

Compact Structures

Zhihong Chong

<http://cse.seu.edu.cn/people/zhchong/>

WebDB.cn Open Group, Southeast University

• Why Advanced Data Structures?

Content

- Week1: Data Structure and Computer Science, Algorithm, Database, WWW...
- Principles of Data Structures—How?
 - Week2: Why and How $O(\log n)$ access time?
 - Week3: Dynamic data structures and Analysis
 - Week4: Randomized Data Structure
 - Week5: Augmented Data Structure
 - Week6: Data Structures in Distributed Environments
- New Problems—Applications
 - Week7: Data Structures in Frontiers of Research
 - Week8: Exam

Outline Huffman Coding

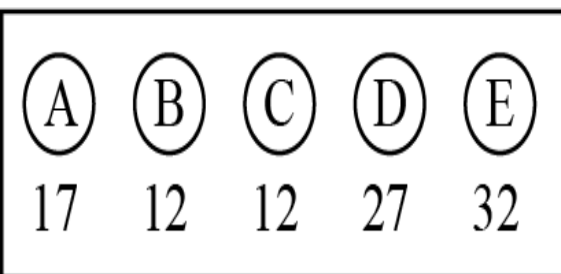
- FP-Tree
- Sketches

Huffman coding

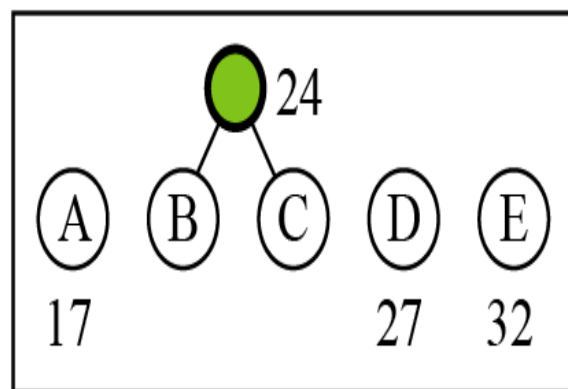
Huffman coding assigns shorter codes to symbols that occur more frequently and longer codes to those that occur less frequently. For example, imagine we have a text file that uses only five characters (A, B, C, D, E). Before we can assign bit patterns to each character, we assign each character a weight based on its frequency of use. In this example, assume that the frequency of the characters is as shown in Table 15.1.

Table 15.1 Frequency of characters

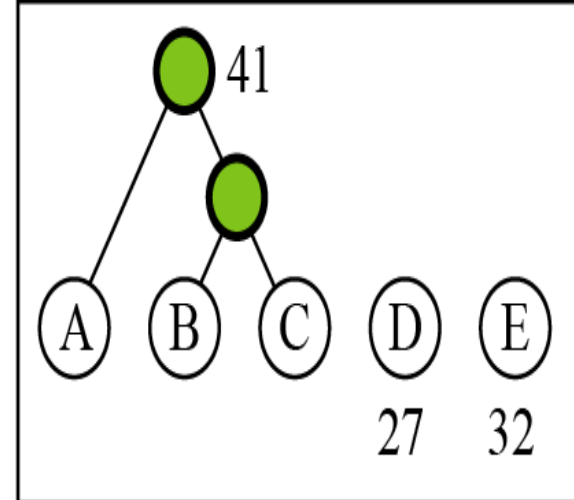
Character	A	B	C	D	E
Frequency	17	12	12	27	32



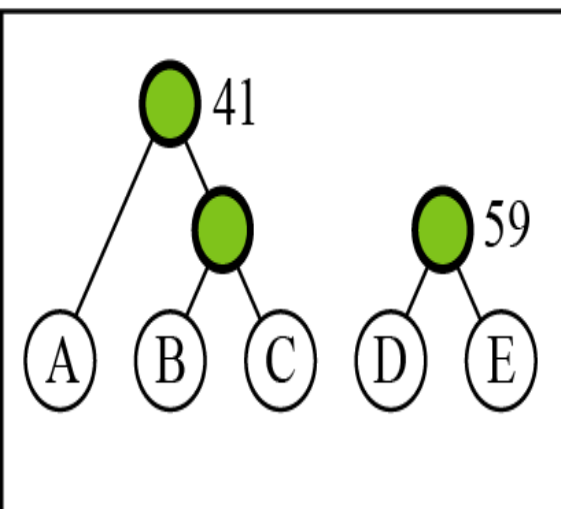
a.



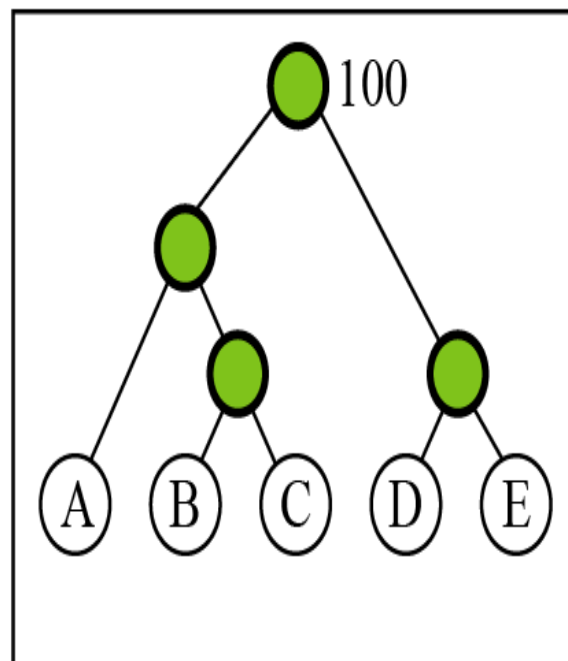
b.



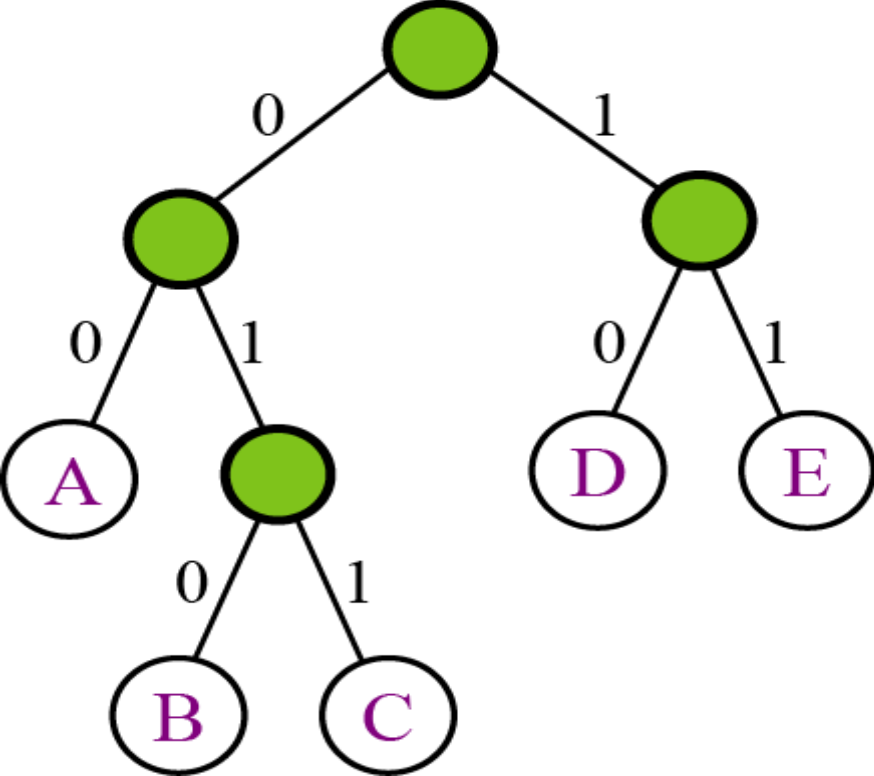
c.



d.



e.



A: 00	D: 10
B: 010	E: 11
C: 011	

Code

- A character's code is found by starting at the root and following the branches that lead to that character. The code itself is the bit value of each branch on the path, taken in sequence.

Text

EAEBAECDEA

Encoder

A	→	00	D	→	10
B	→	010	E	→	11
C	→	011			

1100110100011011101100

Huffman code

Discover the power of custom
layouts

Huffman code

1100110100011011101100

00	→	A	010	→	B
10	→	D	011	→	C
11	→	E			

Decoder

EAEBAECDEA

Text

Discover the power of custom
layouts

Outline

- Frequent Pattern Mining: Problem statement and an example
- Review of Apriori-like Approaches
- FP-Growth:
 - Overview
 - FP-tree:
 - structure, construction and advantages
 - FP-growth:
 - FP-tree → conditional pattern bases → conditional FP-tree
→ frequent patterns
- Experiments
- Discussion:
 - Improvement of FP-growth
- Conclusion Remarks

Frequent Pattern Mining: An Example

Given a transaction database DB and a minimum support threshold ξ , find all frequent patterns (item sets) with support no less than ξ .

