Analysis of Massive Data Sets

http://www.fer.hr/predmet/avsp

Prof. dr. sc. Siniša Srbljić

Doc. dr. sc. Dejan Škvorc

Doc. dr. sc. Ante Đerek

Faculty of Electrical Engineering and Computing
Consumer Computing Laboratory

Finding Frequent Itemsets

Goran Delač, PhD

Outline

- **Motivation**
  - Market-Basket model
  - Applications
- **Finding frequent itemsets**
  - Computation model
  - A-Priori algorithm
  - Refinements

Motivation

Market-Basket Model

Market-Basket Model

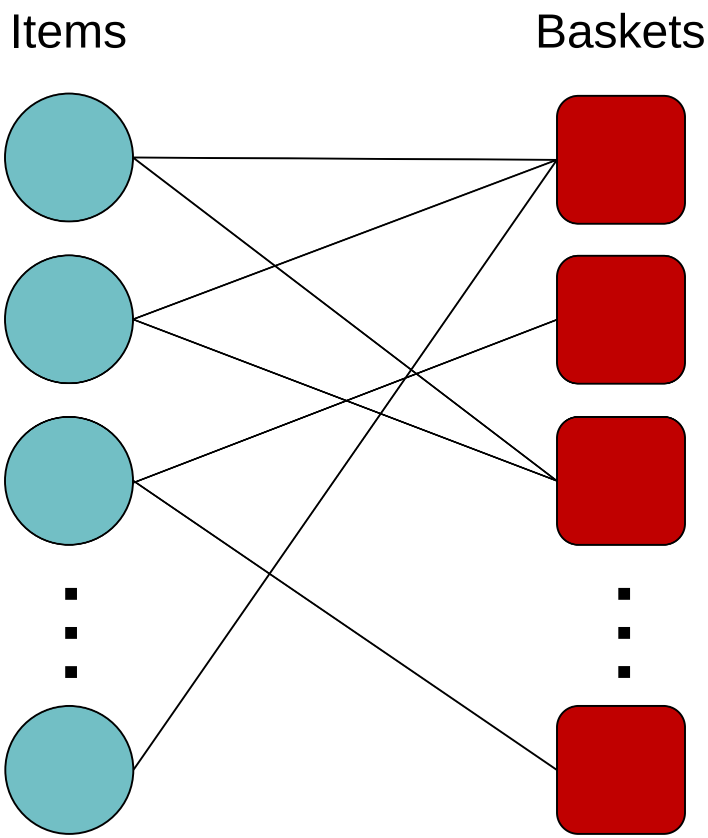
Many-to-many relationship
between ***items*** and ***baskets***

Problem: How to identify items that are ***frequently*** bought together?

Solution: Identification of association rules, i.e. discovery of frequent sets of items (baskets)

Market-Basket Model

Market-Basket Model: Many-to-many Relationship

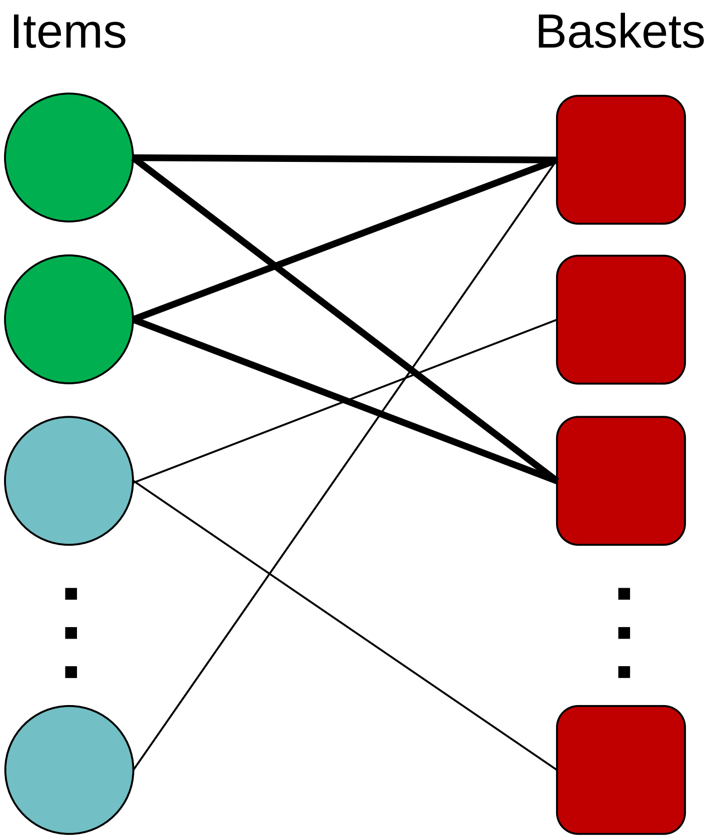


- Large number of items
- Large number of baskets
- Baskets contain a small subset of items

Problem: How many item pairs can be found in at least **k** baskets?

Market-Basket Model

Market-Basket Model: Many-to-many Relationship



- Large number of items
- Large number of baskets
- Baskets contain a small subset of items

Problem: How many item pairs can be found in at least **k** baskets?

k=2
The two green items are **often** bought together

Market-Basket Model

The story of beer and diaper

People with small children often buy beer and diapers together as they lack the time to go out

Rule: Put close together beer, diaper + … ? (chips,..)

