Knowledge Discovery from Data Streams: Frequent Pattern Mining

João Gama
LIAAD-INESC Porto,
University of Porto, Portugal

Outline

Introduction

- Frequent pattern mining refers to finding patterns that occur greater than a pre-specified threshold value.
- Patterns refer to items, itemsets, or sequences.
- Threshold refers to the percentage of the pattern occurrences to the total number of transactions. It is termed as *Support*.

Introduction

- Finding frequent patterns is the first step for the discovery of association rules in the form of $A \rightarrow B$.
- Apriori algorithm represents a pioneering work for association rules discovery
  R Agrawal and R Srikant, *Fast Algorithms for Mining Association Rules*. VLDB 2004.
- An important step towards improving the performance of association rules discovery was FP-Growth
  J. Han, J. Pei, and Y. Yin. *Mining Frequent Patterns without Candidate Generation* SIGMOD 2000

Introduction

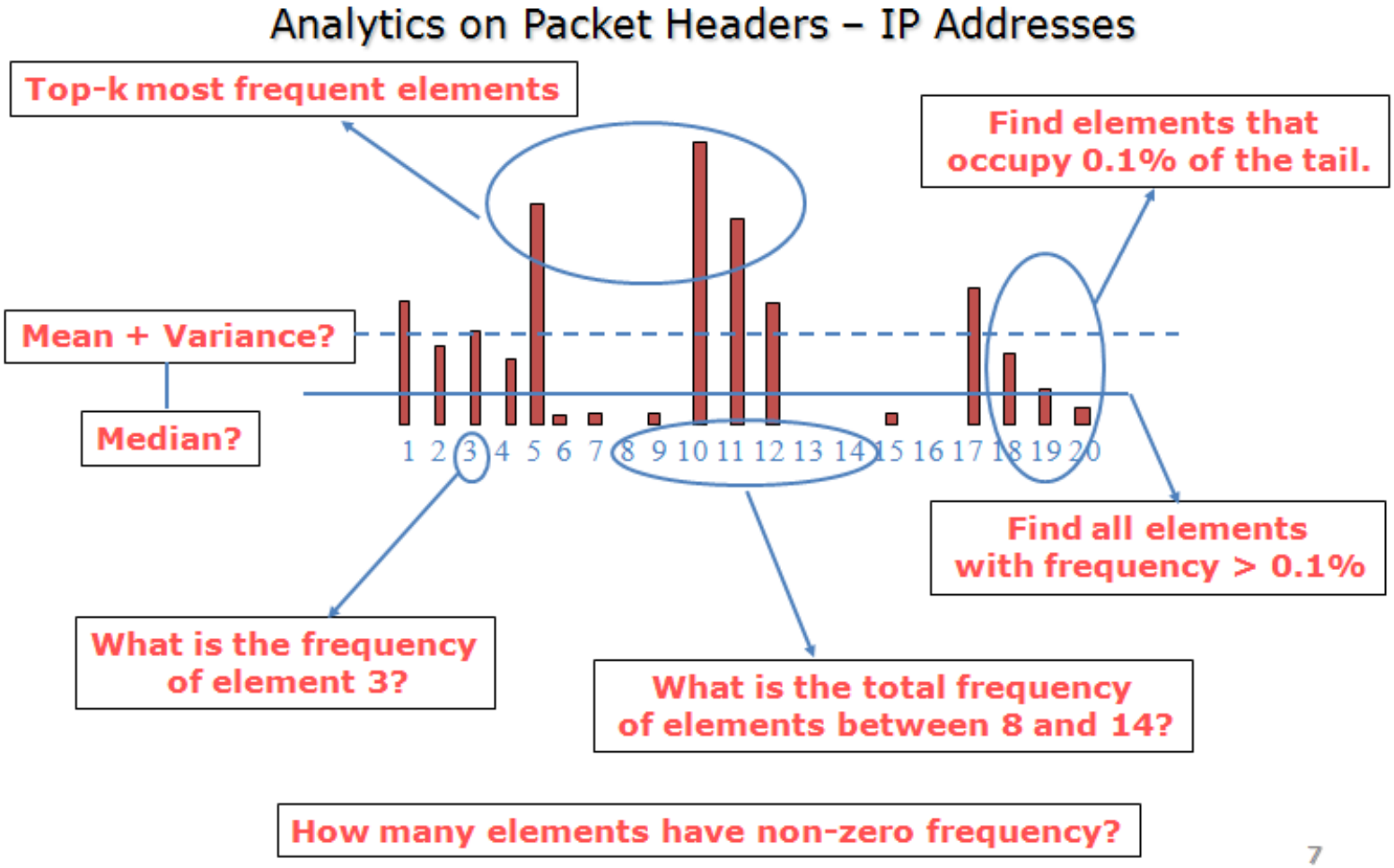
- Many measurements have been proposed for finding the strength of the rules.
- The very frequently used measure is *support*.
  - The support $Supp(X)$ of an itemset X is defined as the proportion of transactions in the data set which contain the itemset.
- Another frequently used measure is *confidence*.
  - Confidence refers to the probability that set B exists given that A already exists in a transaction.
  - Confidence $(A \rightarrow B)$ = Supp (AB) / Supp (A)

Frequent Pattern Mining in Data Streams

The process of frequent pattern mining over data streams differs from the conventional one as follows:

- The technique should be linear or sublinear: **You Have Only One Look**.
- Frequent items, *heavy hitters*, and itemsets are often the final output.

Motivation



Slide from R. Motwani talk.

Outline

Definitions

Given a stream S of m items $\langle e_1, e_2, \ldots e_m \rangle$ the frequency of an item $e \in S$ is $f(e) = |\{e_j \in S : e_j = e\}|$.

- The exact ϕ-frequent items are those with $f(e) > \phi \times m$, with $\phi \leq 1$
- The ϵ-approximate frequent items those with $f(e) > (\phi - \epsilon) \times m$, with $\phi \leq 1$

Tasks

Main tasks:

- Representing sets
- **Frequency estimates for all elements in the stream**:
  Sketch-based techniques: linear projection of the input
  - Count-min sketch
  - FM sketch
- **Top-k items**:
  Counter-based techniques:
  monitor a subset of items
  - The *Frequent* Algorithm
  - The *Space-Saving* Algorithm

Bloom Filters

Bloom, B. (1970), *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM 13 (7)

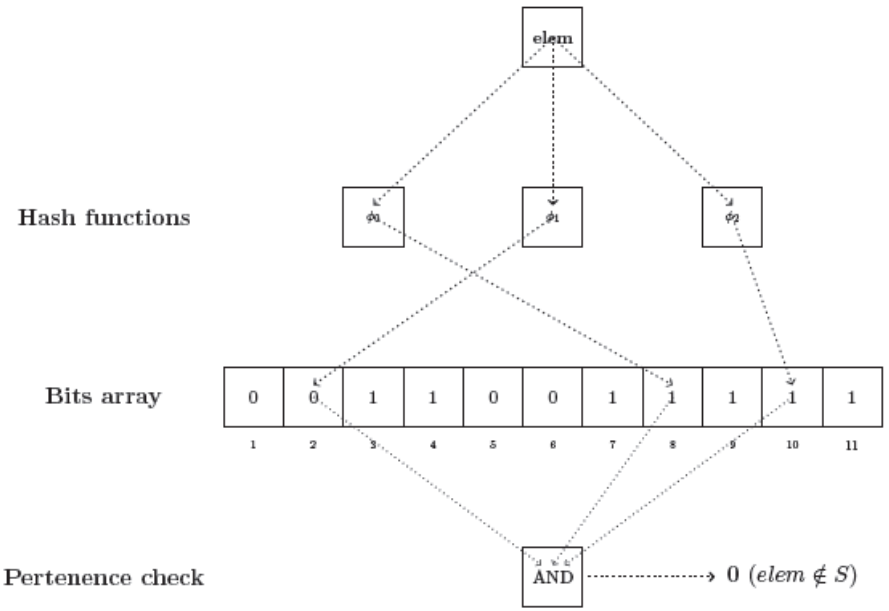
- A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set.
- A query returns either *inside set (may be wrong)* or *definitely not in set*.
- Elements can be added to the set.
- Properties:
  - False positive retrieval results are possible, but false negatives are not;
  - The more elements that are added to the set, the larger the probability of false positives.

