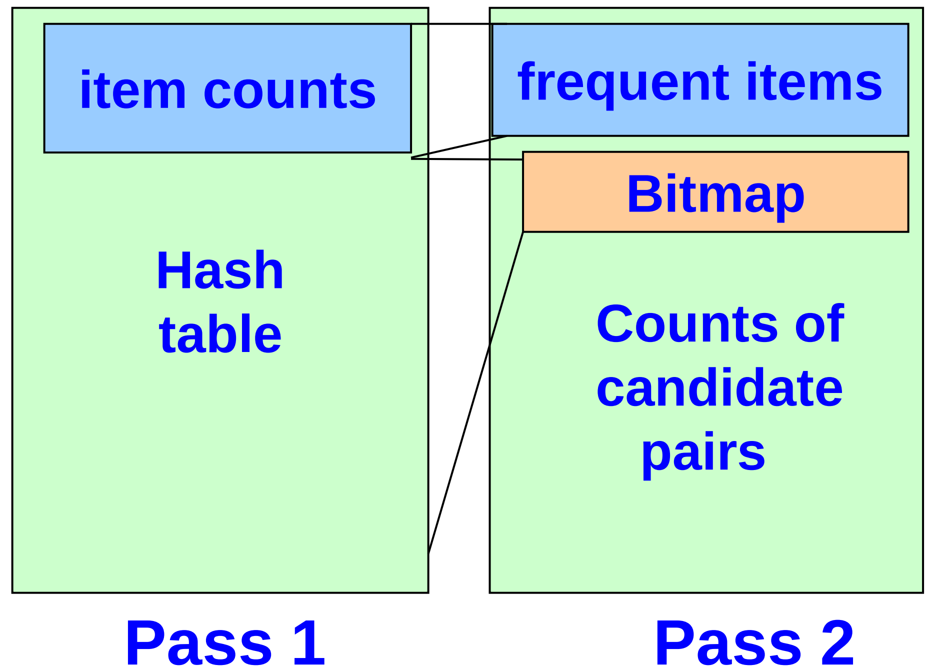
- In pass 1 of the Apriori algorithm
  - only individual item counts are stored
  - remaining memory is unused

- In pass 2, the pair `(i,j)` may not be frequent even if `i` and `j` are frequent
  - but we must still count its frequency (hence need to store it in memory)

- Can we use the idle memory (in pass 1) to reduce the memory required in pass 2?

PCY (Park-Chen-Yu) Algorithm

- Pass 1 of PCY: In addition to item counts, maintain a hash table with *as many buckets as can fit in memory*
  - Each pair of items is hashed to one bucket
    - Collisions are possible!
  - Every time a pair is met in a basket, increase the count of its bucket in the hash table by 1
- Pass 2 of PCY: we only count pairs that hash to frequent buckets



- *Multistage* improves PCY (later)

PCY Algorithm – Pass 1

New in PCY

```
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
    }
}
```

- Pairs of items need to be generated
- Before Pass 1 Organize Main Memory
  - Space to count each item: One (typically) 4-byte integer per item
  - Use the rest of the space for as many integers, representing buckets, as we can

Observations about Buckets

- We are not just interested in the presence of a pair
  - but also if its support is $\geq s$

- If a bucket contains a frequent pair, then the bucket is surely frequent

- A bucket can be frequent even without any frequent pair (*false positives*)
  →We cannot eliminate any member (pair) of a "frequent" bucket