Input:	DB:	<u>TID</u>	<u>Items bought</u>
		100	{f, a, c, d, g, i, m, p}
		200	{a, b, c, f, l, m, o}
		300	{b, f, h, j, o}
		400	{b, c, k, s, p}
		500	{a, f, c, e, l, p, m, n}

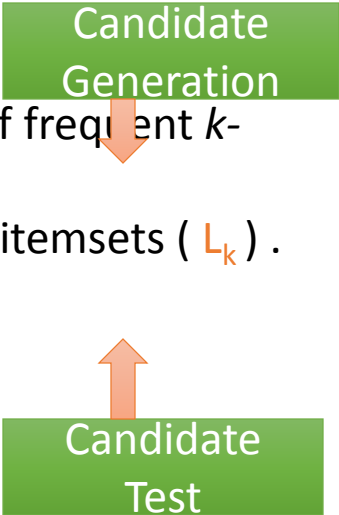
Minimum support: $\xi = 3$

Output: all frequent patterns, i.e., *f, a, ..., fa, fac, fam, fm, am...*

Problem Statement: How to **efficiently** find all frequent patterns?

Apriori

- Main Steps of Apriori Algorithm:
 - Use frequent $(k - 1)$ -itemsets (L_{k-1}) to generate **candidates** of frequent k -itemsets C_k
 - Scan database and count each pattern in C_k , get frequent k -itemsets (L_k) .
- E.g. ,



<i>TID</i>	<i>Items bought</i>	<i>Apriori iteration</i>	
100	{f, a, c, d, g, i, m, p}	C1	f,a,c,d,g,i,m,p,l,o,h,j,k,s,b,e,n
200	{a, b, c, f, l, m, o}	L1	f, a, c, m, b, p
300	{b, f, h, j, o}	C2	fa, fc, fm, fp, ac, am, ...bp
400	{b, c, k, s, p}	L2	fa, fc, fm, ...
500	{a, f, c, e, l, p, m, n}	...	

Performance Bottlenecks of Apriori

- Bottlenecks of *Apriori*: **candidate generation**
 - Generate huge candidate sets:
 - 10^4 frequent 1-itemset will generate 10^7 candidate 2-itemsets
 - To discover a frequent pattern of size 100, e.g., $\{a_1, a_2, \dots, a_{100}\}$, one needs to generate $2^{100} \approx 10^{30}$ candidates.
 - **Candidate Test** incur multiple scans of database: each candidate

Overview of FP-Growth: Ideas

- Compress a large database into a compact, *Frequent-Pattern tree* (FP-tree) structure
 - highly compacted, but complete for frequent pattern mining
 - avoid costly repeated database scans
 - Develop an efficient, FP-tree-based frequent pattern mining method (FP-growth)
 - A divide-and-conquer methodology: decompose mining tasks into smaller ones
 - Avoid candidate generation: sub-database test only.
-

FP-tree: Construction and Design

Construct FP-tree

Two Steps:

1. Scan the transaction DB for the first time, find frequent items (single item patterns) and order them into a list **L** in frequency descending order.

e.g., **L**={f:4, c:4, a:3, b:3, m:3, p:3}

In the format of (item-name, support)

2. For each transaction, order its frequent items according to the order in **L**; Scan DB the second time, construct FP-tree by putting each **frequency ordered transaction** onto it.
-

FP-tree Example: step 1

Step 1: Scan DB for the first time to generate **L**

<i>TID</i>	<i>Items bought</i>
100	{ <i>f, a, c, d, g, i, m, p</i> }
200	{ <i>a, b, c, f, l, m, o</i> }
300	{ <i>b, f, h, j, o</i> }
400	{ <i>b, c, k, s, p</i> }
500	{ <i>a, f, c, e, l, p, m, n</i> }



<i>Item</i>	<i>frequency</i>
<i>f</i>	4
<i>c</i>	4
<i>a</i>	3
<i>b</i>	3
<i>m</i>	3
<i>p</i>	3



By-Product of First Scan
of Database

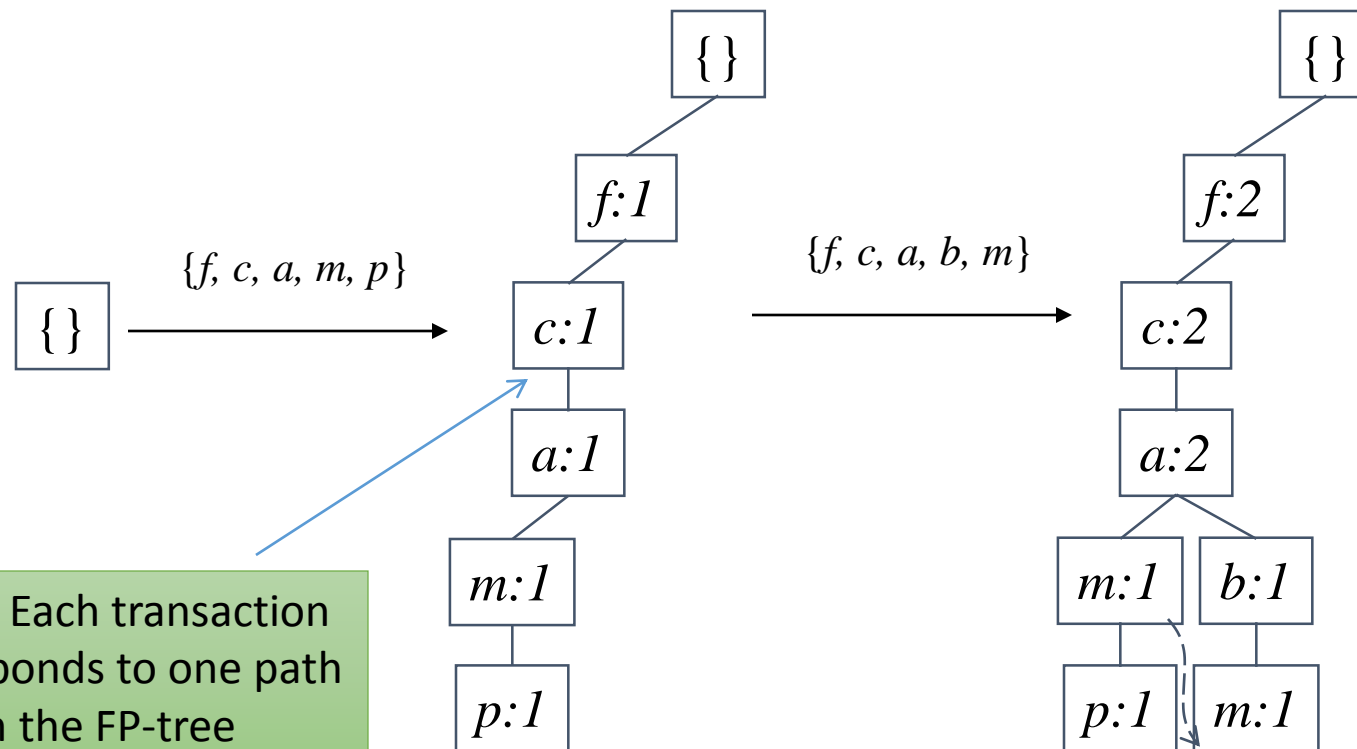
FP-tree Example: step 2

Step 2: scan the DB for the second time, order frequent items in each transaction

<i>TID</i>	<i>Items bought</i>	<i>(ordered) frequent items</i>
100	{f, a, c, d, g, i, m, p}	{f, c, a, m, p}
200	{a, b, c, f, l, m, o}	{f, c, a, b, m}
300	{b, f, h, j, o}	{f, b}
400	{b, c, k, s, p}	{c, b, p}
500	{a, f, c, e, l, p, m, n}	{f, c, a, m, p}