Bloom Filters

- **Data Structure**
  - An empty Bloom filter is a bit array of m bits, all set to 0.
  - There must also be k different hash functions defined, each of which maps some element to one of the m array positions with a uniform random distribution.
- **To add** an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.
- **To query** for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions. If any of the bits at these positions are 0, the element is definitely not in the set.

Bloom Filters

A Bloom filter of m bits, and k hash functions



The probability p of false positives is: $ln(p) = -\frac{m}{n} ln(2)^2$, where n is the number of inserted elements.

Illustrative Problem

We manage mobile push notifications for our customers, and one of the things we need to guard against is sending multiple notifications to the same user for the same campaign. Push notifications are routed to individual devices/users based on push notification tokens generated by the mobile platforms. Because of their size (anywhere from 32b to 4kb), its non-performant for us to index push tokens or use them as the primary user key.
On certain mobile platforms, when a user uninstalls and subsequently re-installs the same app, we lose our primary user key and create a new user profile for that device. Typically, in that case, the mobile platform will generate a new push notification token for that user on the reinstall. However, that is not always guaranteed. So, in a small number of cases we can end up with multiple user records in our system having the same push notification token.

As a result, to prevent sending multiple notifications to the same user for the same campaign, we need to filter for a relatively small number of duplicate push tokens from a total dataset that runs from hundreds of millions to billions of records. To give you a sense of proportion, the memory required to filter just 100 Million push tokens is 100M * 256 = 25 GB!

The solution - Bloom filter

- Allocate a bit array of size m. Choose k independent hash functions $h_i(x)$ whose range is $[0 \ldots m-1]$ For each data element, compute hashes and turn on bits Bloom filter For membership query q, apply hashes and check if all the corresponding bits are 'on' Note that bits might be turned 'on' by hash collisions leading to false positives; i.e a non-existing element may be reported to exist and the goal is to minimise this.
- On hash functions
  Hash functions for Bloom filter should be independent and uniformly distributed. Cryptographic hashes like MD5 or SHA1 are not good choices for performance reasons. Some of the suitable fast hashes are MurmurHash, FNV hashes and Jenkin's Hashes.
- We use MurmurHash
  It's fast: 10x faster than MD5
  Good distribution: passes chi-squared test for uniformity
  Avalanche effect: sensitive to even slightest input changes Independent enough

Sizing the Bloom filter

Sizing the bit array involves choosing optimal number of hash functions to minimise false-positive probability.

With m bits, k hash functions and n elements, the false positive probability, i.e the probability of all the corresponding k bits are 'on', falsely when the element does not exist

$p = (1 - [1 - \frac{1}{m}]^{kn})^k \approx (1 - e^{-\frac{kn}{m}})^k$

for given m, n, optimal k that minimises p i.e

$\frac{dp}{dk} = 0 \implies k = \frac{m}{n} ln(2) \implies m = -\frac{nln(p)}{(ln(2))^2}$

so, for 100 Million push tokens with 0.001 error probability

$m = -\frac{100000000 * ln(0.001)}{(ln(2))^2} = 171MB$

This is significant improvement from 25 GB.

The Top-k Elements Problem

Count the top-K most frequent elements in a stream.

First Approach

Maintain a count for each element of the alphabet.
Return the k first elements in the sorted list of counts.

Problems

Exact and Efficient solution for small alphabets.
Large alphabets: Space inefficient – large number of zero counts.

The *Frequent* Algorithm

J. Misra and D. Gries, *Finding repeated elements*. Science of Computer Programming, 1982.

Maintain partial information of interest; monitor only a subset m of elements.

- For each element e in the stream
  - If e is monitored: Increment $Count_e$
  - Else
    - If there is a $Count_j == 0$
      Replace element j by e and initialize $Count_e = 1$
    - Else Subtract 1 to each $Count_i$

The *Space Saving* Algorithm

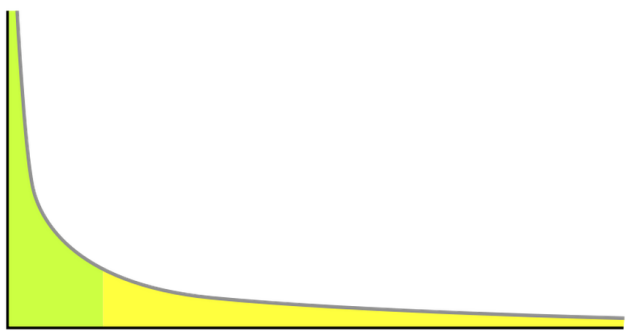
Metwally, D. Agrawal, A. Abbadi, *Efficient Computation of Frequent and Top-k Elements in Data Streams*, ICDT 2005

Maintain partial information of interest; monitor only a subset m of elements.

- For each element e in the stream
  - If e is monitored: Increment $Count_e$
  - Else
    - Let e_m be the element with least hits min.
    - Replace e_m with e with $count_e = min + 1$

The Space Saving Algorithm: Insights

- Efficient for skewed data!
- Ensures no false negatives are kept in the top-k list:
  no non frequent item is in the top-k list.
- It allows false positive in the list:
  some non frequent items appear in the list.
- If the popular elements evolve over time, the elements that are growing more popular will gradually be pushed to the top of the list.