Trick: Lower the price of beer, rise the price of diapers

Association rules need to be used ***frequently*** in order to be profitable

Frequent Itemsets vs. Similar Items

Frequent itemsets are about finding the ***absolute*** number a certain itemset appears (***frequency!***)

Finding similar items is about finding very ***similar***, but not necessarily the same sets of items (frequency is **not** important)

Frequent Itemsets vs. Similar Items

Frequent itemsets – traditional retailers (physical stores)

- Rules need to be frequent in order to be profitable – the store is a physical entity used by all customers

Finding similar items – web stores

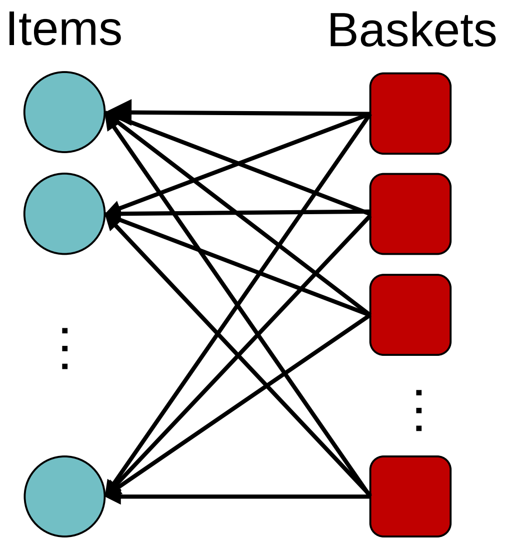
- It pays off to know which items could be possibly bought together even by a small number of customers
- Finding correlation between rarely purchased items
- Profiting from the long tail

Applications

- **Any general many-to-many mapping between two kinds of entities**
- **Market-Basket Model**
  - Finding items that are frequently bought together
- **Detecting drug side-effects**
  - Baskets: patients, Items: drugs and side-effects
- **Plagiarism**
  - Baskets: sentences, Items: documents
- **Biomarkers**
  - Baskets: patient information, Items: biomarkers and diseases

Applications

- **Finding Communities in Graphs (e.g. Facebook, Twitter)**
  - Baskets: nodes, Items: outgoing neighbours
  - Searching for **complete** bipartite subgraphs



Frequent Itemsets

- **Itemset Support Threshold**
  - Support threshold ***s*** for itemset ***i*** is the number of baskets that contain all items in ***i*** (in the observed dataset)
  - Expressed as an absolute number of baskets or more often as a fraction of the total number of baskets

- **Definition**
  - Given a support threshold ***s***, the item subsets that appear in at least ***s*** baskets are called ***frequent itemsets***

Frequent Itemsets

- **Items = {<u>b</u>eer, <u>c</u>hips, <u>j</u>uice, <u>w</u>ine}**
- **Support threshold = 3 / 6 baskets**

$B_1 = \{b, c, j, w\}$

$B_2 = \{b, c, j\}$

$B_3 = \{b, j, w\}$

$B_4 = \{c, j, w\}$

$B_5 = \{b, c\}$

$B_6 = \{b, w\}$

Frequent itemsets: {b}, {c}, {j}, {w}, {b,c}, {b,j}, {b,w}, {c,j}, {j,w}

Association Rules

- **Application of frequent itemsets!**

- **If-then rules that describe what a basket is likely to contain**
  - $\{i_1, i_2, i_3, \ldots i_n\} \rightarrow j$ implies that if a basket contains items $i_1, i_2, i_3, \ldots i_n$, it is also likely to contain item j

- **In practice, we want to find rules that really matter**
  - **Confidence**
  - **Interest**

Association Rules

- **Confidence**
  - For $\boldsymbol{I = \{i_1, i_2, i_3, \ldots\ i_n\}}$
  - $\boldsymbol{conf(I \rightarrow j) = \text{support}(I \cup j)\ /\ \text{support}(I)}$

- **Not all high-confidence rules are interesting**
  - E.g. the rule $X \rightarrow$ *beer* can have high confidence if *beer* is simply purchased very often (independently of X)

Association Rules

- **Interest**
  - ***Fr* [*j*]** fraction of baskets containing item *j*
  - ***interest*($I \rightarrow j$) = *conf*($I \rightarrow j$) - *Fr* [*j*]**

$B_1 = \{b, c, j, w\}$ $B_4 = \{c, j, w\}$

$B_2 = \{b, c, j\}$ $B_5 = \{b, c\}$

$B_3 = \{b, j, w\}$ $B_6 = \{b, w\}$

$c \rightarrow b$

conf ($c \rightarrow b$) = 3 / 4

Fr(b) = 5 / 6

interest($c \rightarrow b$) = 3/4 – 5/6

Association Rules

- Finding association rules
  - Find all association rules with
    - Support ≥ s (support of $i_1, i_2, i_3, \ldots i_n$)
    - Confidence ≥ c
  - If a rule $\{i_1, i_2, i_3, \ldots i_n\} \rightarrow j$ has high support and confidence, both $\{i_1, i_2, \ldots i_n\}$ and $\{i_1, i_2, \ldots i_n, j\}$ will be ***frequent***
    - In realistic scenarios s ≥ 1%, c ≥ 50%

Association Rules

- Finding association rules

1. Find all frequent itemsets ***I*** (taking into account ***s***)

2. Generate rules
   - For every subset ***A*** of ***I*** generate rule ***A*** → ***I \ A***
   - Compute **confidence** and select appropriate rules

Key problem: Finding frequent itemsets

Finding frequent itemsets

- Algorithms
  - A-priori
  - Elcat (Equivalence Class Transformation)
  - FP-growth (Frequent Pattern)

Finding frequent itemsets

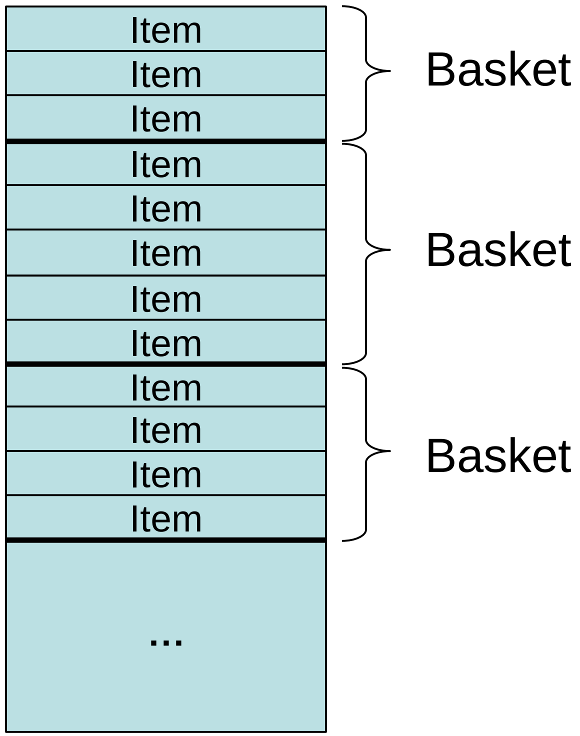
- Computation model
- A-priori algorithm
- Refinements
  - Handling larger data sets in memory
  - Limiting the number of passes

Finding frequent itemsets

- Naïve approach – Frequent pairs
  - Read the data once and count all the pairs
    - For each basket containing **n** items generate all the possible n(n-1) / 2 pairs in two nested loops
  - Issues: although **n** can be small, the overall number of items can be large
    - 253 million items sold by Amazon.com (August 2014)
    - That equals to 32 * 10^{15}
    - If pairs counts are 4-byte integers, approximately 128 million gigabytes are needed to store them!