FP-tree Example: step 2

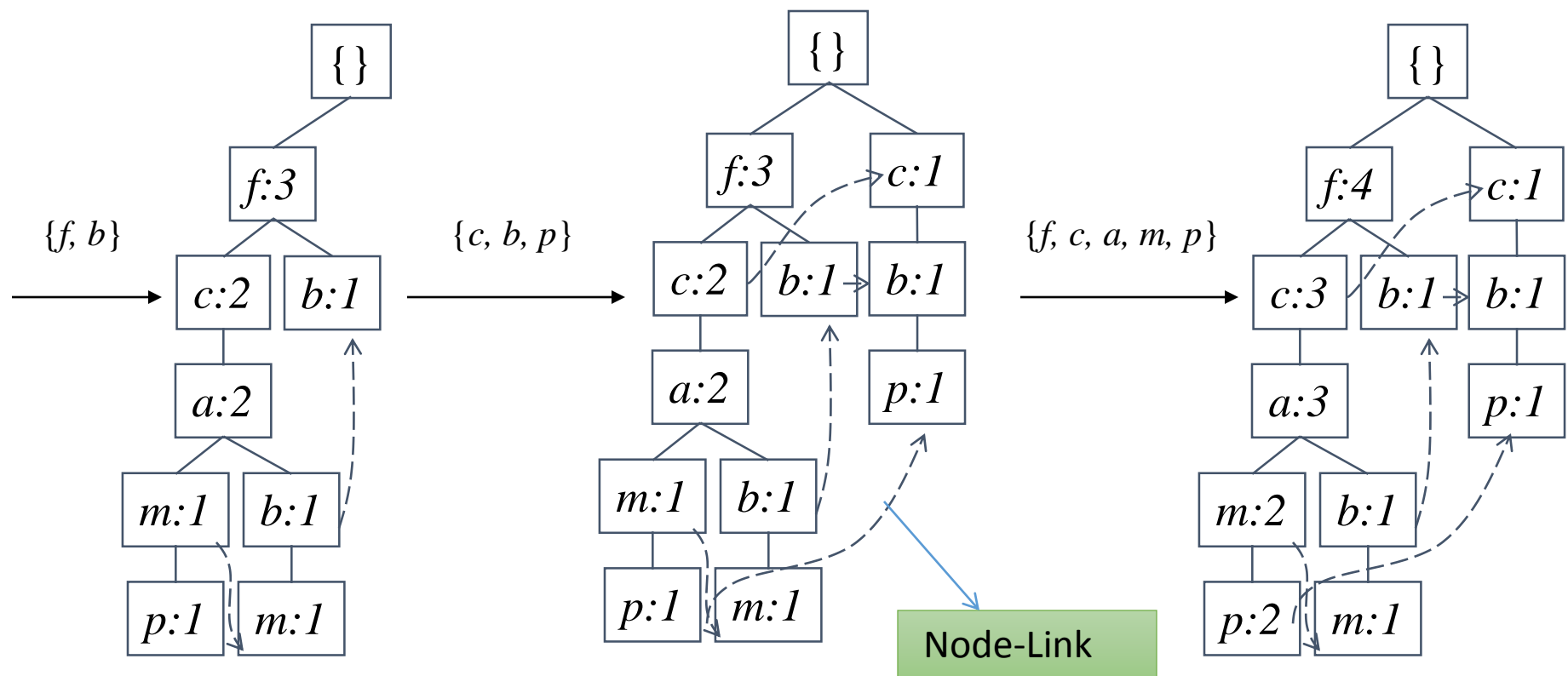
Step 2: construct FP-tree



NOTE: Each transaction corresponds to one path in the FP-tree

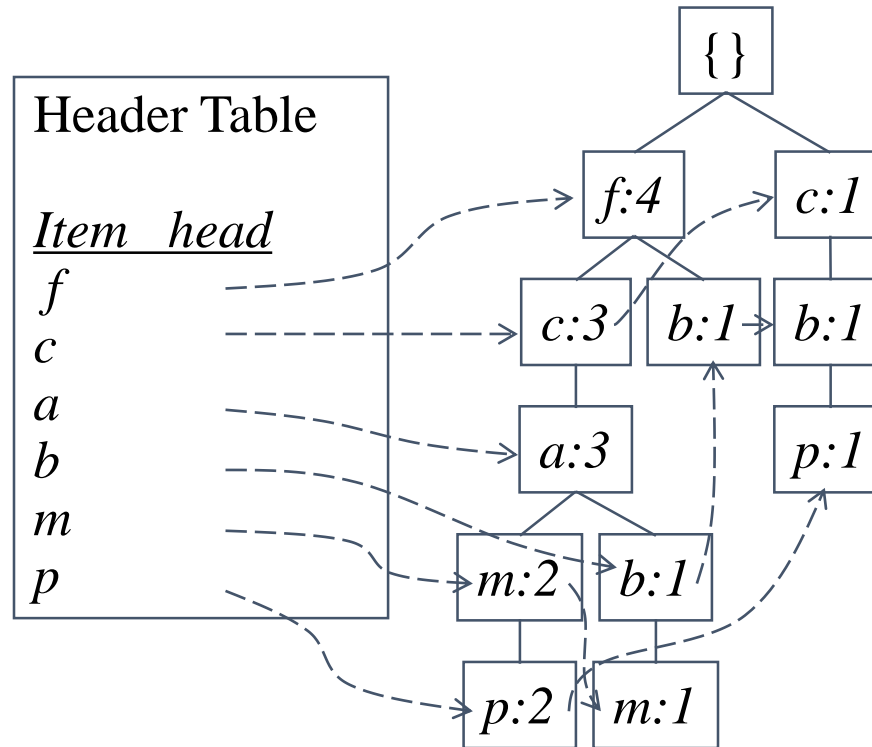
FP-tree Example: step 2

Step 2: construct FP-tree



Construction Example

Final FP-tree



FP-Tree Definition

- **FP-tree is a frequent pattern tree** . Formally, FP-tree is a tree structure defined below:
 1. One **root** labeled as “null”, a set of *item prefix sub-trees* as the children of the root, and a *frequent-item header table*.
 2. Each **node** in *the item prefix sub-trees* has three fields:
 - **item-name** : register which item this node represents,
 - **count**, the number of transactions represented by the portion of the path reaching this node,
 - **node-link** that links to the next node in the FP-tree carrying the same item-name, or null if there is none.
 3. Each **entry** in the *frequent-item header table* has two fields,
 - **item-name**, and
 - **head of node-link** that points to the first node in the FP-tree carrying the item-name.
-

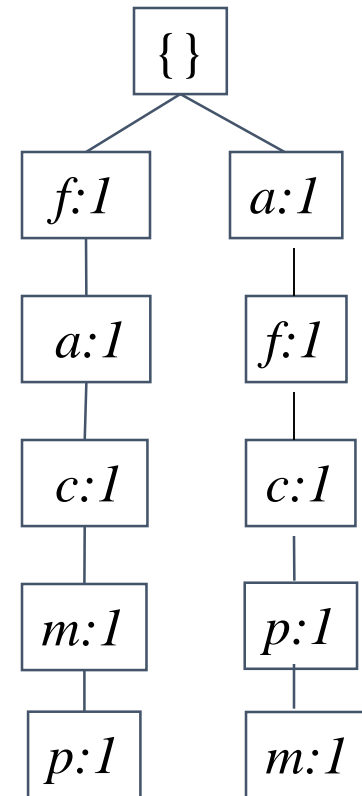
Advantages of the FP-tree Structure

- The most significant advantage of the FP-tree
 - Scan the DB only twice and twice only.
- Completeness:
 - the FP-tree contains all the information related to mining frequent patterns (given the min-support threshold). Why?
- Compactness:
 - The size of the tree is bounded by the occurrences of frequent items
 - The height of the tree is bounded by the maximum number of items in a transaction

Questions?

- Why descending order?
- Example 1:

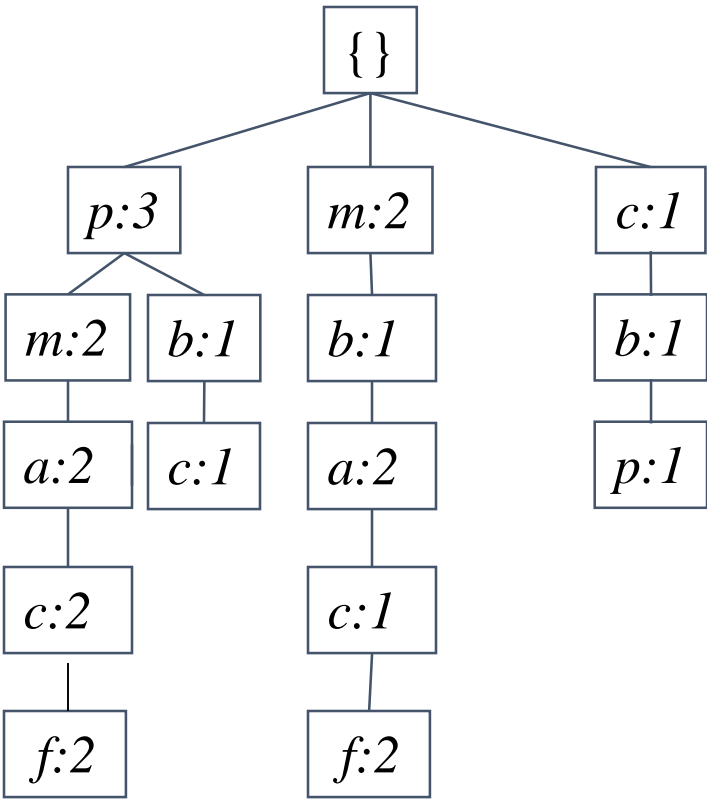
<i>TID</i>	<i>(unordered) frequent items</i>
100	{ <i>f, a, c, m, p</i> }
500	{ <i>a, f, c, p, m</i> }



Questions?

- Example 2:

<i>TID</i>	<i>(ascended) frequent items</i>
100	{ <i>p</i> , <i>m</i> , <i>a</i> , <i>c</i> , <i>f</i> }
200	{ <i>m</i> , <i>b</i> , <i>a</i> , <i>c</i> , <i>f</i> }
300	{ <i>b</i> , <i>f</i> }
400	{ <i>p</i> , <i>b</i> , <i>c</i> }
500	{ <i>p</i> , <i>m</i> , <i>a</i> , <i>c</i> , <i>f</i> }



This tree is larger than FP-tree, because in FP-tree, more frequent items have a higher position, which makes branches less

FP-growth:
Mining Frequent Patterns
Using FP-tree

Mining Frequent Patterns Using FP-tree

- General idea (divide-and-conquer)

Recursively grow frequent patterns using the FP-tree: looking for shorter ones recursively and then concatenating the suffix:

- For each frequent item, construct its **conditional pattern base**, and then its **conditional FP-tree**;
- Repeat the process on each newly created conditional FP-tree until the resulting FP-tree is empty, or it contains only one path (single path will generate all the combinations of its sub-paths, each of which is a frequent pattern)

3 Major Steps

Starting the processing from the end of list **L**:

Step 1:

Construct **conditional pattern base** for each item in the header table

Step 2

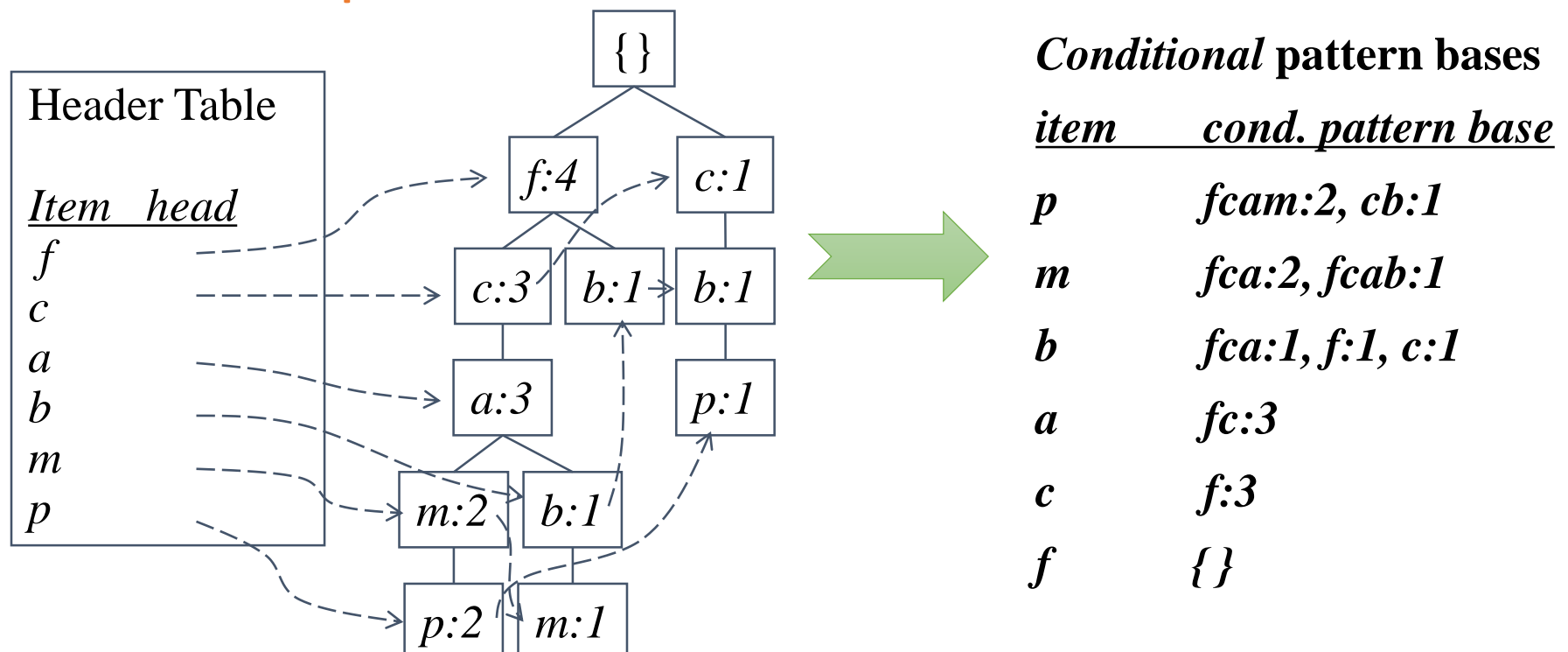
Construct **conditional FP-tree** from each conditional pattern base

Step 3

Recursively mine conditional FP-trees and grow frequent patterns obtained so far. If the conditional FP-tree contains a **single path**, simply enumerate all the patterns

Step 1: Construct Conditional Pattern Base

- Starting at the bottom of frequent-item header table in the FP-tree
- Traverse the FP-tree by following the link of each frequent item
- Accumulate all of **transformed prefix paths** of that item to form a **conditional pattern base**

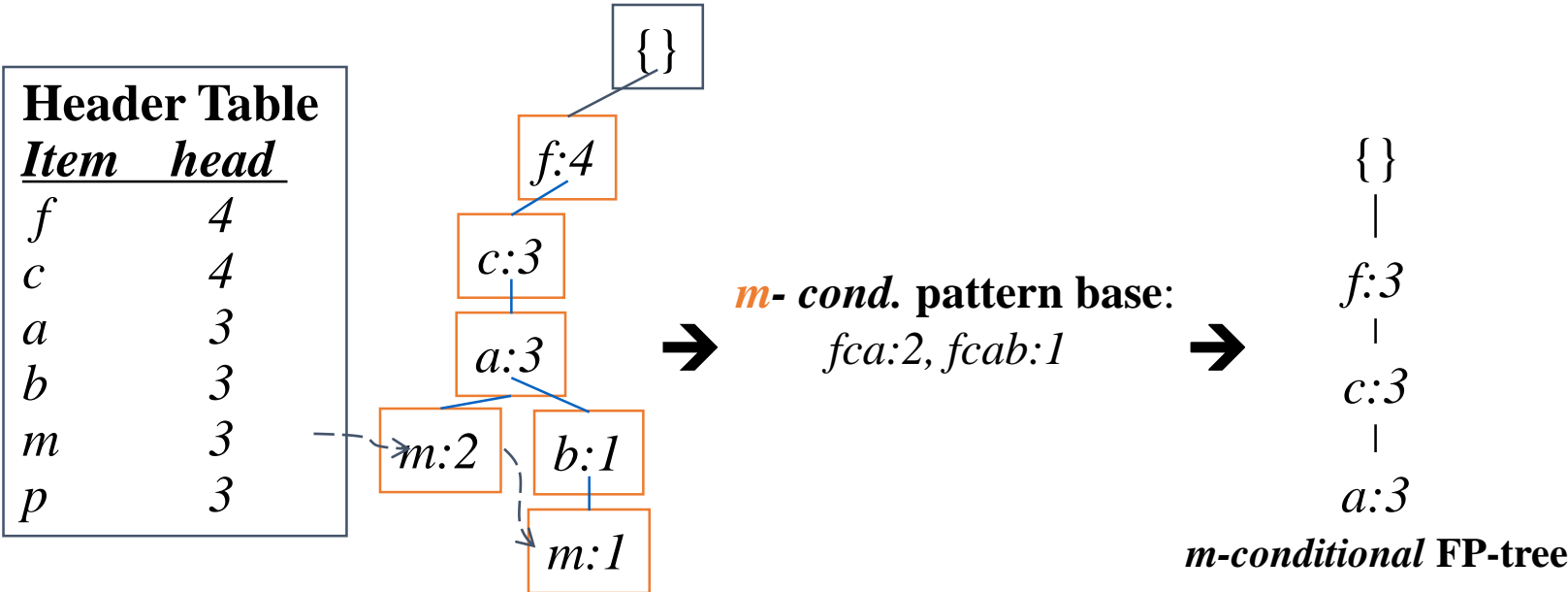


Properties of FP-Tree

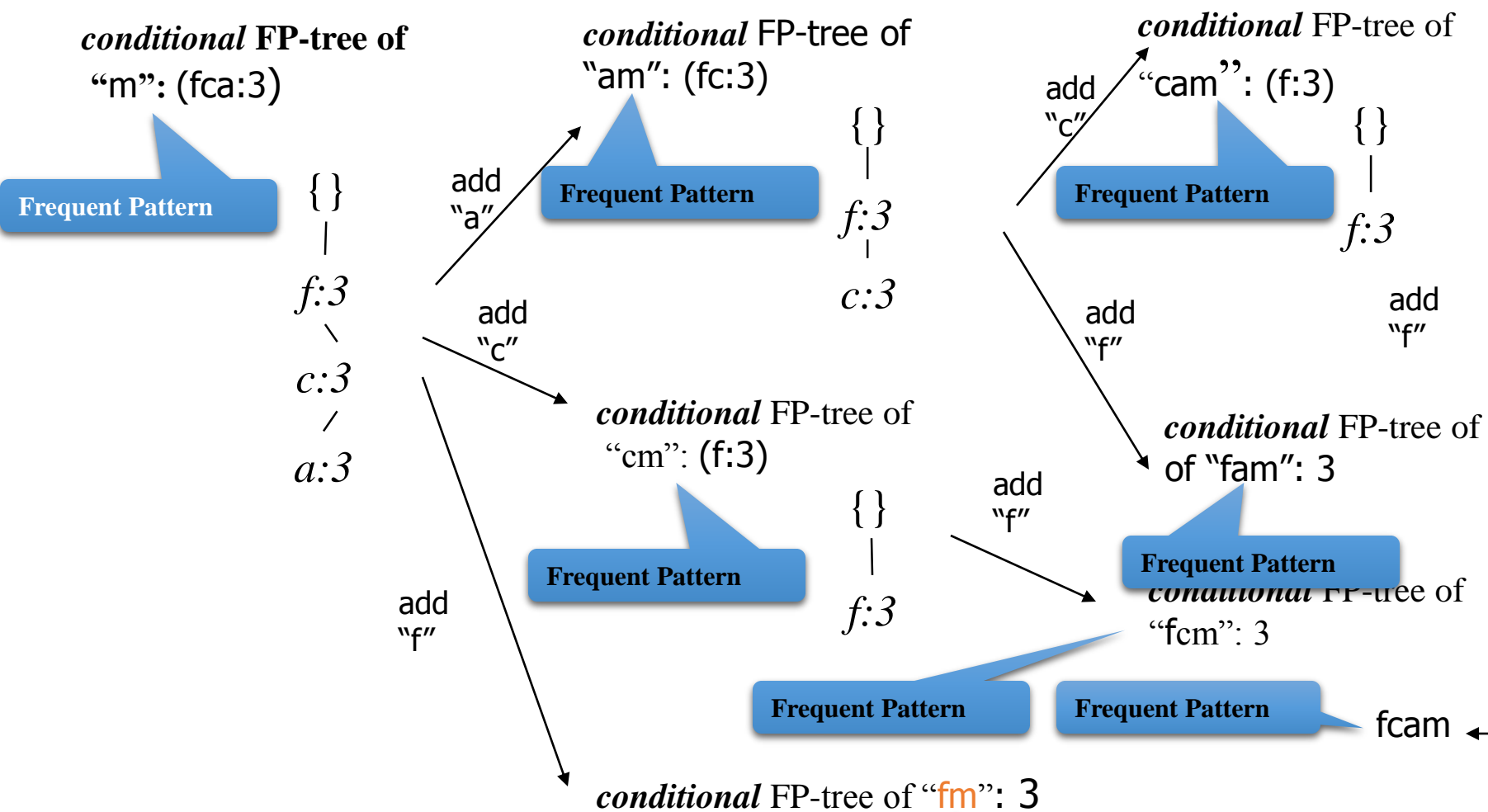
- Node-link property
 - For any frequent item a_i , all the possible frequent patterns that contain a_i can be obtained by following a_i 's node-links, starting from a_i 's head in the FP-tree header.
- Prefix path property
 - To calculate the frequent patterns for a node a_i in a path P , only the prefix sub-path of a_i in P need to be accumulated, and its frequency count should carry the same count as node a_i .