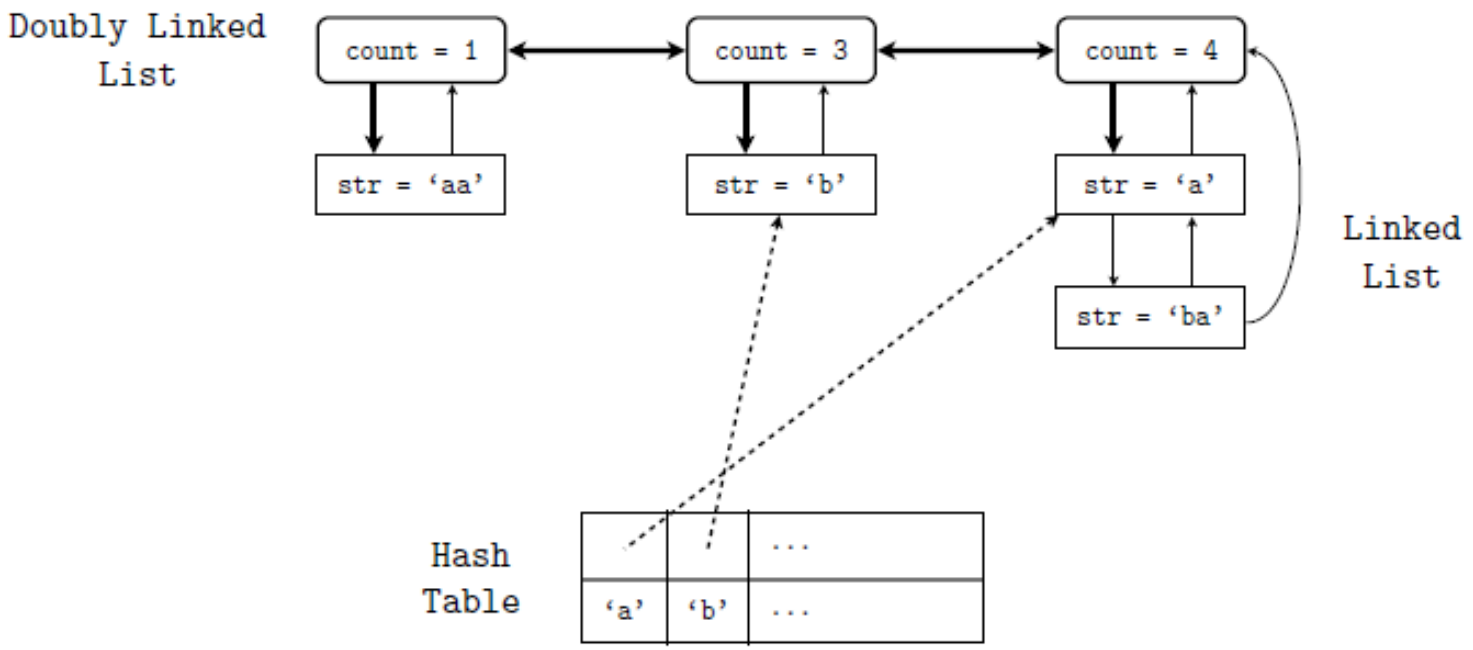


The Space Saving Algorithm: Properties

The retrieval operation has a parameter $\epsilon \geq 1/K$, takes time $O(1/\epsilon)$ and returns at time t a set of at most K pairs of the form $(x; c_x)$.
The key properties of the sketch are:

1. This set is guaranteed to contain every x such that $f(x) \geq \epsilon \times t$;
2. For each $(x; c_x)$ in the set, $0 \leq c_x - f(x) \geq t/K$.

The Space Saving Algorithm: Implementation



Conceptual representation of the Space-Saving data structure.

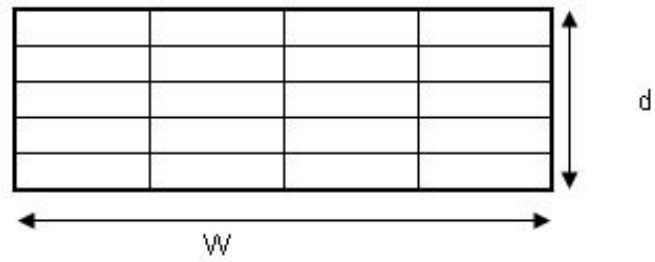
The Count-Min Sketch

Cormode & Muthukrishnan. *An improved data stream summary: The count-min sketch and its applications*. Journal of Algorithms, 2005.

Used to approximately solve: Point Queries, Range Queries, Inner Product queries.

Simple sketch idea

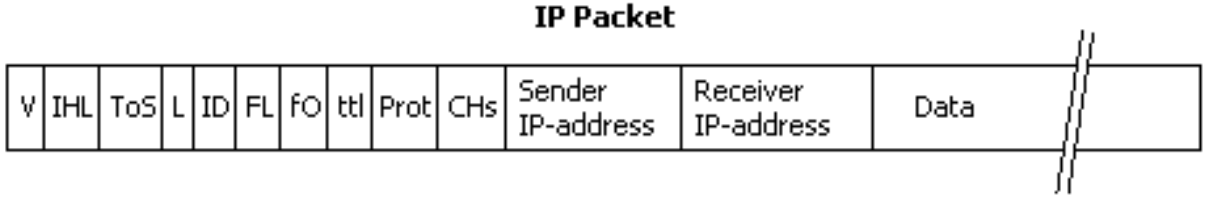
- Creates a small summary as an array of $w \times d$ in size $W = 2/\epsilon$, $d = log(1/\delta)$
- Use d hash functions to map vector entries to $[1..w]$
- Works on Insert-only and Insert-Delete model streams



$$W = 2/\epsilon,\ d = log(1/\delta)$$

Count-Min Sketch

Example: Count the number of packets from the set of IPs that cross a server in a network.

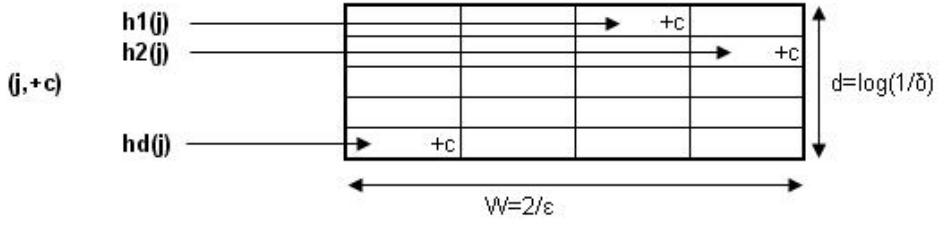


CM Sketch Update

Update:

Each entry in vector x is mapped to one bucket per row.

Increment the corresponding counter: $CM[k, h_k(j)] += 1$.

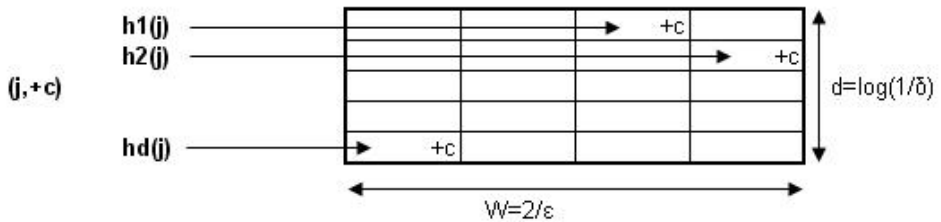


Count-Min Sketch

Example: Count the number of packets from the set of IPs that cross a server in a network.

CM Sketch Query

Query: How many packets from IP j?
Estimate $\hat{x}[j]$ by taking $min_k CM[k, h_k(j)]$



The estimate guarantees:

- $x[j] \leq \hat{x}[j]$
- $\hat{x}_i \leq \epsilon \times ||x_i||_1$, with probability $1 - \delta$.

FM Sketches

Count the Number of Distinct Values in a Stream

How many different IPs have been in observed in a stream of TCP/IP packets?

- Assume that the domain of the attribute is $\{0, 1, \ldots, M-1\}$.
- The problem is trivial if we have space linear in M.
- Is there an approximate solution using space $log(M)$?

FM Sketches for Distinct Value Estimation

Flajolet and Martin; *Probabilistic Counting Algorithms for DataBase Applications*, JCSS, 1983

- Maintain a *Hash Sketch* = BITMAP array of L bits,, where $L = \mathcal{O}(log(M))$, initialized to 0.
- Assume a hash function $h(x)$ that maps incoming values $x \in [0, \ldots, M-1]$, *uniformly* across $[0, \ldots, 2^{(L-1)}]$.
- Let $lsb(y)$ denote the position of the least-significant 1 bit in the binary representation of y.
- For each incoming value x, set BITMAP$[lsb(h(x))] = 1$.