Computation Model

- **Data Representation**
  - Data is usually stored in flat files
    - Stored on disk (or possibly DFS)
    - Sequences of baskets
    - Average basket size is small
  - The file does not fit into main memory
    - Many items and baskets
    - Baskets are expanded into pairs, triplets, …
    - I/O operations
    - Cost?



Computation Model

- **Cost of mining frequent itemsets**
  - Counting item subsets in each basket
    - **k** nested loops are used to generate subsets of size **k**
    - Usually, **k** is small and the baskets are small (contain **n** items), e.g. for basket size n = 30 and k = 2 there are $\binom{30}{2} = 435$ pairs
  - I/O operations - often dominant cost
    - Dataset often does not fit into main memory
    - Even if **k** gets large, **n** will decrease as there will be less baskets that meet the support threshold **s**, i.e. the itemsets in baskets are not considered frequent
    - Appropriate support threshold **s** needs to be selected

Computation Model

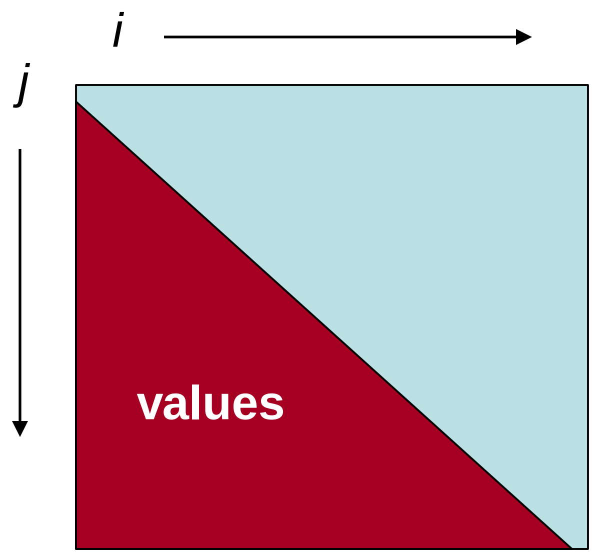
- **Main memory bottleneck**
  - Memory needs to be sufficiently large to store frequent itemset counts
  - Algorithms are **limited** by the memory size
    - Swapping the count data would diminish performance by many orders of magnitude

Computation Model

- **How to make it work for massive datasets?**
  - Limit the number of passes an algorithm makes over the data
  - Make frequent itemset counts fit into main memory

Computation Model

- **Storing counts in memory**
  - Item names should be indexed as integers (hash map)
  - Triangular matrix
    - 2-dimensional array $a[\,i, j\,]$, $i < j$



Computation Model

- **Storing counts in memory**
  - Triangular matrix
    - More generally one-dimensional array is used
    - Pair { i, j } is stored at $a[k]$ for $1 \le i < j \le n$ where:

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

    - If counts are stored as 4 byte integers, approximately $2n^2$ bytes are needed for n items

Computation Model

- **Storing counts in memory**
  - Hash table - Storing triplets
    - Triplet { *i*, *j, c* } is stored so that $i < j$ are item indices and c is the itemset count
    - **Hash table** { *i*, *j* } → *c*
    - If counts are stored as 4 byte integers, 12 bytes are needed to store a singe itemset count
    - However, itemset counts are stored **only** when ***c* > 0**
    - More efficient if significantly fewer then **1/3** of the possible itemsets occur in the baskets (realistic scenario!)

Computation Model

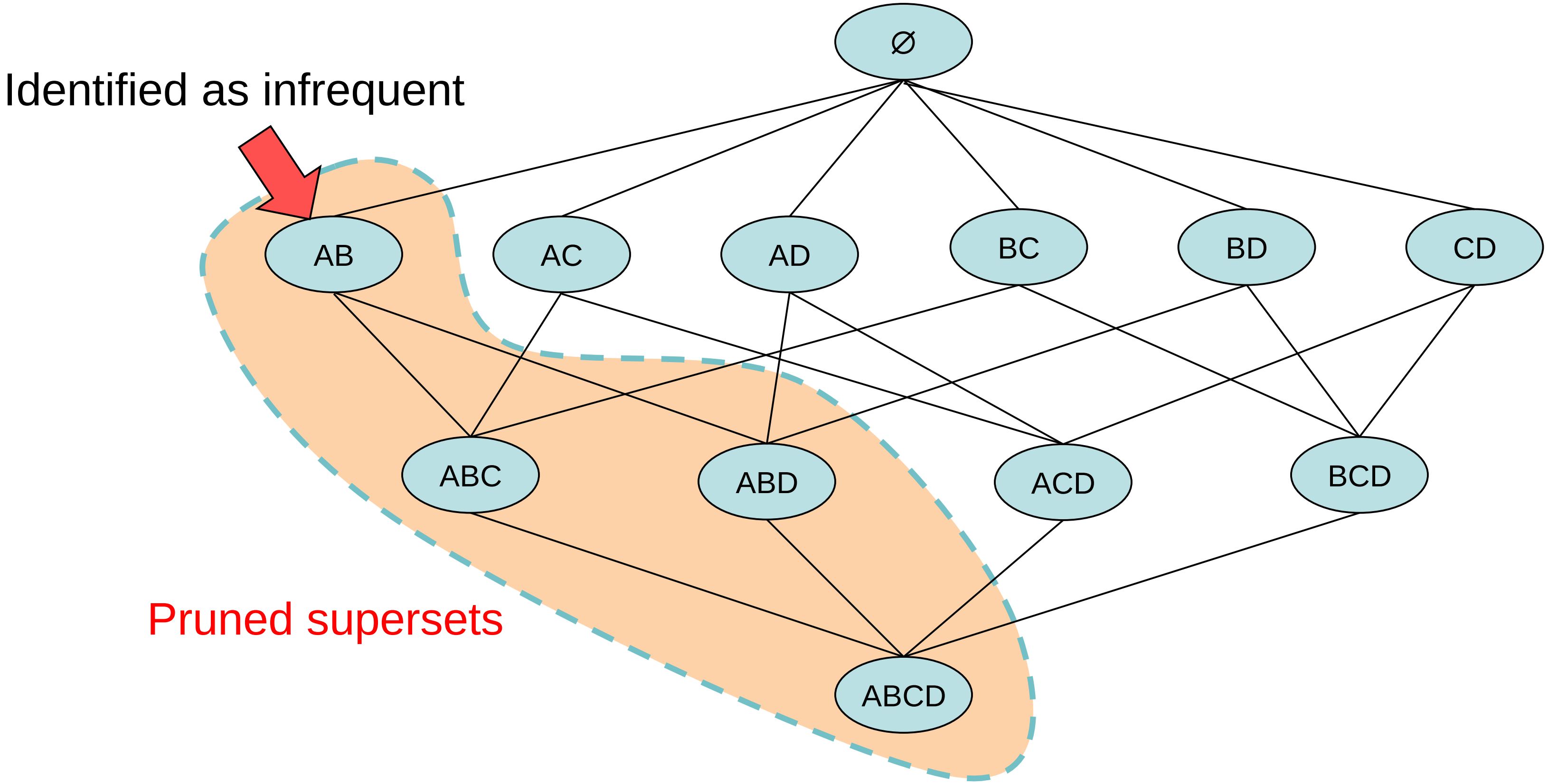
- **Storing counts in memory**
  - Triangular matrix
  - Storing Triplets (hash table)
  - Still the use of main memory presents limitations as it might not be possible to store the data for all itemsets (pairs, triplets, etc..)
  - Can this situation be improved?