Step 2: Construct Conditional FP-tree

- For each pattern base
 - Accumulate the count for each item in the base
 - Construct the conditional FP-tree for the frequent items of the pattern base




Step 3: Recursively mine the conditional FP-tree



Principles of FP-Growth

- Pattern growth property
 - Let α be a frequent itemset in DB, B be α 's conditional pattern base, and β be an itemset in B. Then $\alpha \cup \beta$ is a frequent itemset in DB iff β is frequent in B.
 - Is “*fcabm*” a frequent pattern?
 - “fcab” is a branch of m's conditional pattern base
 - “b” is **NOT** frequent in transactions containing “fcab”
 - “bm” is **NOT** a frequent itemset.
-

Conditional Pattern Bases and Conditional FP-Tree

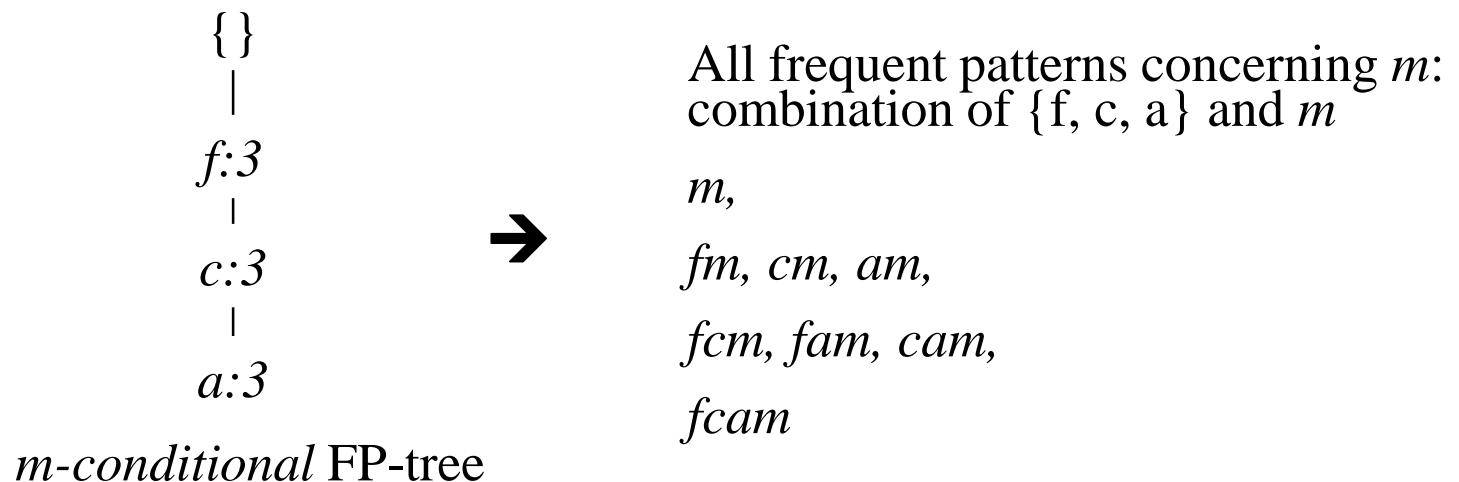


Item	Conditional pattern base	Conditional FP-tree
p	{(fcam:2), (cb:1)}	{(c:3)} p
m	{(fca:2), (fcab:1)}	{(f:3, c:3, a:3)} m
b	{(fca:1), (f:1), (c:1)}	Empty
a	{(fc:3)}	{(f:3, c:3)} a
c	{(f:3)}	{(f:3)} c
f	Empty	Empty

order of L

Single FP-tree Path Generation

- Suppose an FP-tree T has a single path P . The complete set of frequent pattern of T can be generated by enumeration of all the combinations of the sub-paths of P



Summary of FP-Growth Algorithm

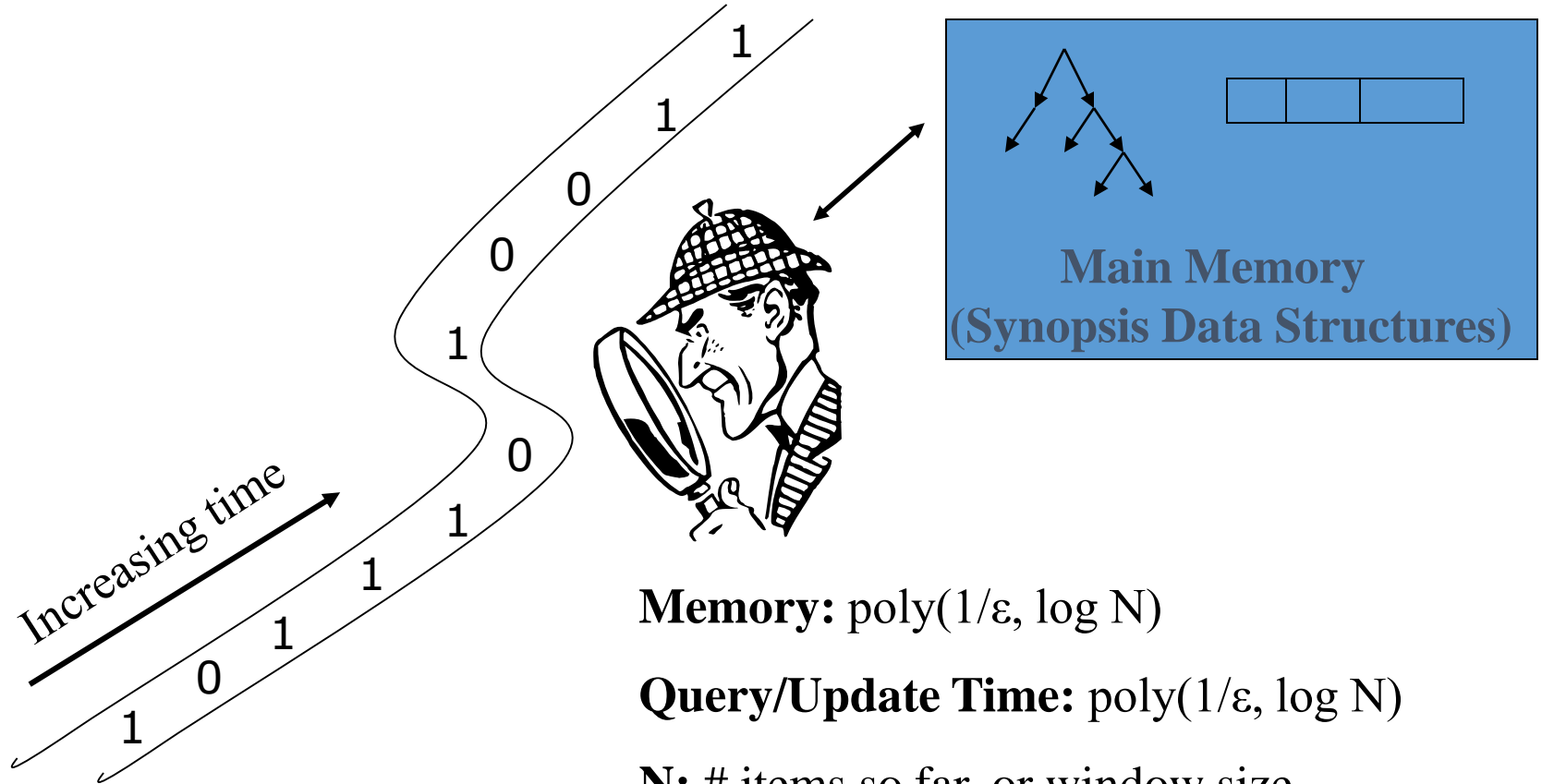
- Mining frequent patterns can be viewed as first mining 1-itemset and progressively growing each 1-itemset by mining on its conditional pattern base recursively
- Transform a frequent k-itemset mining problem into a sequence of k frequent 1-itemset mining problems via a set of conditional pattern bases

Efficiency Analysis

Facts: usually

1. FP-tree is much smaller than the size of the DB
 2. Pattern base is smaller than original FP-tree
 3. Conditional FP-tree is smaller than pattern base
- ➔ mining process works on a set of usually much smaller pattern bases and conditional FP-trees
 - ➔ Divide-and-conquer and dramatic scale of shrinking