FM Sketches for Distinct Value Estimation

Example:

BITMAP:

5	4	3	2	1	0
0	0	0	0	0	0

$x = 5 \rightarrow h(x) = 101100 \rightarrow lsb(h(x)) = 2$

BITMAP:

5	4	3	2	1	0
0	0	0	1	0	0

FM Sketches for Distinct Value Estimation

Example

Stream: 1,3,2,1,2,3,4,3,1,2,3,1,...

h(x) = 3x+1 mod 5

Processing the Stream

h(Stream)=4,5,2,4,2,5,3,5,4,2,5,4

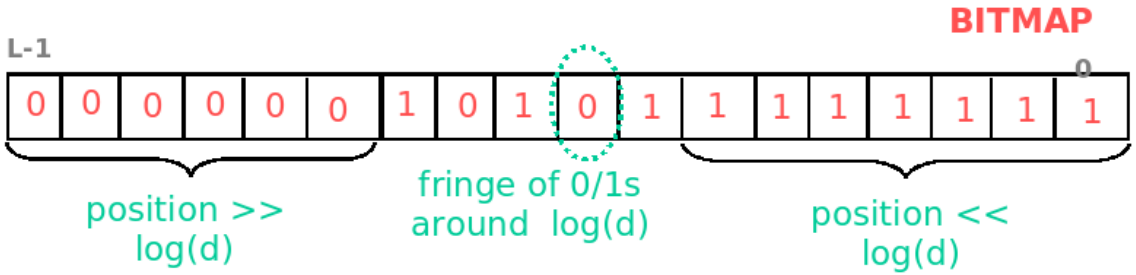
lsb(h(x))=2,0,1,2,1,0,0,0,2,1,0,2

Result

5	4	3	2	1	0
0	0	0	1	1	1

R= 2 Output =4

FM Sketches for Distinct Value Estimation



- By uniformity through $h(x)$:
  $P(BITMAP[k] = 1) = Prob(10^k) = 1/2^{k+1}$
- Let R be the position of the rightmost zero in BITMAP
- R is an indicator of $log(d)$
- Flajolet and Martin [FM85] prove that $E[R] = log(\phi M)$, where $\phi = .7735$
- Estimate of $M = 2^R/\phi$

Outline

Frequent Items (Heavy Hitters) in Data Streams

Manku and Motwani have two master algorithms in this area:

- Sticky Sampling
- Lossy Counting

G. S. Manku and R. Motwani. *Approximate Frequency Counts over Data Streams*, in Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), Hong Kong, China, August 2002.

Sticky Sampling

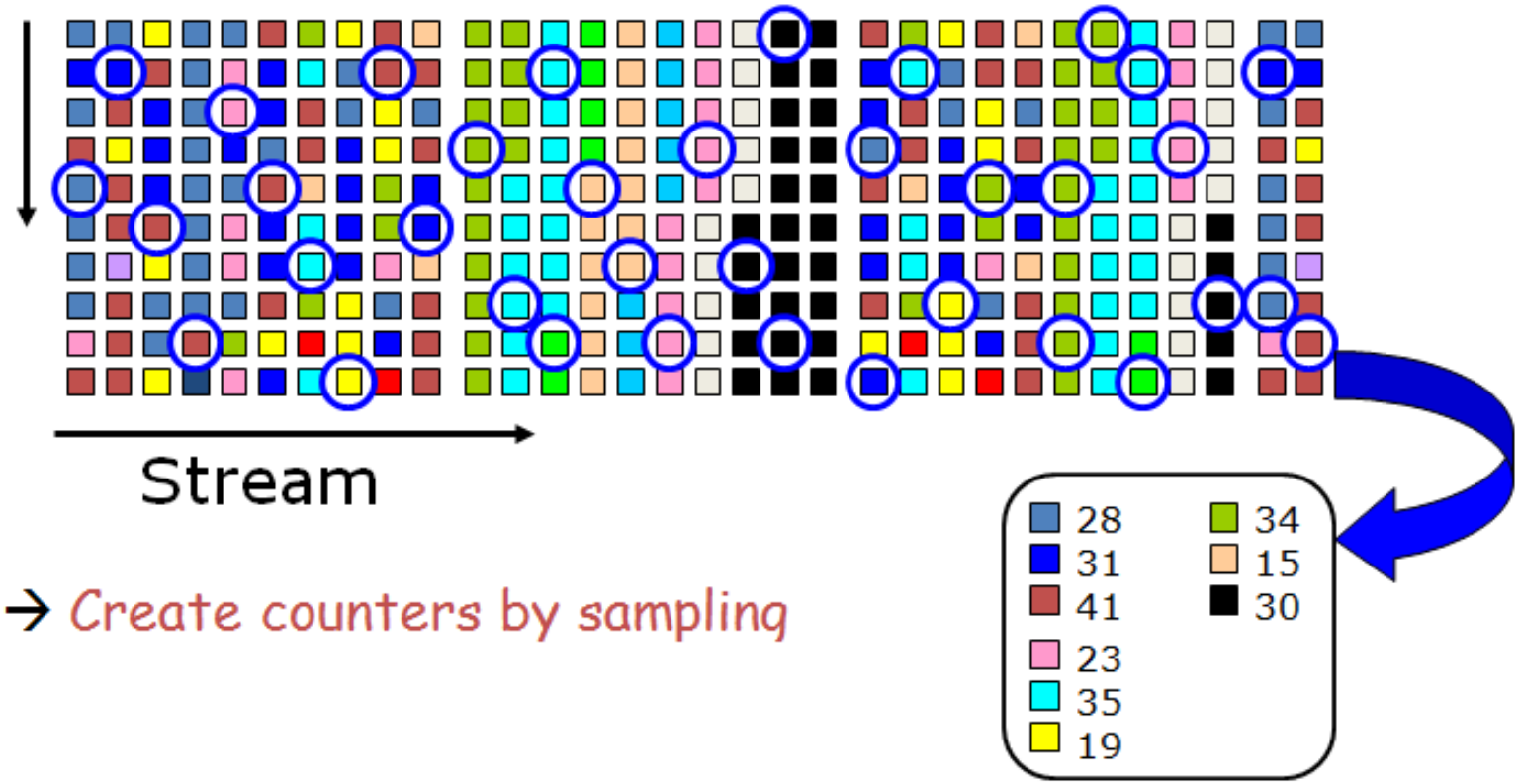
Sticky sampling is a probabilistic technique.

- The user inputs three parameters
  - Minimum Support(s)
  - Admissible Error (ϵ)
  - Probability of failure (δ)
- A simple data structure is maintained that has entries of data elements and their associated frequencies (e, f).
- The sampling rate decreases gradually with the increase in the number of processed data elements: $t = \frac{1}{\epsilon} log(s^{-1}\delta^{-1})$

Sticky Sampling

- For each incoming element in a data stream, the data structure is checked for an entry
  - If an entry exists, then increment the frequency
  - Otherwise sample the element with the current sampling rate.
  - If selected, then add a new entry, else the element is ignored.
- With every change in sampling rate, an unbiased coin toss is done for each entry with decreasing the frequency with every unsuccessful coin toss
- If the frequency goes down to zero, the entry is released

Sticky Sampling



Slide from R. Motwani talk.