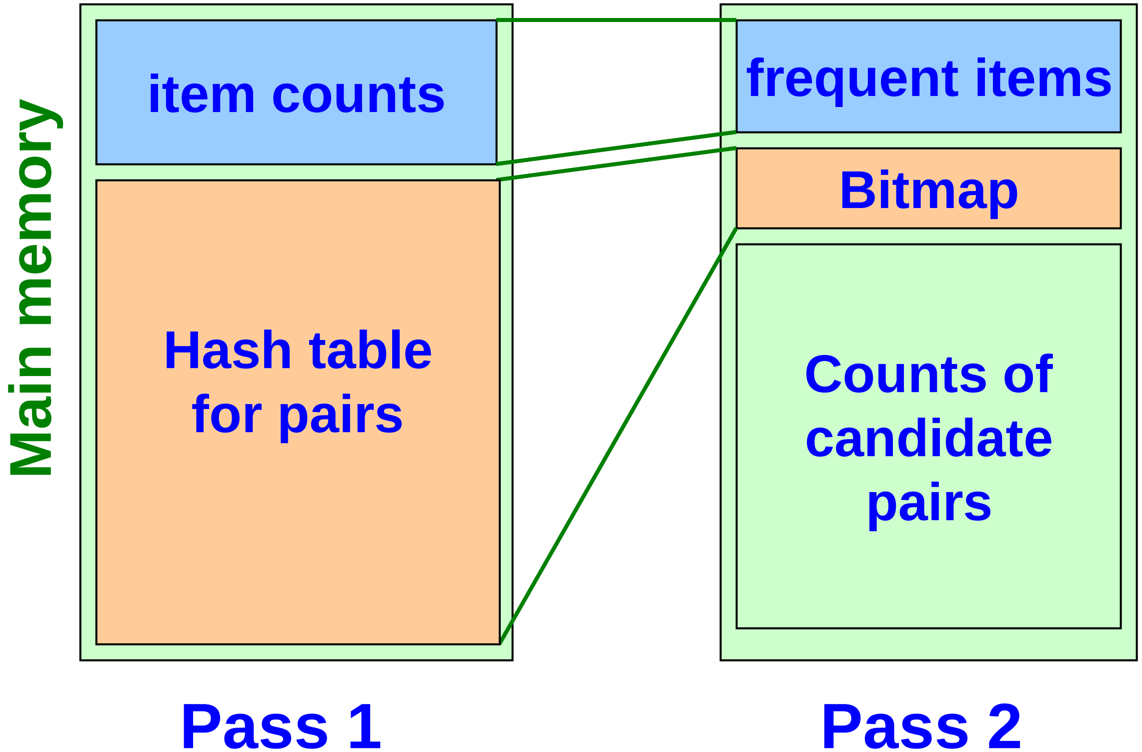
- If a bucket is not frequent, no pair in that bucket could possibly be frequent
  - ◆For a bucket with total count < s, none of its pairs can be frequent
  → We can safely eliminate pairs of non-frequent buckets

PCY Algorithm – Between Passes

- In pass 2, only count pairs that hash to frequent buckets
  - We must count again because:
    - we did not keep the information on the pairs
    - collisions are possible
  - We do not need the count information from pass 1 any more
  - What we need is an indication on whether a pair is possibly frequent or not
- Bit vector serves this purpose well (and consumes less space)
  - 1 means bucket count exceeds the support **s** (it is frequent); 0 for non-frequent
  - The hash value now corresponds to the bit position
- 4-byte (32-bit) integers are replaced by bits → bit-vector requires 1/32 of memory
- Also, decide which items are frequent and list them for the second pass