A-Priori Algorithm

- **Rakesh Agrawal (1993)**
- **2-pass algorithm that limits the need for main memory**
- **Key idea: Monotonicity of itemsets**
  - If an itemset ***I*** is **frequent** then each of its **subsets** must also be **frequent**
  - If ***I*** appears ***s*** times then, its subset ***J*** appears at least ***s*** times (it can also appear in other itemsets!)
  - If no superset of an itemset ***I*** is frequent, ***I*** is called the **maximal itemset**
- **Bottom-up approach**
  - start by counting pairs, then triplets, etc.

A-Priori Algorithm

- Reducing problem space by pruning infrequent supersets



A-Priori Algorithm

- **First pass**
  - Generate a table that translates item names to integers
    - Hash table
  - Count occurrence of each item across all the baskets
    - Array indexed by indices stored in the hash table
    - Size of the array is proportional to the number of items **n**

- **Intermediate step**
  - Generate frequent items table
    - **Scan** through the **counts** and mark the **items** that **meet** the support threshold **s** as **frequent**

A-Priori Algorithm

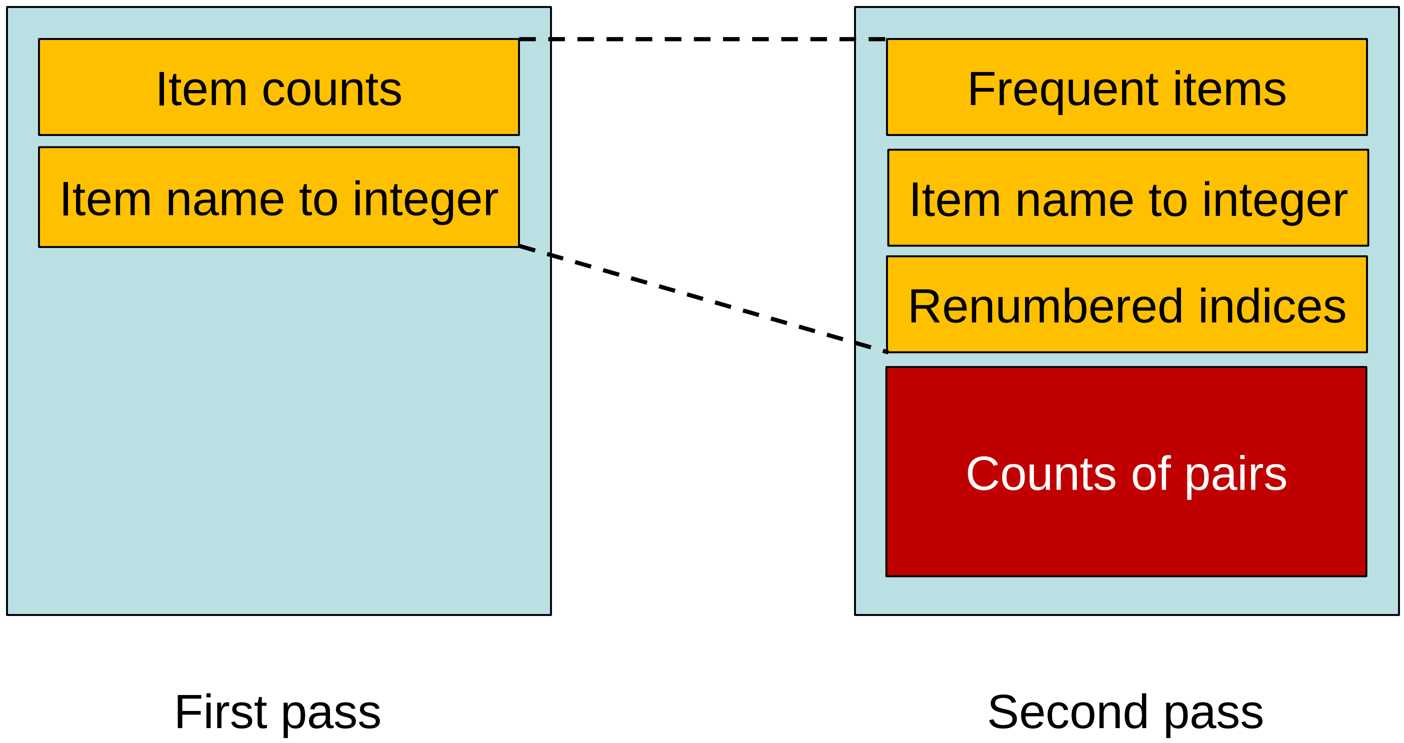
- **Second Pass**
  - For each basket look in the frequent items table and see which items are frequent
  - Generate all item pairs using **only** the **frequent items** in the basket
  - Add one to the count for each generated pair in the data structure used to store counts