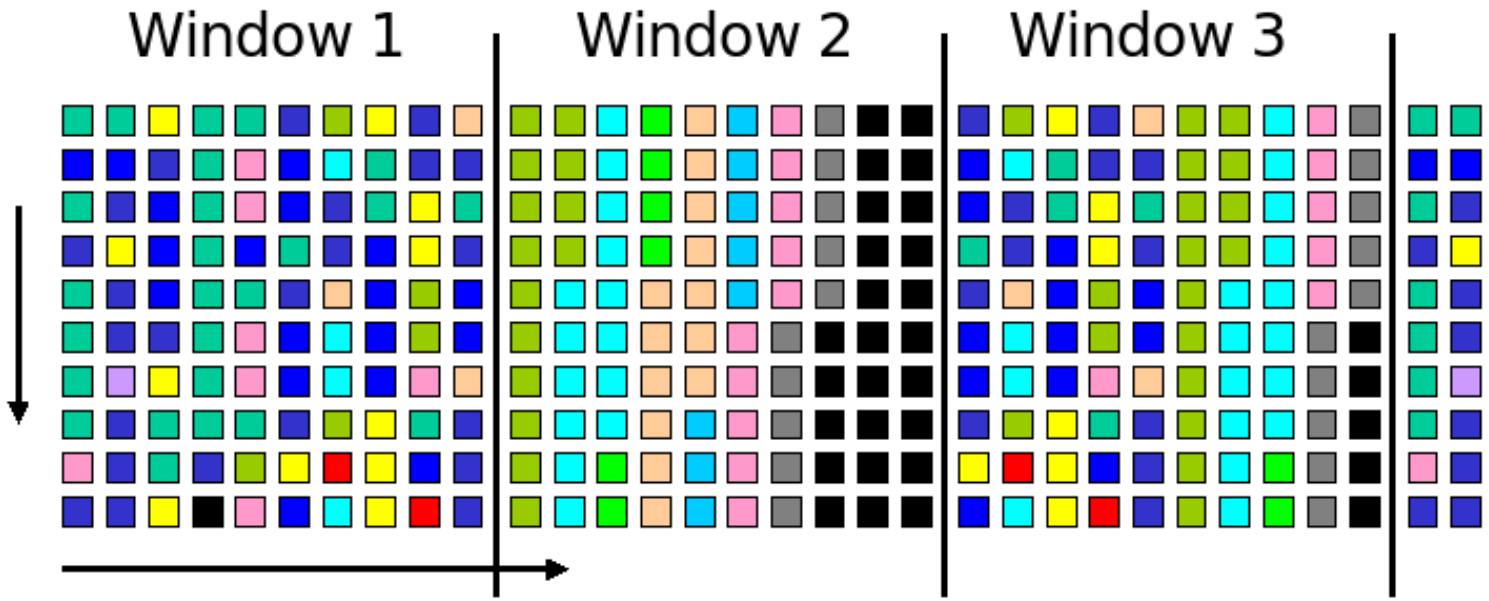
Lossy Counting

- Lossy counting is a deterministic technique.
- The user inputs two parameters
  - Minimum Support (s)
  - Admissible Error (ϵ)
- The data structure has entries of data elements, their associated frequencies (e, f, $\triangle$) where $\triangle$ is the maximum possible error in f.
- The stream is conceptually divided into buckets with a width $w = 1/\epsilon$.
- Each bucket is labeled by a value of N/w, where N starts from 1 and increases by 1.

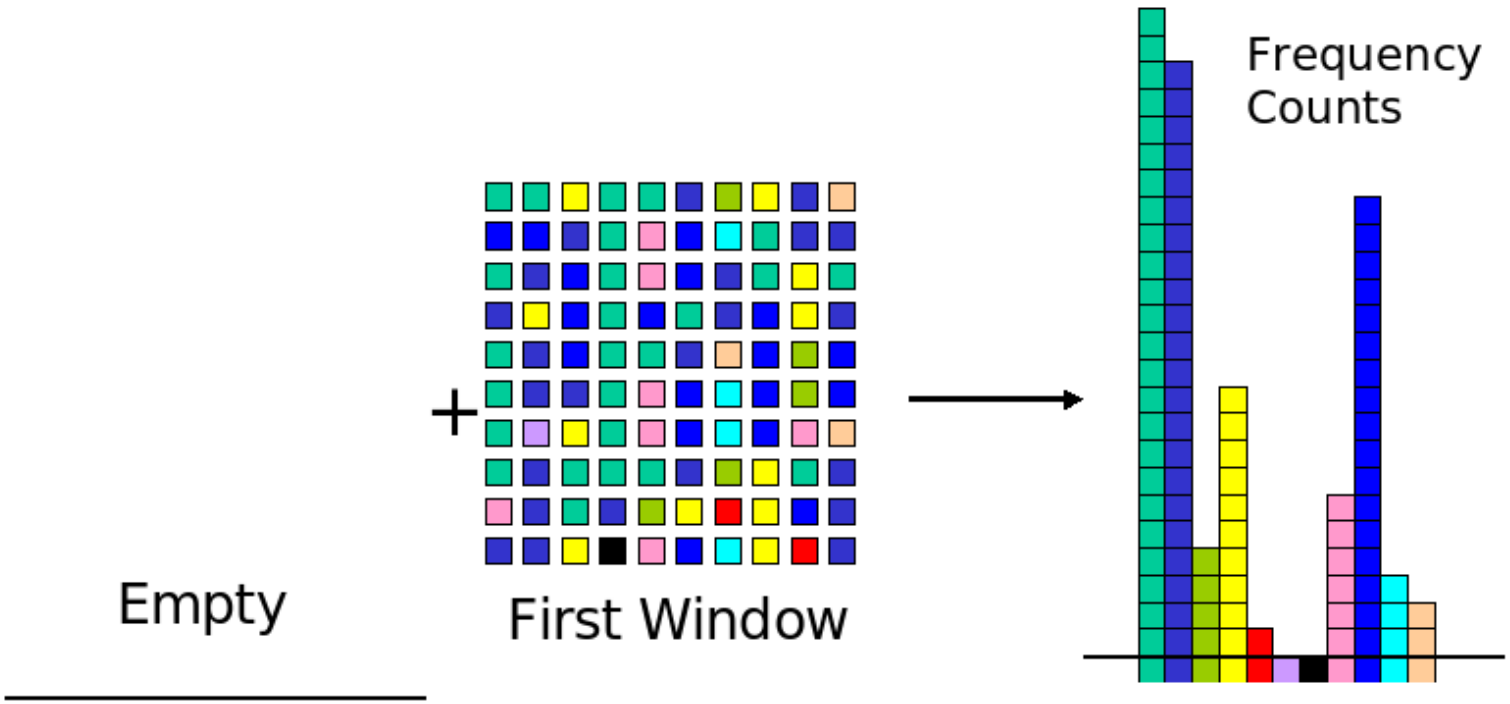
Lossy Counting

- For a new incoming element, the data structure is checked
  - If an entry exists, then increment the frequency
  - Otherwise, add a new entry with $\triangle = b_{current} - 1$ where $b_{current}$ is the current bucket label.
- When switching to a new bucket, all entries with $f + \triangle < b_{current}$ are deleted.