PCY Algorithm – Pass 2

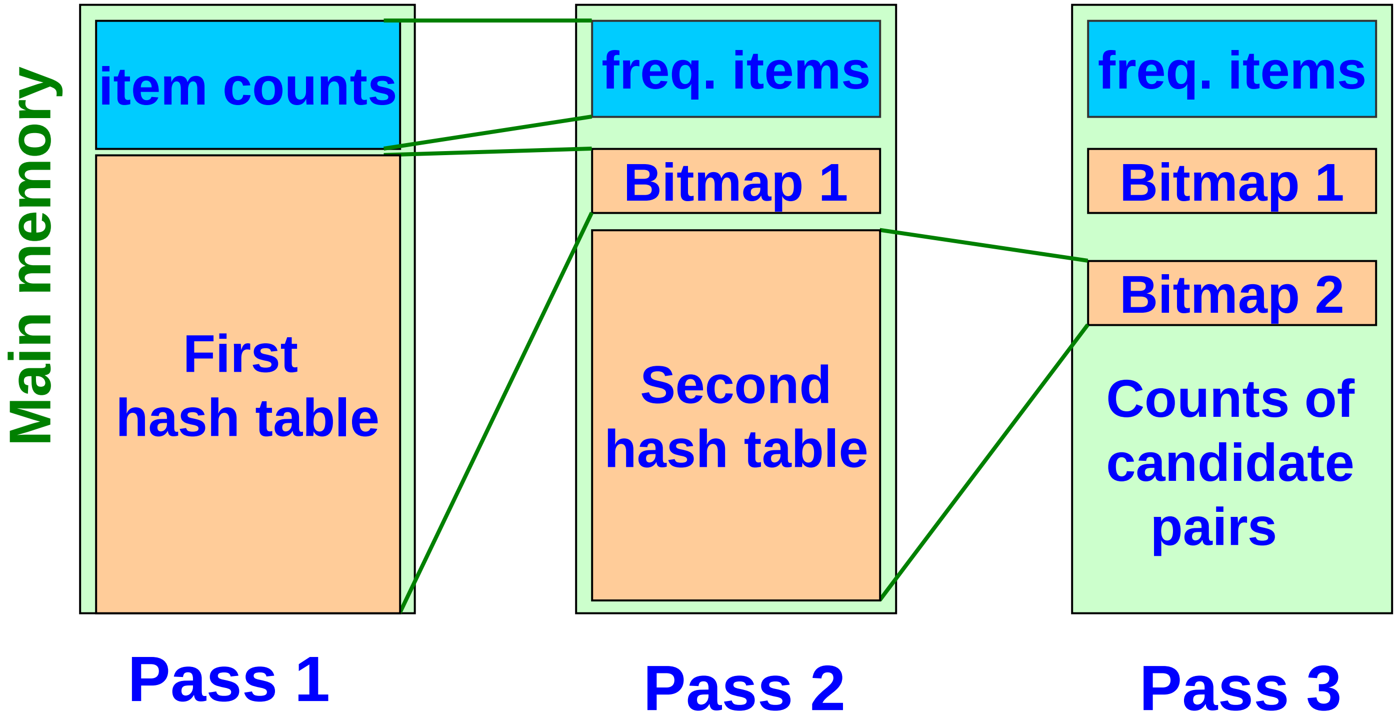
- Count all pairs $\{i, j\}$ that meet the conditions for being a candidate pair:
  - Both i and j are frequent items
  - The pair $\{i, j\}$, hashes to a bucket whose bit in the bit vector is 1
- Both conditions are necessary for the pair to have a chance of being frequent



Refinement: A *Multistage* Algorithm

- Limit the number of candidates to be counted
  - Remember: memory is the bottleneck
  - Still need to generate all itemsets but we only want to count/keep track of the ones that are frequent
- Key idea: After Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY
  - *i* and *j* are frequent, and
  - `{i,j}` hashes to a frequent bucket from Pass 1
- On *middle* pass, fewer pairs contribute to buckets, so fewer *false positives* – frequent buckets with no frequent pair
- Uses several successive hash tables---requires more than two passes

Multistage Picture



Pass 1

Count items
Hash pairs `{i,j}`

Pass 2

Hash pairs `{i,j}` into Hash2 iff:
`i,j` are frequent,
`{i,j}` hashes to freq. bucket in B1

Pass 3

Count pairs `{i,j}` iff:
`i,j` are frequent,
`{i,j}` hashes to freq. bucket in B1
`{i,j}` hashes to freq. bucket in B2