A-Priori Algorithm

- **Second Pass**
  - If the counts are stored in a triangular matrix, items should be renumbered to save space
  - A new table that translates the old indices is added
  - Then the storage size is $2m^2 < 2n^2$ where m is the number of frequent items and n total number of items

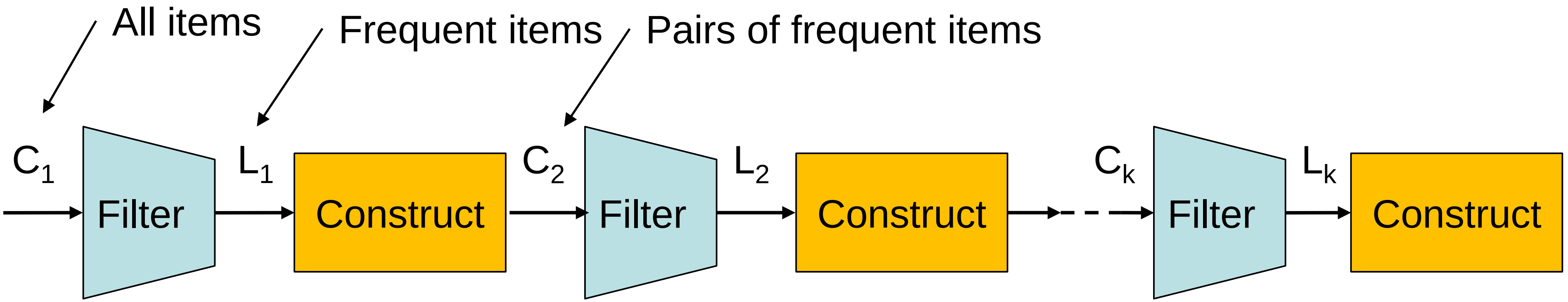
A-Priori Algorithm

- **Main memory**



A-Priori Algorithm

- **All frequent itemsets**
  - **k-tuple**: repeat first and second pass ***k*** times
  - $\mathbf{C_k}$ – candidate tuples, have support ≥ s according to step k -1
  - $\mathbf{L_k}$ – filtered C_k so that only frequent k-tuples remain



A-Priori Algorithm

- **Beyond A-Priori**
  - Limitations on main memory
    - PCY algorithm
  - k passes through the data set are needed to find k-tuples
    - Random sampling
    - SON algorithm

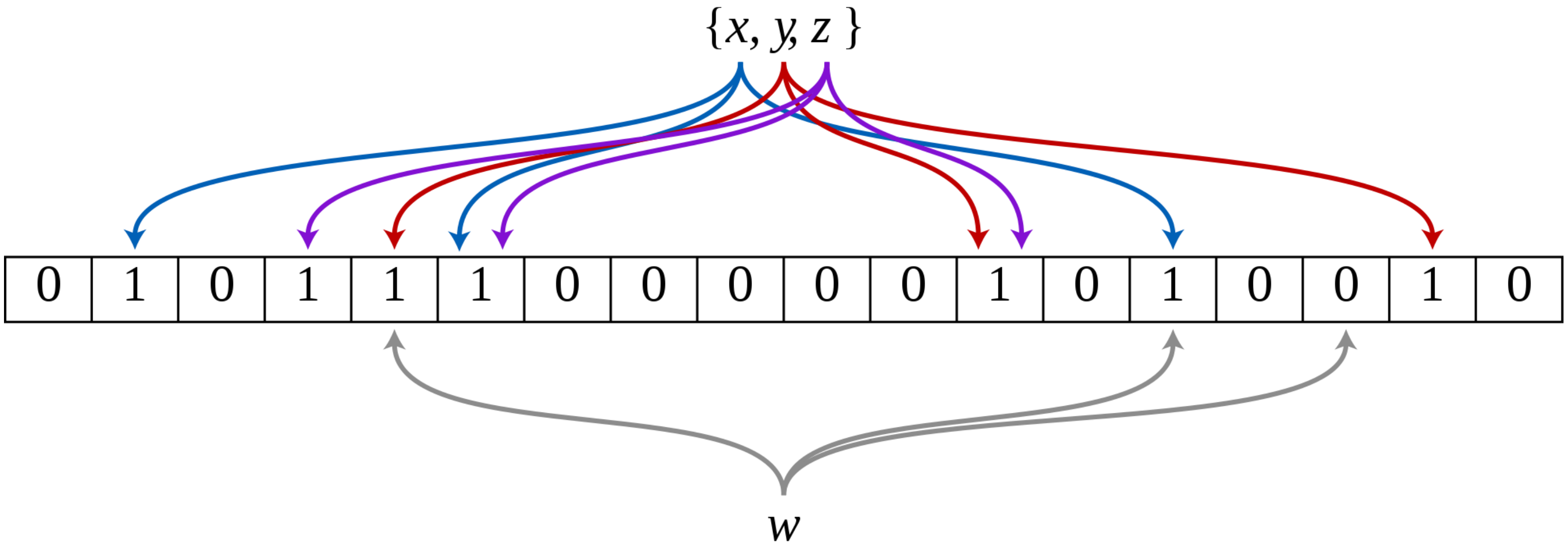
PCY Algorithm

- **Park, Chen, Yu (1995)**
- **Improvement to A-Priori algorithm**

- **Idea**
  - **Most** of the main **memory** in the **first pass** of the A-Priori algorithm is **free**
  - This free space can be used to **reduce the memory requirements** in the second pass (storing itemset counts can be resource demanding)