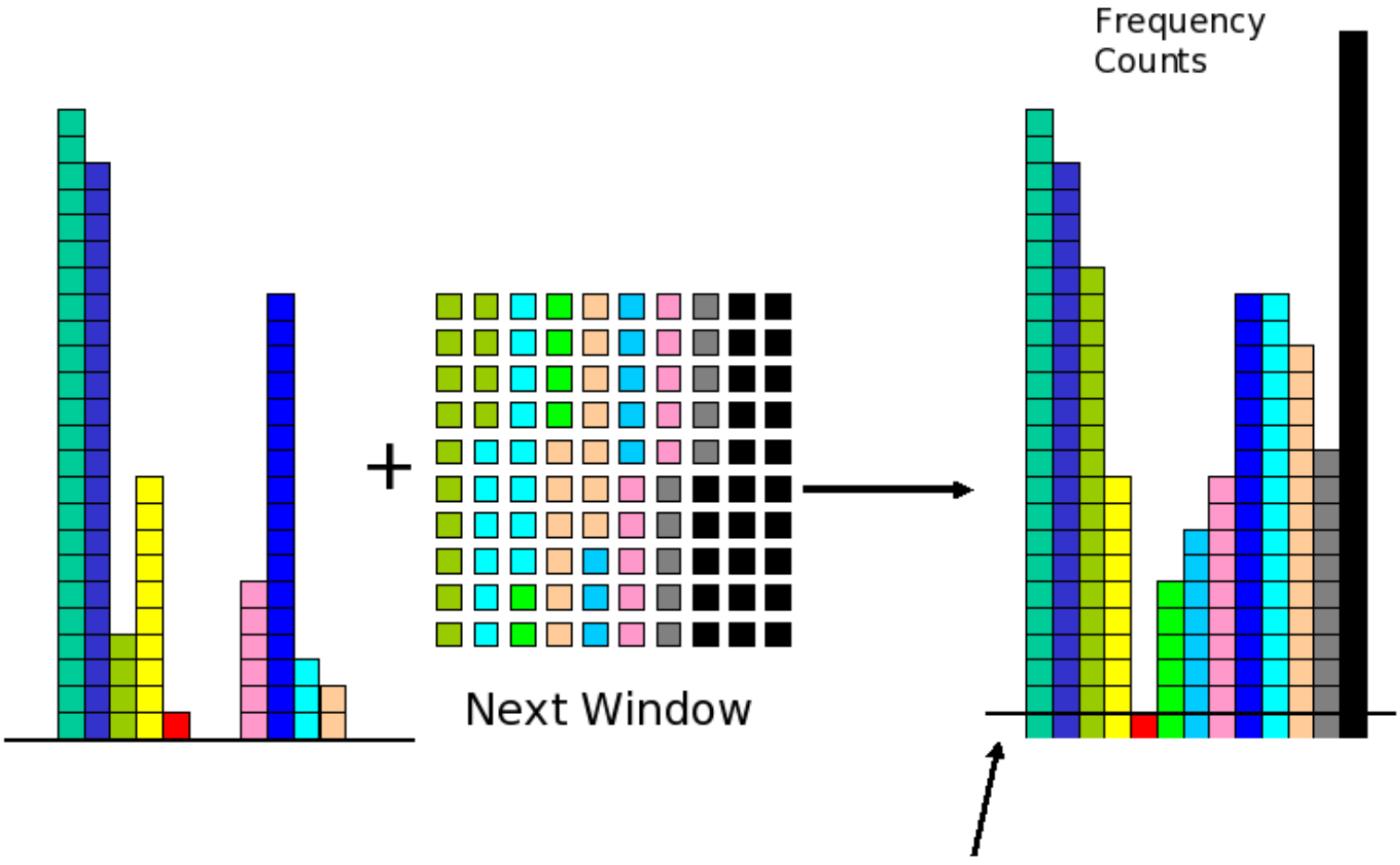
Lossy Counting: Example



Lossy Counting: Example



Lossy Counting: Example



At window boundary, decrement all counters

Error Analysis

Output:

- Elements with counter values exceeding $s \times N - \epsilon \times N$

How much do we undercount?

- If the current size of stream is N and window-size $= 1/\epsilon$ then *frequency error* $\leq$ *#window* $= \epsilon \times N$

Approximation guarantees:

- Frequencies underestimated by at most $\epsilon \times N$
- No false negatives
- False positives have true frequency at least $s \times N - \epsilon \times N$

How many counters do we need?

- Worst case: $1/\epsilon log(\epsilon N)$ counters

Frequent Itemsets in Data Streams

Manku and Motwani has extended Lossy Counting to find frequent itemsets.

G. S. Manku and R. Motwani. *Approximate Frequency Counts over Data Streams*, VLDB 2002

- The technique follows the same steps with batch processing of transactions according to memory availability.
- All subsets of the stored batch are checked and pruned.
- If the frequency of a new entry is greater than the number of buckets currently in memory, then a new entry is added to the data structure.