Stream Model of Computation



Data Stream

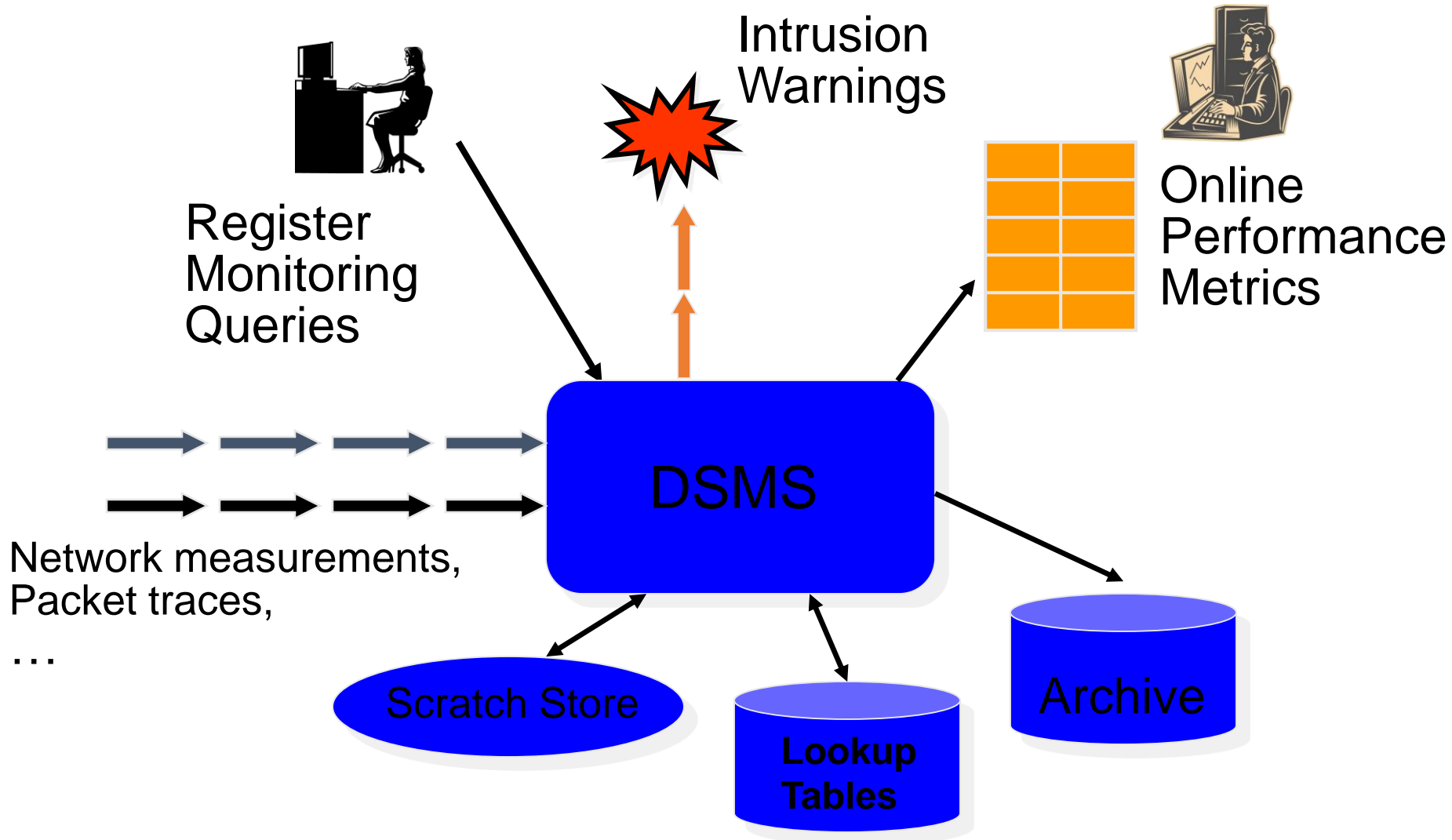
Memory: $\text{poly}(1/\epsilon, \log N)$

Query/Update Time: $\text{poly}(1/\epsilon, \log N)$

N: # items so far, or window size

ε : error parameter

“Toy” Example – Network Monitoring

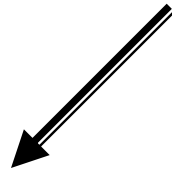
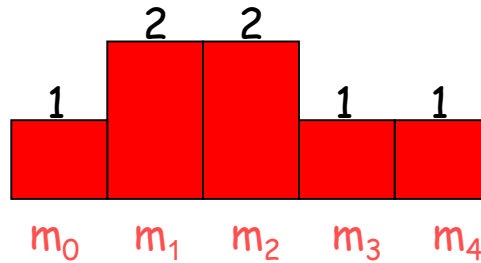


Generalized Stream Model

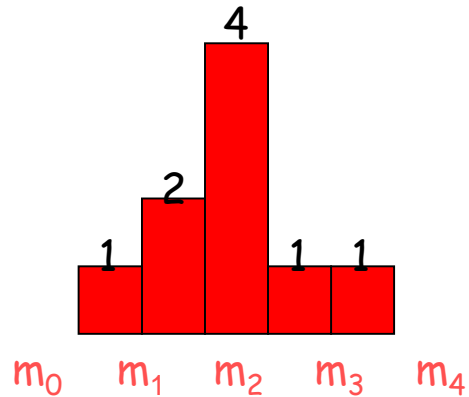


- Input Element (i,a)
 - a copies of domain-value i
 - increment to ith dimension of **m** by a
 - a need not be an integer
- Negative value – captures deletions

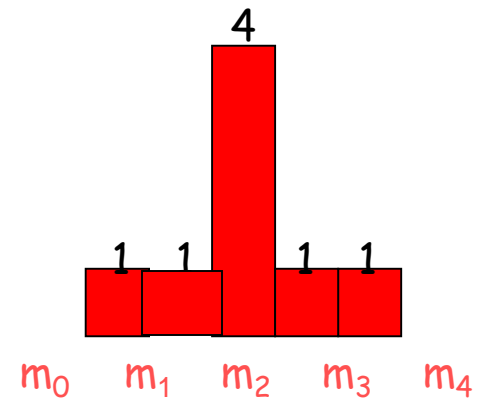
Example



On seeing element $(i,a) = (2,2)$



On seeing element $(i,a) = (1,-1)$



Problem 1 – Top-K List

[Charikar-Chen-Farach-Colton]

The Google Problem

Return **list** of **k** most frequent items in stream

Motivation

search engine queries, network traffic, ...

Remember

Saw **lower bound** recently!

Solution

Data structure **Count-Sketch** → maintaining count-estimates of high-frequency elements

• Notation

Definitions

- Assume $\{1, 2, \dots, N\}$ in order of frequency
- m_i is frequency of i^{th} most frequent element
- $m = \sum m_i$ is number of elements in stream

• FindCandidateTop

- **Input:** stream S , int k , int p
- **Output:** list of p elements containing top k
- Naive sampling gives solution with $p = \Omega(m \log k / m_k)$

• FindApproxTop

- **Input:** stream S , int k , real ϵ
- **Output:** list of k elements, each of frequency $m_i > (1-\epsilon) m_k$
- Naive sampling gives no solution

Main Idea

- Consider
 - single counter X
 - hash function $h(i): \{1, 2, \dots, N\} \rightarrow \{-1, +1\}$
- Input element $i \rightarrow$ update counter $X += Z_i = h(i)$
- For each r , use XZ_r as estimator of m_r

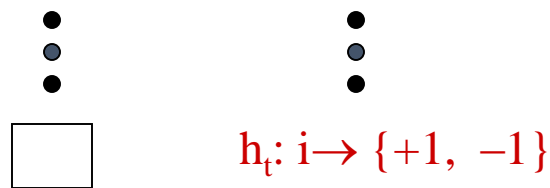
Theorem: $E[XZ_r] = m_r$

Proof

- $X = \sum_i m_i Z_i$
- $E[XZ_r] = E[\sum_i m_i Z_i Z_r] = \sum_i m_i E[Z_i Z_r] = m_r E[Z_r^2] = m_r$
- Cross-terms cancel

Finding Max Frequency Element

- Problem – $\text{var}[X] = F_2 = \sum_i m_i^2$
- Idea – t counters, independent 4-wise hashes h_1, \dots, h_t



- Use $t = O(\log m \cdot \sum m_i^2 / (\epsilon m_1)^2)$
- **Claim:** New Variance $< \sum m_i^2 / t = (\epsilon m_1)^2 / \log m$
- Overall Estimator
 - repeat + median of averages
 - with high probability, approximate m_1

Problem with “Array of Counters”