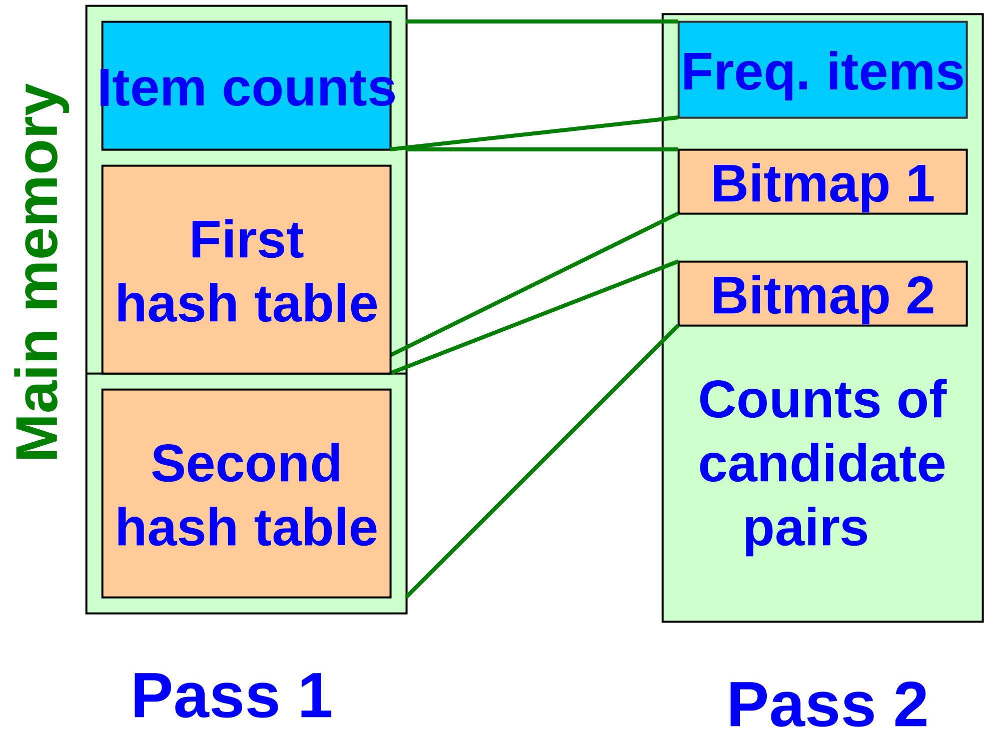
Multistage – Pass 3

- Count only those pairs $\{i, j\}$ that satisfy these candidate pair conditions:
  - Both i and j are frequent items
  - Using the first hash function, the pair $\{i, j\}$ hashes to a bucket whose bit in the first bit-vector is 1
  - Using the second hash function, the pair hashes to a bucket whose bit in the second bit-vector is 1

- Important Points
  - The two hash functions have to be independent
  - We need to check both hashes on the third pass
    - If not, we would wind up counting pairs of frequent items that hashed first to an infrequent bucket but happened to hash second to a frequent bucket
      - reduces the number of false positives!

Refinement: The Multihash Algorithm

- Key idea: use several independent hash tables on the first pass
- Risk: halving the number of buckets doubles the average count
  - We have to be sure most buckets will still not reach count *s*
- If so, we can get a benefit like multistage, but in only 2 passes!



So far, …

- Numerous approaches and refinements have been studied to keep memory consumption low
  - PCY and its refinements (multistage, multihash)

- Either multistage or multihash can use more than two hash functions
  - In multistage, there is a point of diminishing returns, since the bit-vectors eventually consume all of main memory
  - For multihash, the bit-vectors occupy exactly what one PCY bitmap does, but too many hash functions makes all counts $\geq s$

Limited Pass Algorithms

All (Or Most) Frequent Itemsets in $\leq$ 2 Passes

- APriori, PCY, etc., take *k* passes to find frequent *k*–itemsets
- Can we use fewer passes?
- Use 2 or fewer passes for ALL sizes, but may miss some frequent itemsets
  - Approximate solutions
    - Simple algorithm: Use random sampling
    - Savasere, Omiecinski, and Navathe (SON) algorithm
    - Toivonen

Random Sampling

- Take a random sample of the market baskets
- Load the sample in main memory
  - no disk I/O each time you increase the size of itemsets
- Use as your support threshold *s* a suitable, scaled-back number
  - E.g., if your sample is 1/100 of the baskets, use *s*/100 as your support threshold instead of *s*
  - be sure you leave enough space for counts
- Run Apriori or one of its improvements (for itemsets of all sizes, not just pairs)

Copy of sample baskets
Space for counts

Main Memory

Random Sampling: Option

- False positives: Itemset may be frequent in the sample but not in the entire dataset (because of the reduced minsup threshold)
  - Run a second pass through the entire dataset to verify that the candidate pairs are truly frequent
    - Results in eliminating false positives

- False negatives: Itemset is frequent in the original dataset but not picked out from the sample
  - Scanning the whole dataset a second time does **not** help
  - Using smaller threshold helps catch more truly frequent itemsets, but requires more space

SON Algorithm

- Instead of one random sample, process the entire dataset in memory-sized chunks
- An itemset becomes candidate if it is found to be frequent in *at least one* chunk using a scaled-back support threshold (e.g., s/p, where p is the number of chunks)
- On a second pass, count all the candidate itemsets and determine which ones are truly frequent in the entire set
  - No false positives again

- Key "monotonicity" idea: an itemset cannot be frequent in the entire set of baskets unless it is *frequent in at least one chunk*
  - A chunk contains a fraction 1/p of whole file (number of chunks is p)
  - If an itemset is not frequent in any chunk, then the support in each subset is less than $s * 1/p = s/p$ (the scaled-back support threshold)
  - Hence, the support in whole file is less than $s/p * p = s$
    - not frequent!