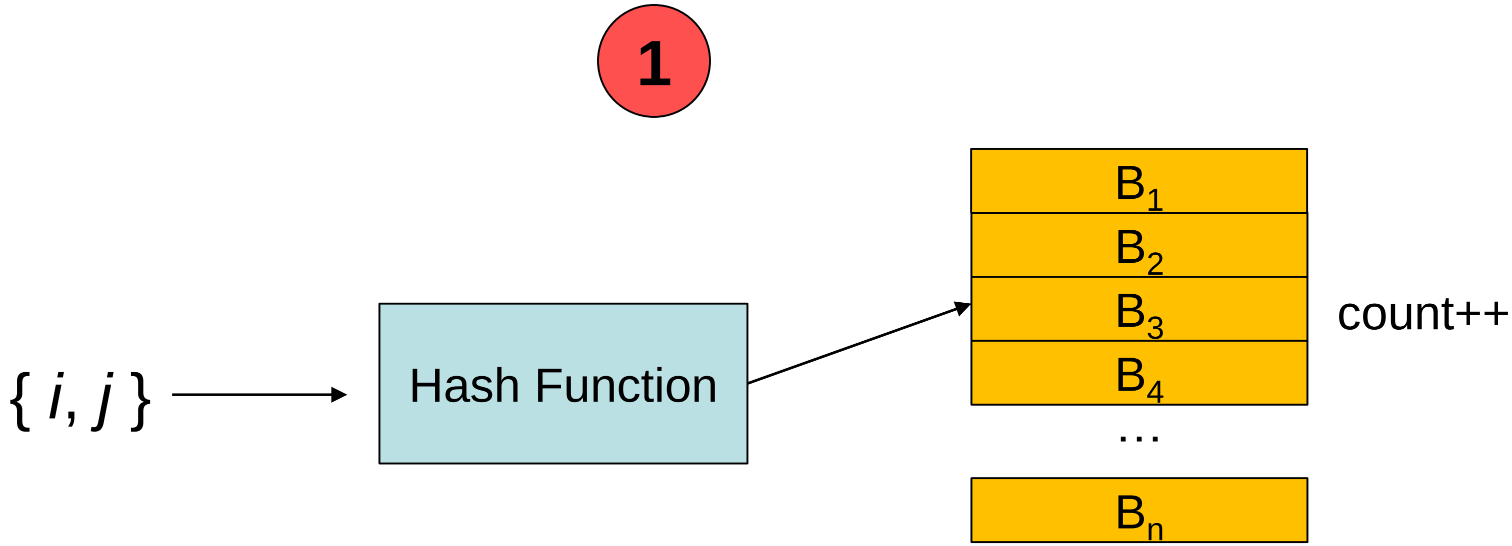
PCY Algorithm

- **How to further reduce memory requirements?**
  - Approach similar to **Bloom filters**



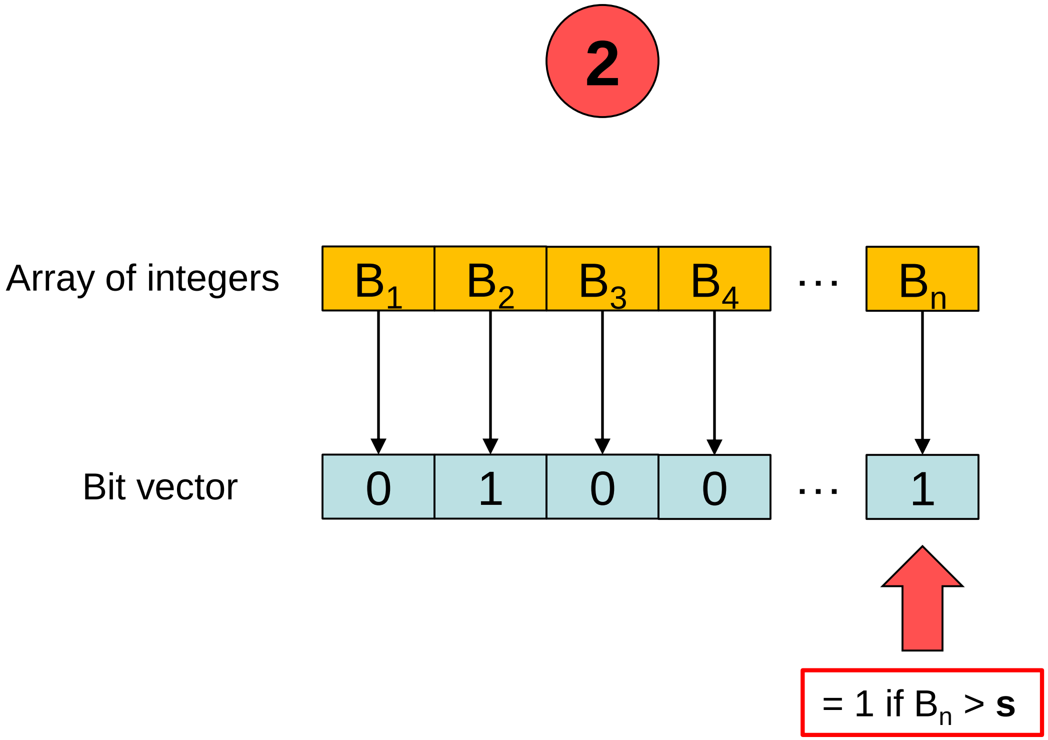
PCY Algorithm

- **How to reduce memory requirements?**
  - Construct a **filter** for infrequent item pairs in the **first pass**



PCY Algorithm

- **How to reduce memory requirements?**
  - Construct a **filter** for infrequent item pairs in the **first pass**



PCY Algorithm

- **First pass**
  - Count items, and generate name to index translation table (A-priori)
  - For each **basket** create **item pairs** in a double loop
    - Only item pairs that consists of frequent items are considered
  - **Hash** each item pair into a **bucket** and increase its count
    - There can be as many buckets as the memory permits but crucially there can be less buckets then there are item pairs