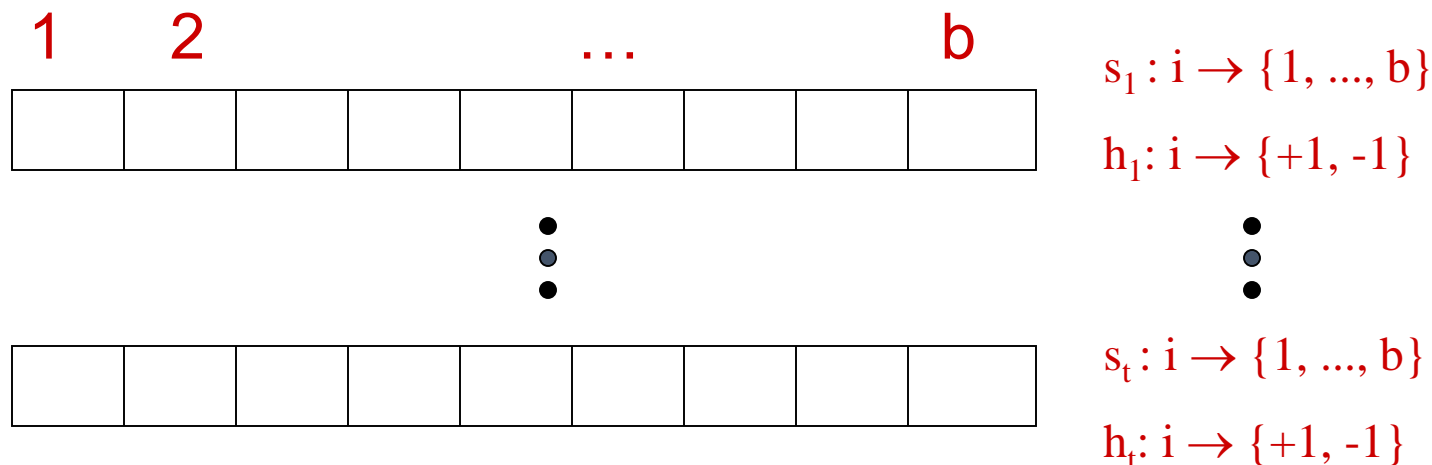
- Variance – dominated by highest frequency
- Estimates for less-frequent elements like k
 - corrupted by higher frequencies
 - variance $\gg m_k$
- Avoiding Collisions?
 - spread out high frequency elements
 - replace each counter with hashtable of b counters

Count Sketch

- Hash Functions

- 4-wise independent hashes h_1, \dots, h_t and s_1, \dots, s_t
- hashes independent of each other

- Data structure: hashtables of counters $X(r, c)$



Overall Algorithm

- $s_r(i)$ — one of b counters in r th hashtable
- Input $i \rightarrow$ for each r , update $X(r, s_r(i)) += h_r(i)$
- Estimator(m_i) = $\text{median}_r \{ X(r, s_r(i)) \cdot h_r(i) \}$
- Maintain heap of k top elements seen so far
- Observe
 - Not completely eliminated collision with high frequency items
 - Few of estimates $X(r, s_r(i)) \cdot h_r(i)$ could have high variance
 - Median not sensitive to these poor estimates

Avoiding Large Items

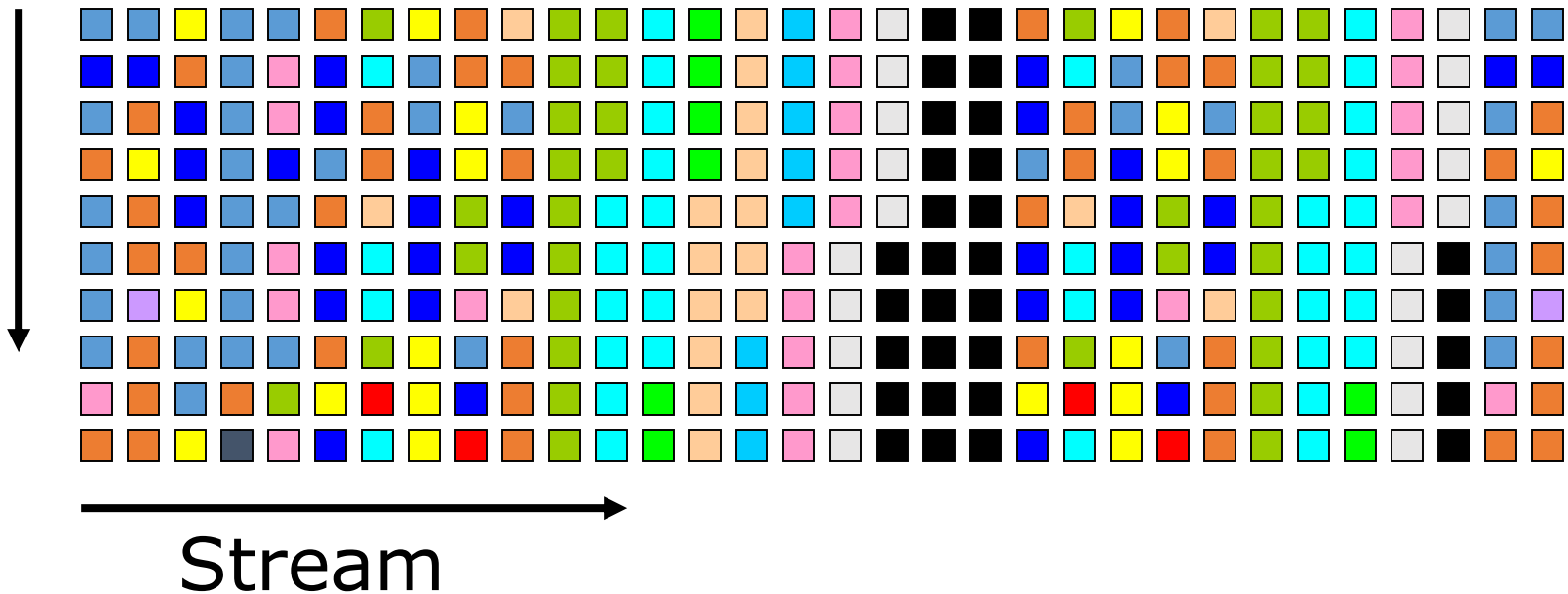
- $b > O(k) \rightarrow$ with probability $\Omega(1)$, no collision with top- k elements
- t hashtables represent independent trials
- Need $\log m/\delta$ trials to estimate with probability $1-\delta$
- Also need – small variance for colliding small elements
- **Claim:**
 $P[\text{variance due to small items in each estimate} < (\sum_{i>k} m_i^2)/b] = \Omega(1)$
- **Final bound** $b = O(k + \sum_{i>k} m_i^2 / (\epsilon m_k)^2)$

Final Results

- Zipfian Distribution: $m_i \propto 1/i^\alpha$ [Power Law]
- FindApproxTop
 - $[k + (\sum_{i>k} m_i^2) / (\epsilon m_k)^2] \log m / \delta$
 - Roughly: sampling bound with frequencies squared
 - Zipfian – gives improved results
- FindCandidateTop
 - Zipf parameter 0.5
 - $O(k \log N \log m)$
 - Compare: sampling bound $O((kN)^{0.5} \log k)$

Problem 2 – Elephants-and-Ants

[Manku-Motwani]

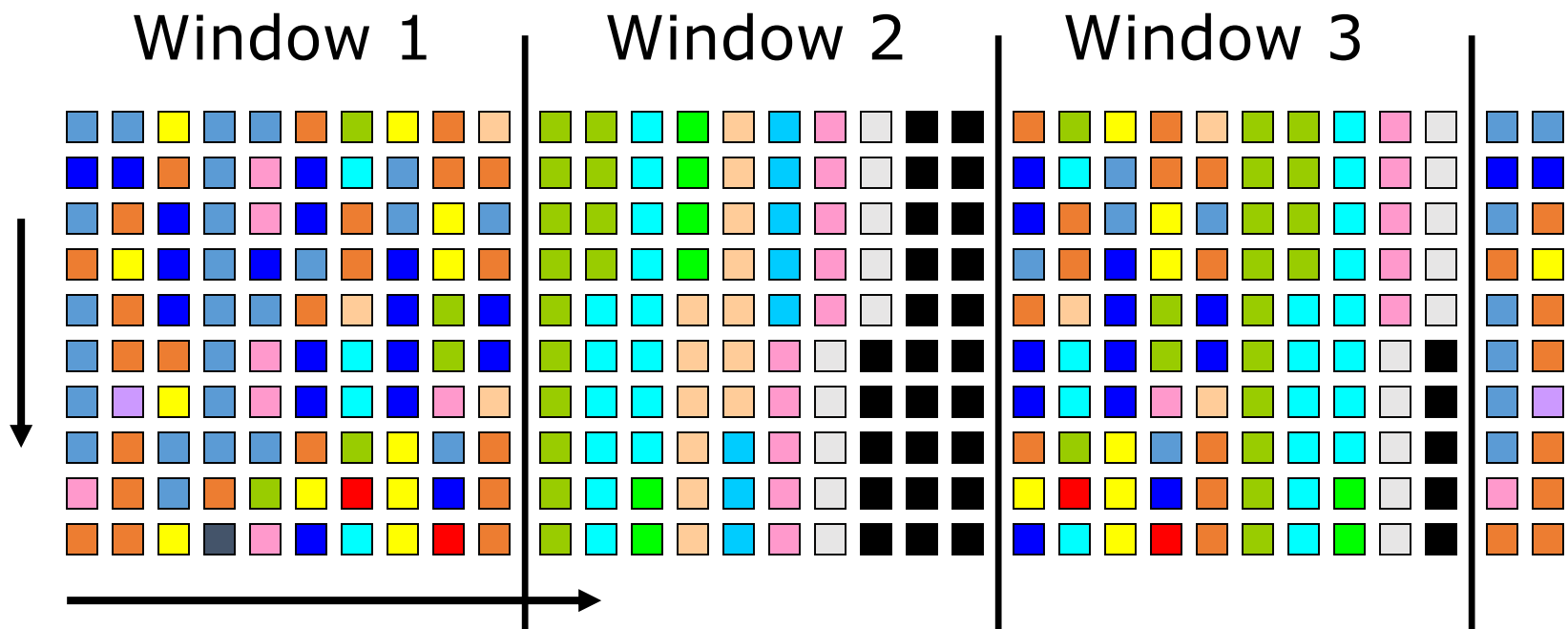


- Identify items whose current frequency exceeds support threshold $s = 0.1\%$.