SON Distributed Version

- SON lends itself to *distributed data mining*
  - MapReduce

- Baskets distributed among many nodes
  - Subsets of the data may correspond to one or more chunks in distributed file system
  - Compute frequent itemsets at each node
    - Phase 1: Find candidate itemsets
  - Distribute candidates to all nodes
  - Accumulate the counts of all candidates
    - Phase 2: Find true frequent itemsets

SON MapReduce: Phase 1

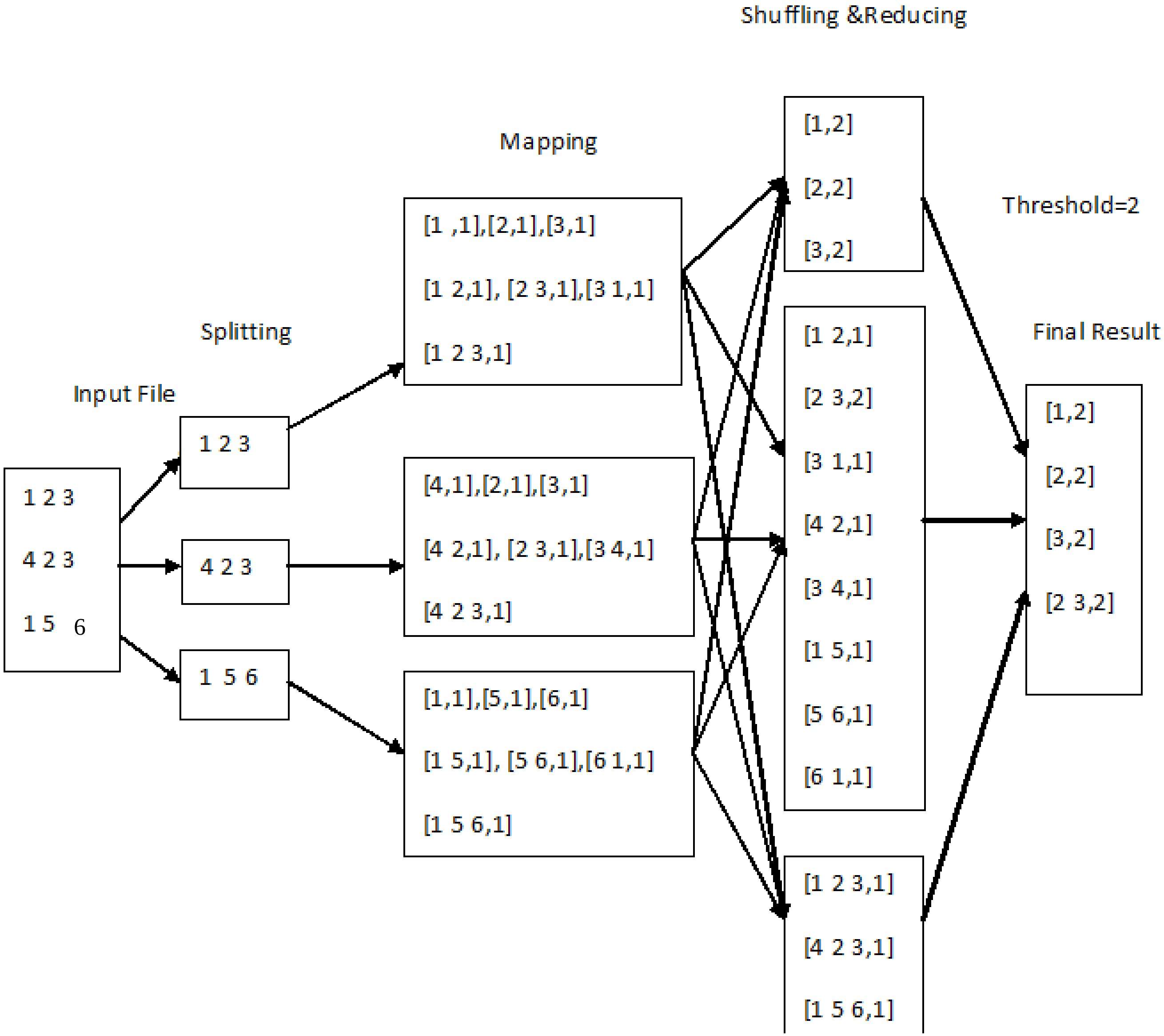
- Map
  - Input is a chunk/subset of all baskets; fraction 1/p of total input file
  - Find itemsets frequent in that subset:
    - Use support threshold = s / p
  - Output is set of key-value pairs `(FrequentItemset,1)` where `FrequentItemset` is found from the chunk
- Reduce
  - Each reducer is assigned a set of keys (itemsets)
  - Produce keys that appear one or more times
  - Frequent in some subset; these are candidate itemsets