PCY Algorithm

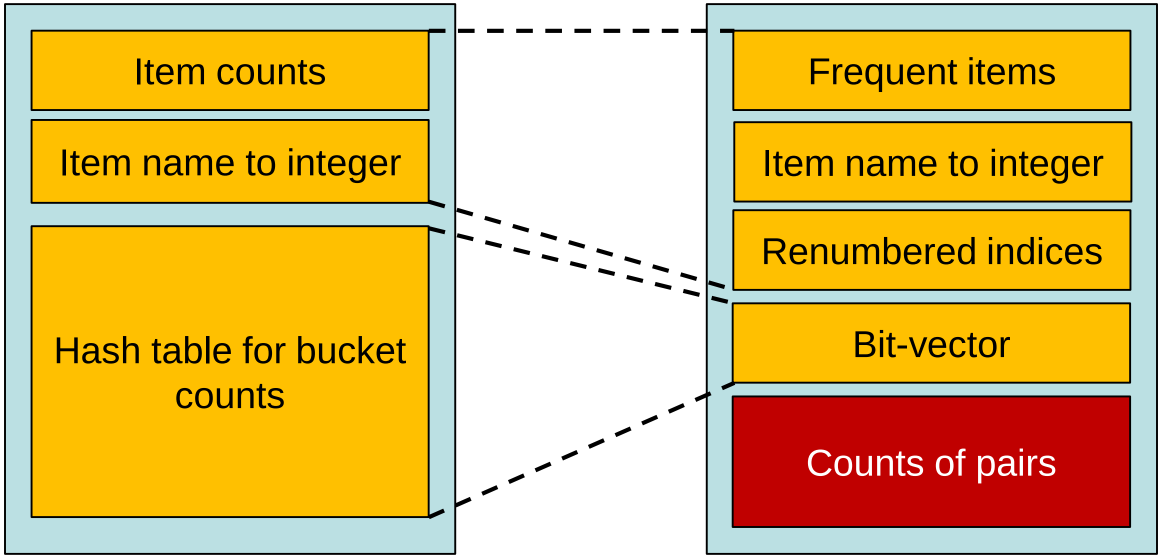
- **Intermediate step**
  - Convert bucket counts into a bit-vector
    - Value 1 is assigned to **frequent buckets**, i.e. those that have count greater or equal to support **s**
    - Value 0 is assigned otherwise
  - 4-byte integer counts are replaced by 1bit, therefore a 1/32 of memory is needed to store the bit-vector

PCY Algorithm

- **Observation on frequent buckets**
  - If a bucket contains a frequent itemset, then it will have count larger or at least equal to **s**, i.e. it will be frequent
    - **However, it can contain 0 frequent itemsets and still be frequent!**
  - If a bucket has count less then **s**, then **none** of the itemsets that hash to it **are frequent**
    - **These itemsets do not need to be counted in the second pass**

PCY Algorithm

- **Second pass**
  - Count the item pair ***{ i, j }*** if and only if:
    - *i* and *j* are **both frequent**
    - The item pair { *i*, *j* } hashes to a bucket that has its bit-vector value set to **1**

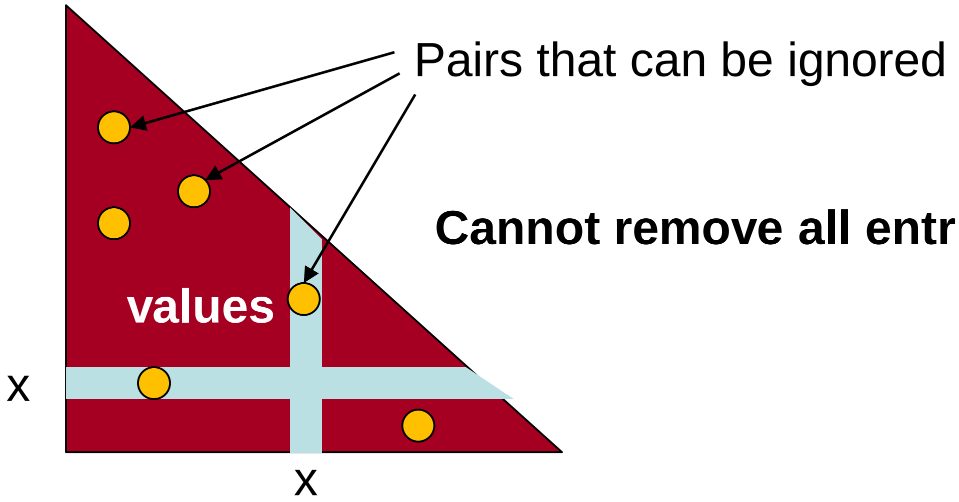


First pass

Second pass

PCY Algorithm

- **Remarks on the PCY algorithm**
  - It usually is more efficient but **not always**
  - Triangular matrix is not suitable to store counts
  - **Why?**
    - Pairs that can be skipped are randomly scattered
    - There is no known way to avoid unused spaces in the matrix



Cannot remove all entries for x!

PCY Algorithm

- **Remarks on the PCY algorithm**
  - Counts should be stored as triplets (e.g. Hash map)
    - PCY is **useful only** if it allows us to avoid counting at least **2/3** of the item pairs
    - Otherwise, A-Priori is more efficient
  - Memory-wise improvements to A-priori, like PCY are generally useful when it comes to finding frequent pairs
  - For triplets, … it is usually sufficient to use A-priori as memory requirements tend to be much lower