[Jacobson 2000, Estan-Verghese 2001]

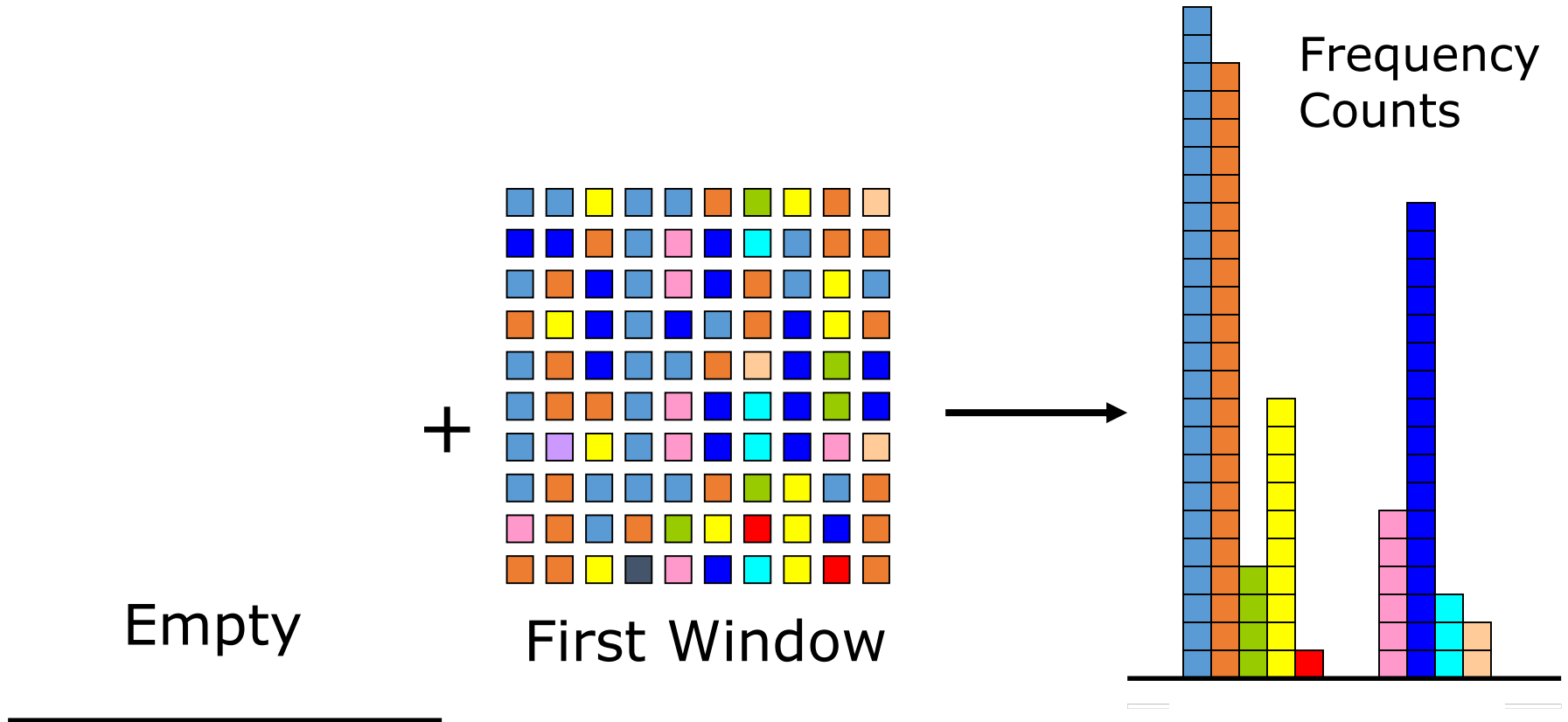
Algorithm 1: Lossy Counting

Step 1: Divide the stream into 'windows'



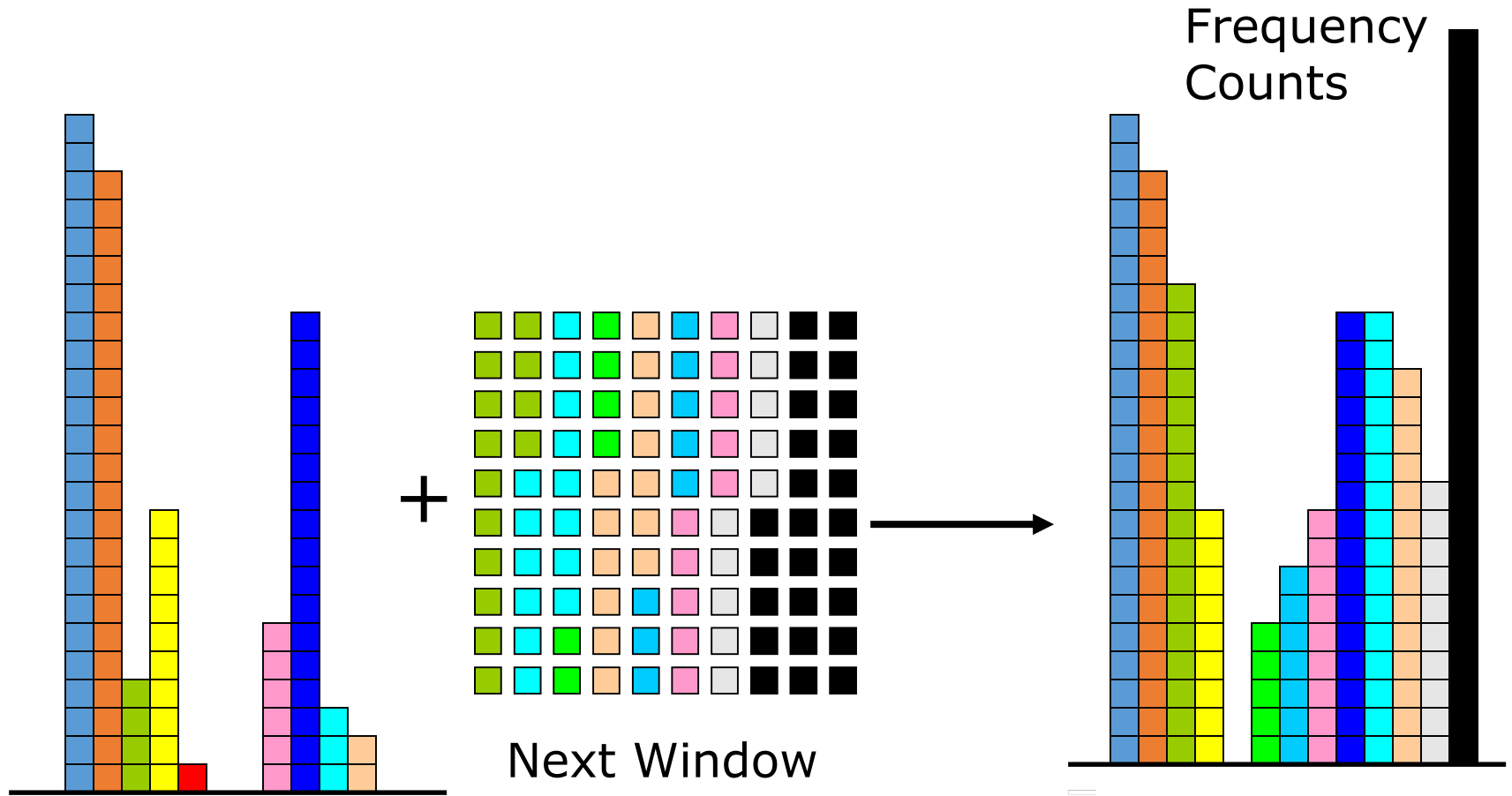
Window-size w is function of support s – specify later...

Lossy Counting in Action ...



At window boundary, decrement all counters by 1

Lossy Counting (continued)



At window boundary, decrement all counters by 1

Error Analysis

How much do we undercount?

If current size of stream = N
and window-size w = $1/\epsilon$

then frequency error $\leq \# \text{ windows} = \epsilon N$

Rule of thumb:

Set $\epsilon = 10\%$ of support s

Example:

Given support frequency $s = 1\%$,
set error frequency $\epsilon = 0.1\%$

Putting it all together...
Output:

Elements with counter values exceeding $(s-\epsilon)N$

Approximation guarantees

Frequencies underestimated by at most ϵN

No false negatives

False positives have true frequency at least $(s-\epsilon)N$

How many counters do we need?

- Worst case bound: $1/\epsilon \log \epsilon N$ counters
- Implementation details...

Number of Counters?

- Window size $w = 1/\epsilon$
- Number of windows $m = \epsilon N$
- n_i – # counters alive over last i windows
- Fact:

- Claim: $\sum_{i=1}^j n_i \leq jw$ for $j = 1, 2, \dots, m$

- Counter must average 1 increment/window to survive

$$\sum_{i=1}^j n_i \leq \sum_{i=1}^j \frac{w}{i} \quad \text{for } j = 1, 2, \dots, m$$

- **# active counters**

$$\sum_{i=1}^m n_i \leq \sum_{i=1}^m \frac{w}{i} \leq w \log m = \frac{1}{\epsilon} \log \epsilon N$$

Enhancements

Frequency Errors

For counter (X, c) , true frequency in $[c, c + \epsilon N]$

Trick: Track number of windows t counter has been active

For counter (X, c, t) , true frequency in $[c, c + t - 1]$

If $(t = 1)$, no error!

Batch Processing

Decrements after k windows