SON MapReduce: Phase 2

- Map
  - Each Map task takes a chunk of the total input data file as well as the output of Reduce tasks from Phase 1
    - All candidate itemsets go to every Map task
  - Output pairs `(CandidateItemset,support)` where the `support` of the `CandidateItemset` is computed among the baskets of the input chunk

- Reduce
  - Each Reduce task is assigned a set of keys, which are candidate itemsets
  - Sums associated values for each key: total support for `CandidateItemset`
  - If total support of itemset >=`s`, emit itemset and its count

SON MapReduce (2 in 1)



www.hadooptpoint.com/finding-frequent-itemsets-using-hadoop-mapreduce-model/

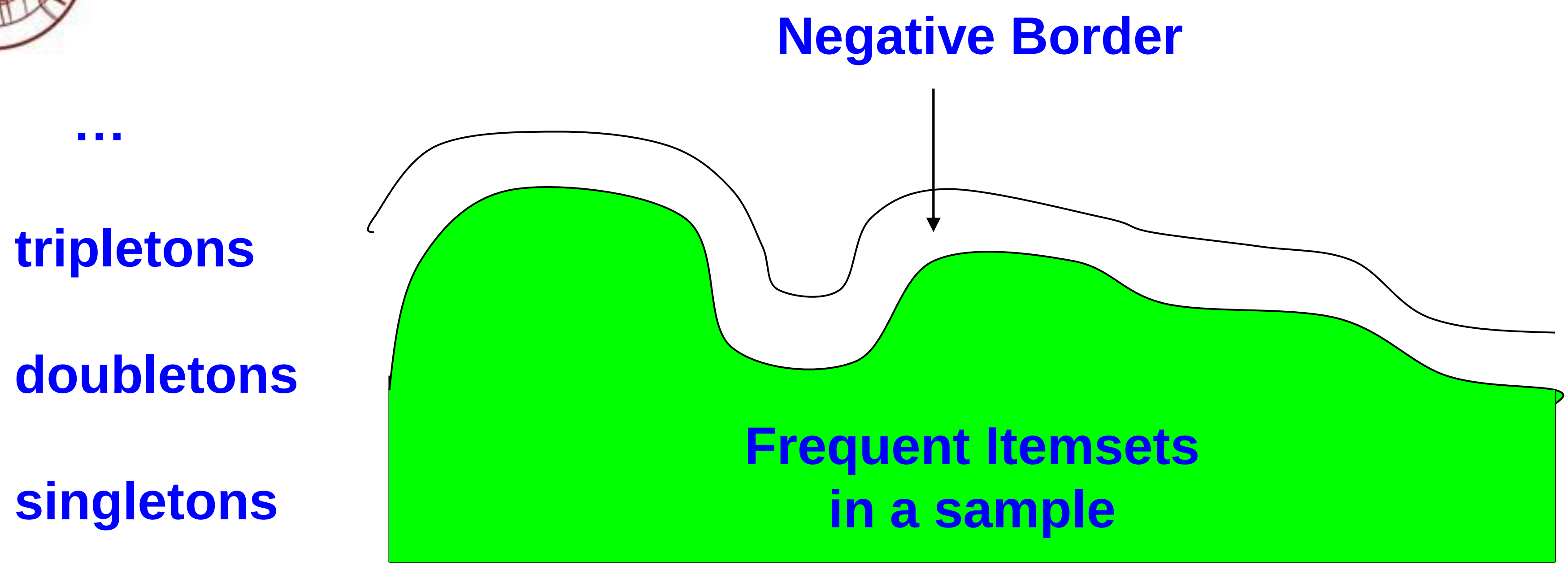
Toivonen's Algorithm

- A *heuristic* algorithm for finding frequent itemsets
- Given sufficient main memory, uses one pass over a small sample and one full pass over data
  - No false positives (always check against the whole)
- BUT, there is a small chance of false negatives
  - May not identify some frequent itemsets
- Then must be repeated with a different sample until it gives an answer
  - small number of iterations needed

Toivonen's Algorithm – First Pass

- Start as in the random sampling algorithm, but lower the threshold slightly for the sample
  - For fraction p of baskets in sample, use 0.8ps (0.9ps) as support threshold
- Goal: avoid missing any itemset that is frequent in the full set of baskets
  - The smaller the threshold the more memory is needed to count all candidate itemsets and the less likely the algorithm will not find an answer
- First pass: Find the itemsets that are frequent in a sample, AND the itemsets that are in the *negative border* of that sample
  - **Negative border:** An itemset is in the negative border of a sample if
    - it is *not* frequent in that sample,
    - but *all* its immediate subsets are

Example: Negative Border



- *ABCD* is in the negative border if and only if:
  1. It is not frequent in the sample, but
  2. *ABC*, *BCD*, *ACD*, and *ABD* are
- *A* is in the negative border if it is not frequent in the sample
  - Because its immediate subset is the empty set (always frequent)
    - unless there are fewer baskets than the support threshold (silly case)

Toivonen's Algorithm – Second Pass

- In a second pass, count all candidate frequent itemsets from the first pass, and also count their negative border
- If no itemset from the negative border turns out to be frequent, then the candidates found to be frequent in the whole data are *exactly* the frequent itemsets
- What if we find that something in the negative border is actually frequent?
  - We must start over again!
- Try to choose the support threshold so the probability of failure is low, while the number of itemsets checked on the second pass fits in main-memory