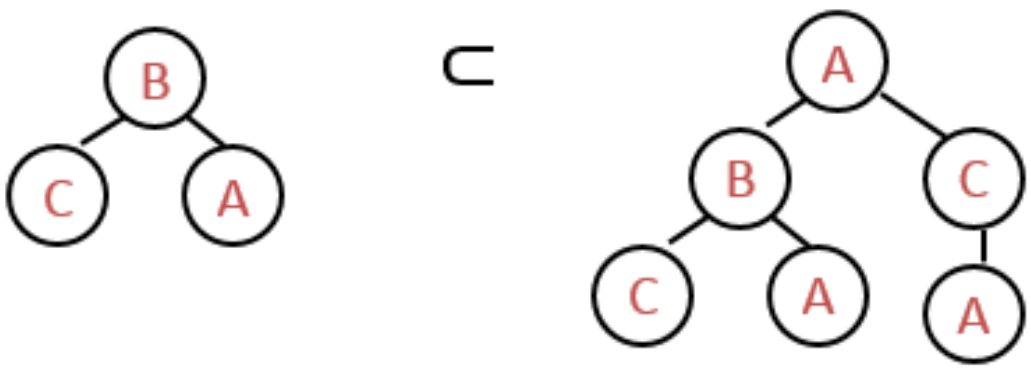
Outline

Pattern mining: definitions

Patterns: sets with a *subpattern* relation $\subset$

$$\{cheese, milk\} \subset \{milk, peanuts, cheese, butter\}$$

$$(search?buy) \subset (home?search?cart?buy?exit)$$

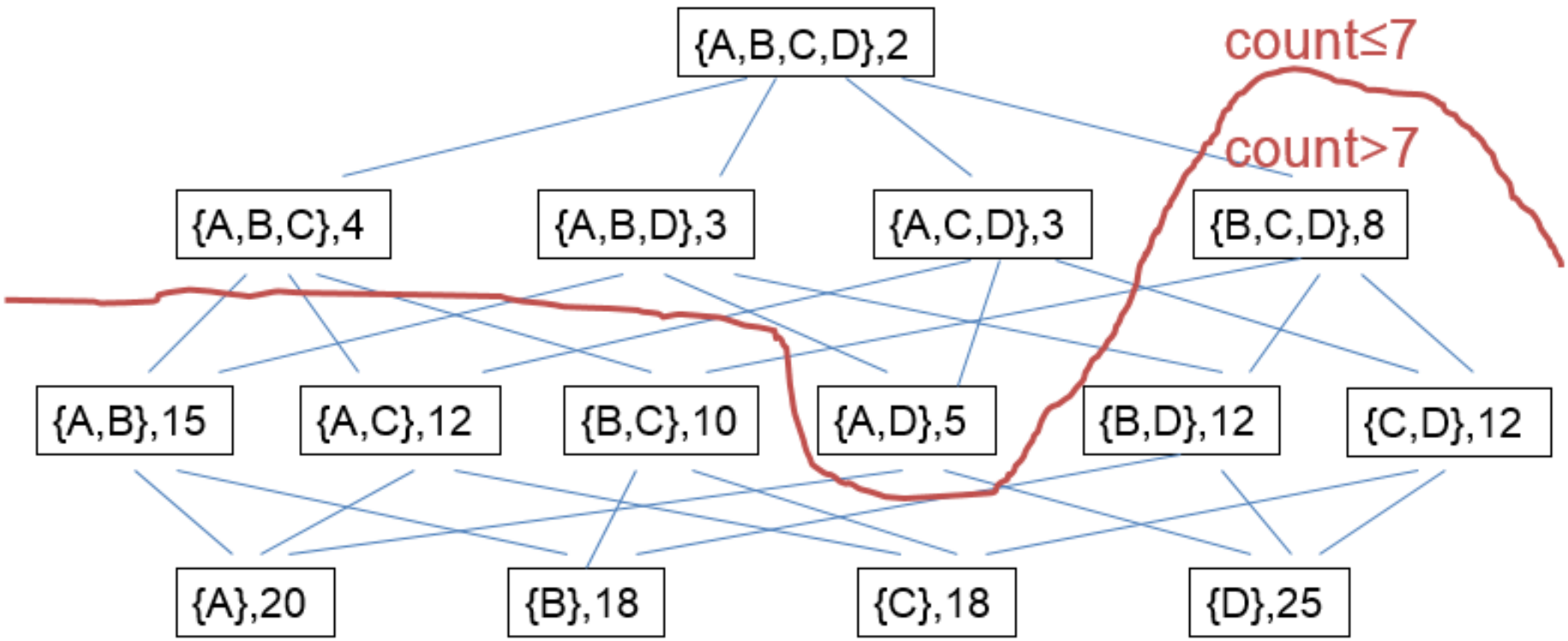


Applications: market basket analysis, intrusion detection, churn prediction, feature selection, XML query analysis, query and clickstream analysis, anomaly detection ...

Pattern mining in streams: definitions

- The support of a pattern T in a stream S at time t is the probability that a pattern T' drawn from S's distribution at time t is such that $T \subset T'$
- **Typical task**: Given access to S, at all times t, produce the set of patterns T with support at least ϵ at time t
- A pattern is closed if no superpattern has the same support.
- No information is lost if we focus only on closed patterns.

Key data structure: Lattice of patterns, with counts



Fundamentals

- *A priori* property: $t \subseteq t' \Rightarrow support(t) \geq support(t')$
- Closed: none of its supersets has the same support
  Can generate all freq. itemsets and their support
- Maximal: none of its supersets is frequent
  Can generate all freq. itemsets (without support)
- Maximal $\subseteq$ Closed $\subseteq$ Frequent $\subseteq$ D

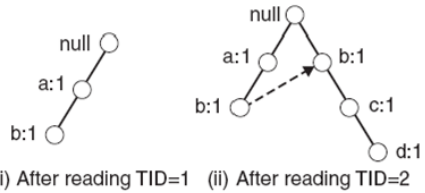
Base Algorithms

- **Apriori:** uses a generate-and-test approach: generates candidate itemsets and tests if they are frequent
  - Generation of itemsets is expensive (in both space and time)
  - Support counting is expensive
    - Subset checking (computationally expansive)
    - Multiple Database scans
- **FP-Growth:** (J. Han, J. Pei, and Y. Yin. *Mining Frequent Patterns without Candidate Generation* SIGMOD 2000)
  Allows frequent itemset discovery without candidate itemset generation. Two step approach:
  - **Step 1:** Built a compact data structure called FP-tree
    Built using 2 scans over the data set
  - **Step 2:** Extracts frequent itemsets directly from the FP-tree
    Traversal through FP-tree.

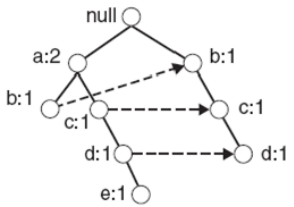
Core Data Structure: FP-Tree

Transaction Data Set

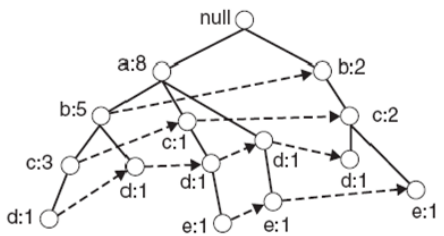
TID	Items
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}



(i) After reading TID=1 (ii) After reading TID=2



(iii) After reading TID=3



(iv) After reading TID=10

- Nodes corresponds to items and have a counter;
- FP-growth reads one transaction at a time and maps it to a path;
- Fixed order is used, so paths can overlap when transactions share items (when they have the same prefix).
- In this case, counters are incremented;
- Pointers are maintained between nodes containing the same item, creating singly linked lists (dotted lines);
- The more paths that overlap, the higher the compression. FP-tree may fit in memory;
- Frequent itemsets extracted from the FP-tree.

Step 1: FP-Tree Construction

FP-Tree is constructed using 2 scans over the data set:

- **Pass 1:**
  - Scan data and find support for each item;
  - Discard infrequent items;
  - Sort frequent items in decreasing order based on their support;
  - Use this order when building the FP-Tree, so common prefixes can be shared.

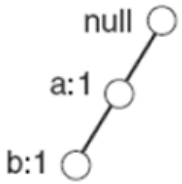
Step 1: FP-Tree Construction (Example)

- **Pass 2:** construct the FP-Tree (see diagram on next slide)
  - Read transaction 1: $\{a,b\}$
    - Create 2 nodes a, and b and the path $null \rightarrow a \rightarrow b$
    - Set counts a and b to 1.
  - Read transaction 2: $\{b,c,d\}$
    - Create 3 nodes b, c and d and the path $null \rightarrow b \rightarrow c \rightarrow d$
    - Set counts to 1
    - Although transactions 1 and 2 share b, the path are disjoint as they don't share a common prefix. Add the link between the b's.
  - Read transaction 3: $\{a,c,d,e\}$
    - It shares common prefix, item a with transaction 1 so the path for transaction 1 and 3 will overlap and the frequency count for node a will be incremented by 1. Add links between the c's and d's.

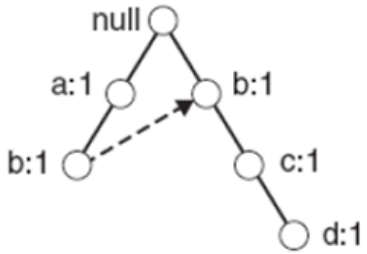
Step 1: FP-Tree Construction

Transaction Data Set

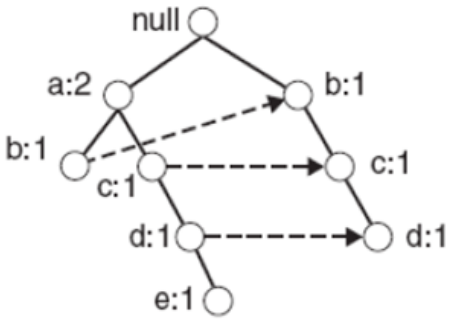
TID	Items
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}



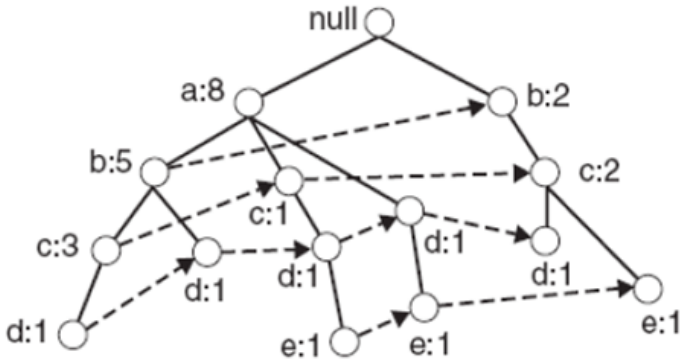
(i) After reading TID=1



(ii) After reading TID=2



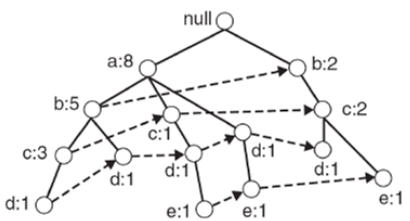
(iii) After reading TID=3



(iv) After reading TID=10

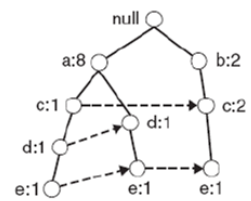
Step 2: Frequent Itemset Generation

- FP-Growth extracts frequent itemsets from the FP-Tree
- Bottom-up algorithm - from the leaves towards the root
  - Divide and conquer: first look for frequet itemsets ending in *e*, then *de*, etc ... then *d*, then *cd*, etc.
- First, extract prefix path sub-trees ending in an item(set). (*Hint:* use the linked lists.)