PCY Algorithm

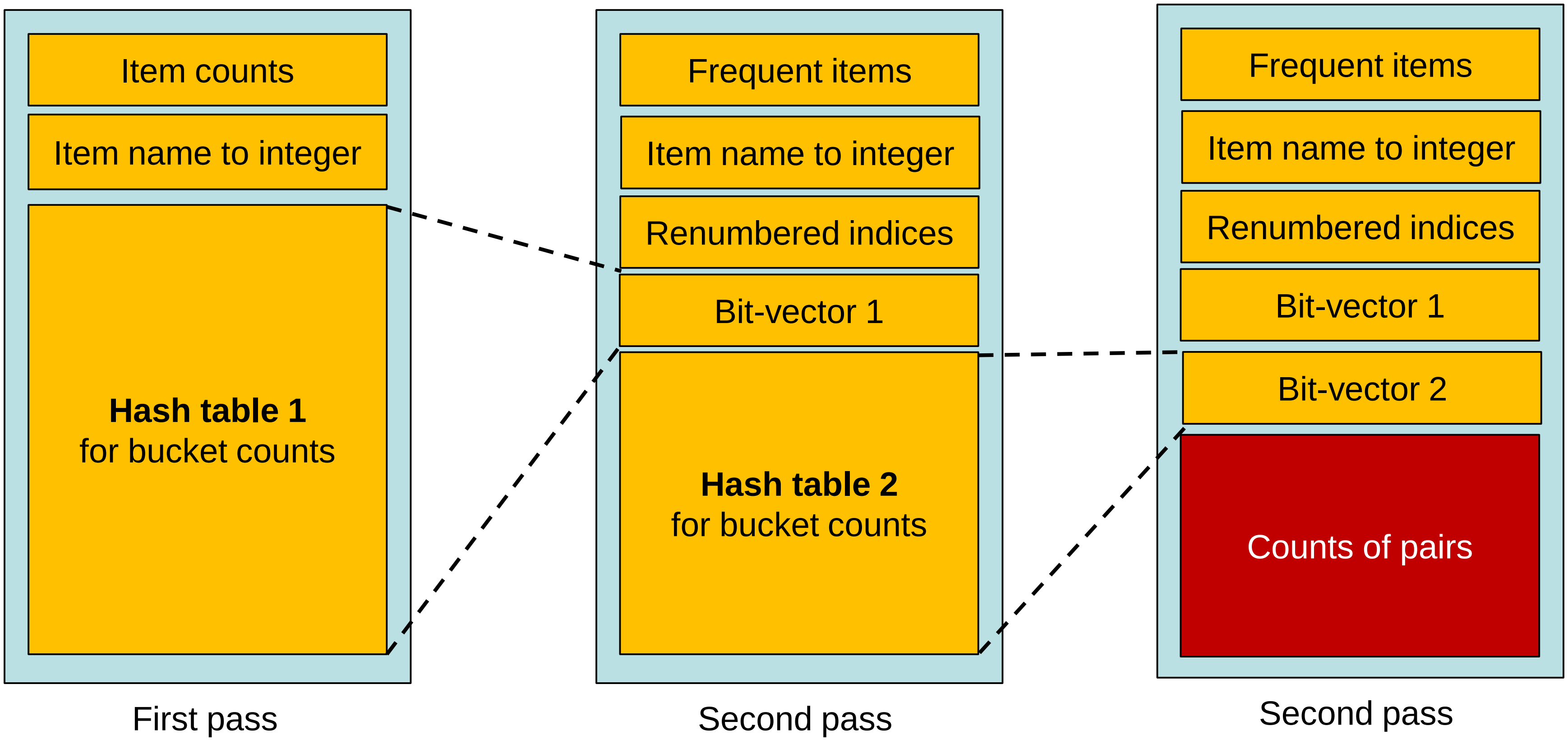
- **Further improvements – Multistage Algorithms**

  - After first pass of PCY item pairs that are candidates to be frequent are **rehashed**

  - Number of candidate pairs is reduced, but an **additional pass** through the dataset is **needed**

PCY Algorithm

- Further improvements – Multistage Algorithms



Limiting the Number of Passes

- **Find frequent itemsets in ≤ 2 passes**
  - Fewer passes at the cost of not finding all frequent itemsets
    - This can be acceptable as having too many association rules is not useful

- **Random sampling**

- **SON algorithm**

Random Sampling

- **Randomly select a sample of the dataset**
  - Each entry is selected with probability **p**
    - Sample size is approx. **p** x **n**
  - If baskets are stored randomly
    - First p x n baskets are selected
    - Random DFS chunks are selected

- **Run A-priori or its improvements on the sample <u>in the main memory</u>**
  - Support threshold should be reduced proportionally
    - If sample size is 1% of the dataset s' should be approx. s/100

Random Sampling

- **Not all frequent itemsets will be discovered**
  - **False positives**: itemsets frequent in the sample, but not in the dataset
  - **False negatives**: itemsets frequent in the data set, but not in the sample

  - However, if an itemset has support >> **s**, there is a low probability of it being a false negative

Random Sampling

- **False positives**
  - Can be eliminated by performing an additional pass in which their frequency is verified against **s**
    - Other itemsets do not need to be counted!

- **False negatives**
  - Can be reduced by further decreasing the support threshold and then eliminating false positives
  - Support threshold **s'** = 0.9 **p** x **n**
  - Lowering **s'** increases memory requirements as more items are counted

SON Algorithm

- **Savasere, Omiecinski and Navathe (1995)**

- **Avoids false positives and negatives**
  - 2 full passes

- **Idea: Divide input into chunks**
  - Chunks are treated as samples
  - Apply random sampling algorithm with support **s’** set to: **p** x **s**, where **p** is chunk size (fraction of the original file)

SON Algorithm

- **Once chunks are processed**
  - Union of all itemsets that are frequent in at least one chunk
    - These are candidates for frequent itemsets
    - **Any** frequent itemset will **surely be frequent** in at least one chunk – no false negatives
- **Second pass:**
  - Count candidate itemsets and eliminate false positives

SON Algorithm: MapReduce

- **2 MapReduce passes**

- **First pass**
  - Map
    - **Input:** File chunk, size **p** x **n**, **p** – fraction of input file
    - Count item occurrences and select those with support > **p** x **s**
    - **Output**: **(F_i, 1)** key: F-frequent item, value: irrelevant

  - Reduce
    - **Input:** (F_1, 1), (F_2, 1), (F_3, 1), …, (F_{n-1}, 1), (F_n, 1)
    - **Output:** F_1, F_2, F_3, … , F_{n-1}, F_n

SON Algorithm: MapReduce

- **Second pass**
  - Map
    - **Input:** portion of input file + (F_1, F_2, F_3, … , F_{n-1}, F_n)
    - Count occurrences of frequent itemset candidates
    - **Output**: **(F_i, v)** key: F-frequent itemset, value: **v** count for $\mathbf{F_i}$ in the input file chunk

  - Reduce
    - **Input:** (F_1, v_1), (F_2, v_2), (F_3, v_3), …, (F_{n-1}, v_4), (F_n, v_n)
    - Select only those itemsets that have support > **s**
    - **Output:** F_i such that v_i, > **s,** for each **i**

Literature

1. J. Leskovec, A. Rajaraman, and J. D. Ullman, "Mining of Massive Datasets“, 2014, Chapter 6 : "Frequent Itemsets" (link)

2. R. Agrawal, and R. Srikant: "Fast Algorithms for Mining Association Rules in Large Databases “, VLDB '94 Proceedings of the 20th International Conference on Very Large Data Bases, Chile, 1994, pp. 487 – 499. (link)

3. A. Savasere, E. Omiecinski, and S. B. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases“, VLDB '95 Proceedings of the 21th International Conference on Very Large Data Bases, Switzerland, 1995, pp. 432– 444. (link)