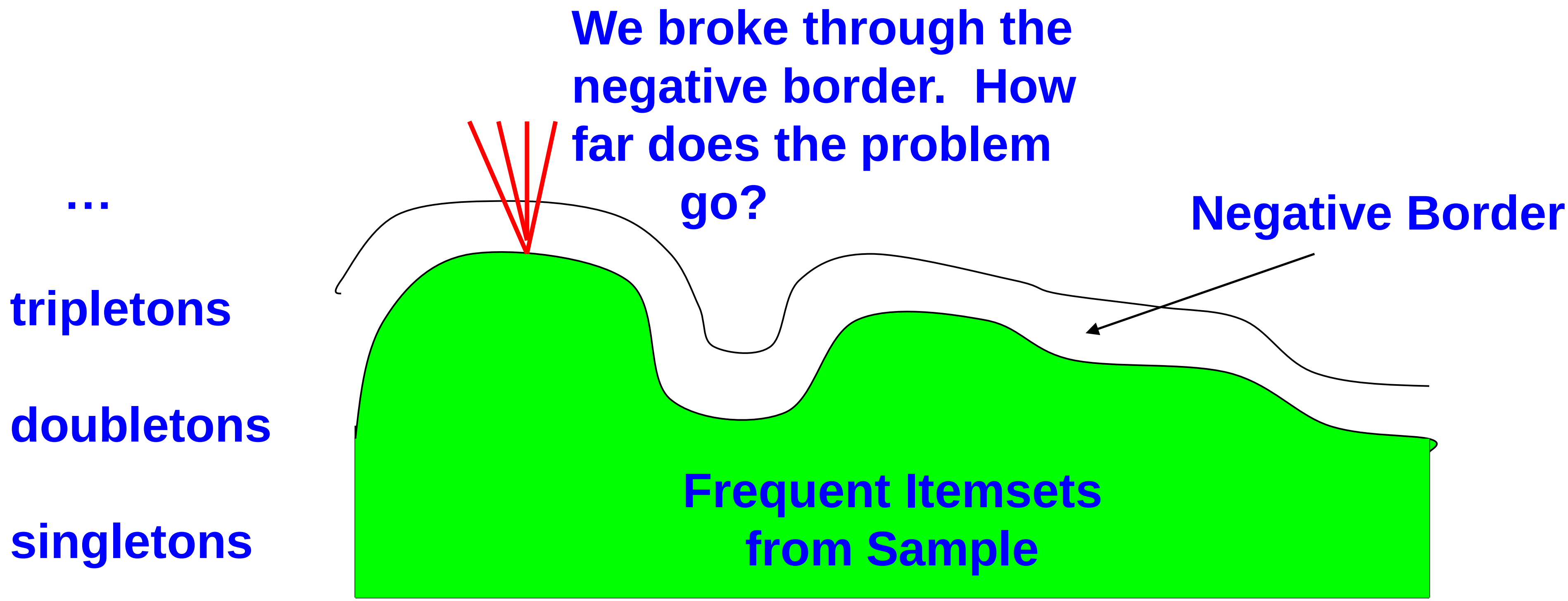
Theorem 1

- Given a data set D and a sample $S \subseteq D$,
  if there is an itemset T that is frequent in D but not frequent in S, then
  there is an itemset T' that is frequent in D and is in the negative border of S
  - False negatives appear in the negative border
- Proof (by contradiction): Suppose that:
  1. There is an itemset $T \in S$ that is frequent in D but not frequent in S, and
  2. No itemset in the negative border of S is frequent in D
- Let *T'* be an immediate subset of *T* that is not frequent in S
- All subsets of T are also frequent in D (*T* is frequent + anti-monotonicity of supp)
  - *T'* is frequent in D
- *Thus*, *T* is in the negative border of S (else not "immediate subset")

Theorem 2

- Given a data set D and a sample S ⊆ D,
if there is an itemset T that is frequent in D and is in the negative border of S,
then there is an itemset that is frequent in D but not frequent in S
  - By definition, any itemset in the negative border of S is not frequent in S. Hence T is frequent in D but not frequent in S
- During the second pass of the algorithm, if we found an itemset T of the negative border to be frequent in D, then we can assume by this theorem that there is an itemset that is frequent in D but not frequent in S;
  - in such a case, we are forced to *restart the algorithm* as we have already failed to discover at least one itemset that is frequent in D
- If we found no itemset of the negative border to be frequent in D, then by the previous theorem we are permitted to terminate the algorithm as we have discovered all the frequent itemsets of D