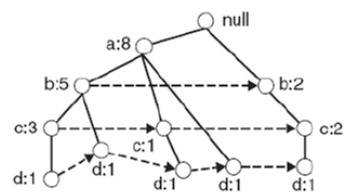


↑ Complete FP-tree

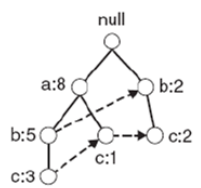
→ **Example:** prefix path sub-trees



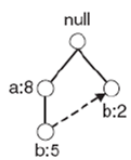
(a) Paths containing node e



(b) Paths containing node d



(c) Paths containing node c

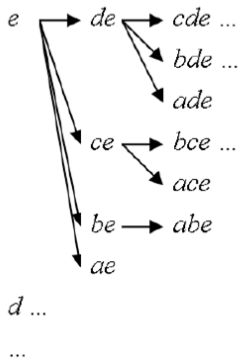


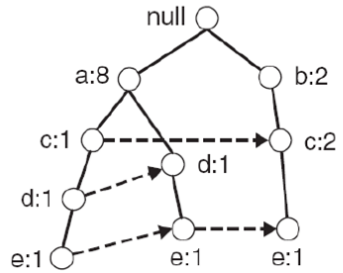
(d) Paths containing node b

(e) Paths containing node a

Step 2: Frequent Itemset Generation

- Each prefix path sub-tree is processed recursively to extract frequent itemsets. Solutions are then merged.
  - **E.g.** the prefix path sub-tree for *e* will be used to extract frequent itemsets ending in *e*, then in *de*,*ce*,*be*, and *ae*, etc.
  - Divide and conquer approach



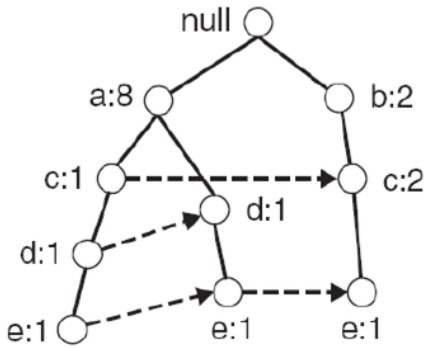


Prefix path sub-tree ending in *e*.

Step 2: Frequent Itemset Generation: Example

Let $minSup = 2$ and extract all frequent itemsets containing e.

1. Obtain the prefix path sub-tree for e:



2. Check if e is a frequent item by adding the counts along the linked list (dotted line). If so, extract it.
   - Yes, count=3 so $\{e\}$ is extracted as a frequent itemset.
3. As e is frequent, find frequent itemsets ending in e, i.e de, ce, be, and ae.
   - i.e. decompose the problem recursively.
   - To do this, we must first to obtain the conditional FP-tree for e.

FP-Stream

C. Giannella, J. Han, J. Pei, X. Yan, P. S. Yu: *Mining frequent patterns in data streams at multiple time granularities*. NGDM (2003)

- Multiple time granularities
- Based on FP-Growth (depth-first search over itemset lattice)
- Pattern-tree with Tilted-time window
  Tilted-time window: logarithmically aggregated time slots (log number of levels, aggregate when the level is full, push the aggregate one level up)
- Time sensitive queries, emphasis on recent history
- High time and memory complexity

Moment

Y. Chi , H. Wang, P. Yu , R. Muntz: *Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window*. ICDM 2004

- Keeps track of boundary below frequent itemsets
- Closed Enumeration Tree (CET) ($\approx$ prefix tree)
  - Infrequent gateway nodes (infrequent)
  - Unpromising gateway nodes (infrequent, dominated)
  - Intermediate nodes (frequent, dominated)
  - Closed nodes (frequent)
- By adding/removing transactions closed/infreq. do not change

Itemset mining

- MOMENT (Chi+ 04) (Sliding window, frequent closed, exact)
- CLOSTREAM (Yen+ 09) (Sliding window, all closed, exact)
- MFI (Li+ 09) (Transaction-sensitive window, frequent closed, exact)
- IncMine (Cheng+ 08) (Sliding window, frequent closed, approximate; faster for moderate approximate ratios)

Sequence, trees, and graph mining

- Frequent subsequence mining:
  MILE (Chen+05), SMDS (Marascu-Masseglia 06), SSBE (Koper-Nguyen 11)
- Bifet+08: Frequent closed unlbeled subtree mining
- Bifet+11: Frequent closed labeled subtree mining; Frequent closed labeled subgraph mining

Outline

Master References

- J. Gama, *Knowledge Discovery from Data Streams*, CRC Press, 2010.
- S. Muthukrishnan *Data Streams: Algorithms and Applications*, Foundations & Trends in Theoretical Computer Science, 2005.
- C. Aggarwal, *Data Streams: Models and Algorithms*, Ed. Charu Aggarwal, Springer, 2007
- B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems, in Proceedings of PODS, 2002.
- Gaber, M, M., Zaslavsky, A., and Krishnaswamy, S., Mining Data Streams: A Review, in ACM SIGMOD Record, Vol. 34, No. 1, March 2005, ISSN: 0163-5808
- C. Giannella, J. Han, J. Pei, X. Yan, P. S. Yu: *Mining frequent patterns in data streams at multiple time granularities*. NGDM (2003)

Bibliography on Frequent Item's

- *What's Hot and What's Not: Tracking Most Frequent Items Dynamically*, by G. Cormode, S. Muthukrishnan, PODS 2003.
- *Dynamically Maintaining Frequent Items Over A Data Stream*, by C. Jin, W. Qian, C. Sha, J. Yu, A. Zhou; CIKM 2003.
- *Processing Frequent Itemset Discovery Queries by Division and Set Containment Join Operators*, by R. Rantzau, DMKD 2003.
- *Approximate Frequency Counts over Data Streams*, by G. Singh Manku, R. Motawani, VLDB 2002.
- *Finding Hierarchical Heavy Hitters in Data Streams*, by G. Cormode, F. Korn, S. Muthukrishnan, D. Srivastava, VLDB 2003.
- J. Han, J. Pei, and Y. Yin. *Mining Frequent Patterns without Candidate Generation* SIGMOD 2000
- Metwally, D. Agrawal, A. Abbadi, *Efficient Computation of Frequent and Top-k Elements in Data Streams*, ICDT 2005
- Y. Chi , H. Wang, P. Yu , R. Muntz: *Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window* ICDM04