If Something in the Negative Border is Frequent . . .



Toivonen's Algorithm

- Provides a simplistic framework for discovering frequent itemsets in large data sets while also providing enough flexibility to enable performance optimizations directed towards particular data sets

- Allows the discovery of all frequent itemsets through a sampling process

- Numerous optimizations and approximations can be made to improve the algorithm's performance on data sets with particular properties
    - E.g., using a slightly lowered threshold will minimize the omission of itemsets that are frequent in the entire dataset
        - such omissions result in additional passes through the algorithm
    - The support threshold should also be kept reasonably high
        - so that the counts for the itemsets in the second pass fit in main memory

Summary

- Market-Basket Data and Frequent Itemsets
  - Many-to-Many relationship
- Associating rules
  - Confidence and Support
- The Apriori Algorithm
  - Combine only frequent subsets
- The PCY algorithm
  - Hash pairs to reduce candidates
- Multi-stage and Multi-hash algorithm
  - Multiple hashes
- Randomized and SON algorithm
  - Sample, divide into chunks and treat as samples by MapReduce
- Toivonen's Algorithm
  - Negative Border

References

- CS246: Mining Massive Datasets Jure Leskovec, Anand Rajaraman, Jeff Ullman, Stanford University, 2014
- CS5344: Big Data Analytics Technology, TAN Kian-Lee, National University of Singapore 2014
- CS059: Data Mining, Panayiotis Tsaparas University of Ioannina, Fall 2012

Research on Pattern Mining: A Road Map