

Mining Data Streams

- Imagine a satellite-mounted remote sensor that is constantly generating data.
- The data are massive (e.g., terabytes in volume), temporally ordered, fast changing, and potentially infinite.
- This is an example of *stream data*.
- Examples
 - Telecommunications data
 - Transaction data from the retail industry
 - Data from electric power grids

Mining Complex data

- ❑ Stream data
 - Massive data, temporally ordered, fast changing and potentially infinite
 - Satellite Images, Data from electric power grids
- ❑ Time-Series data
 - Sequence of values obtained over time
 - Economic and Sales data, natural phenomenon
- ❑ Sequence data
 - Sequences of ordered elements or events (without time)
 - DNA and protein sequences
- ❑ Graphs
 - Represent various kinds of Networks
- ❑ Social Network Analysis

Mining Complex data

- ❑ Stream data

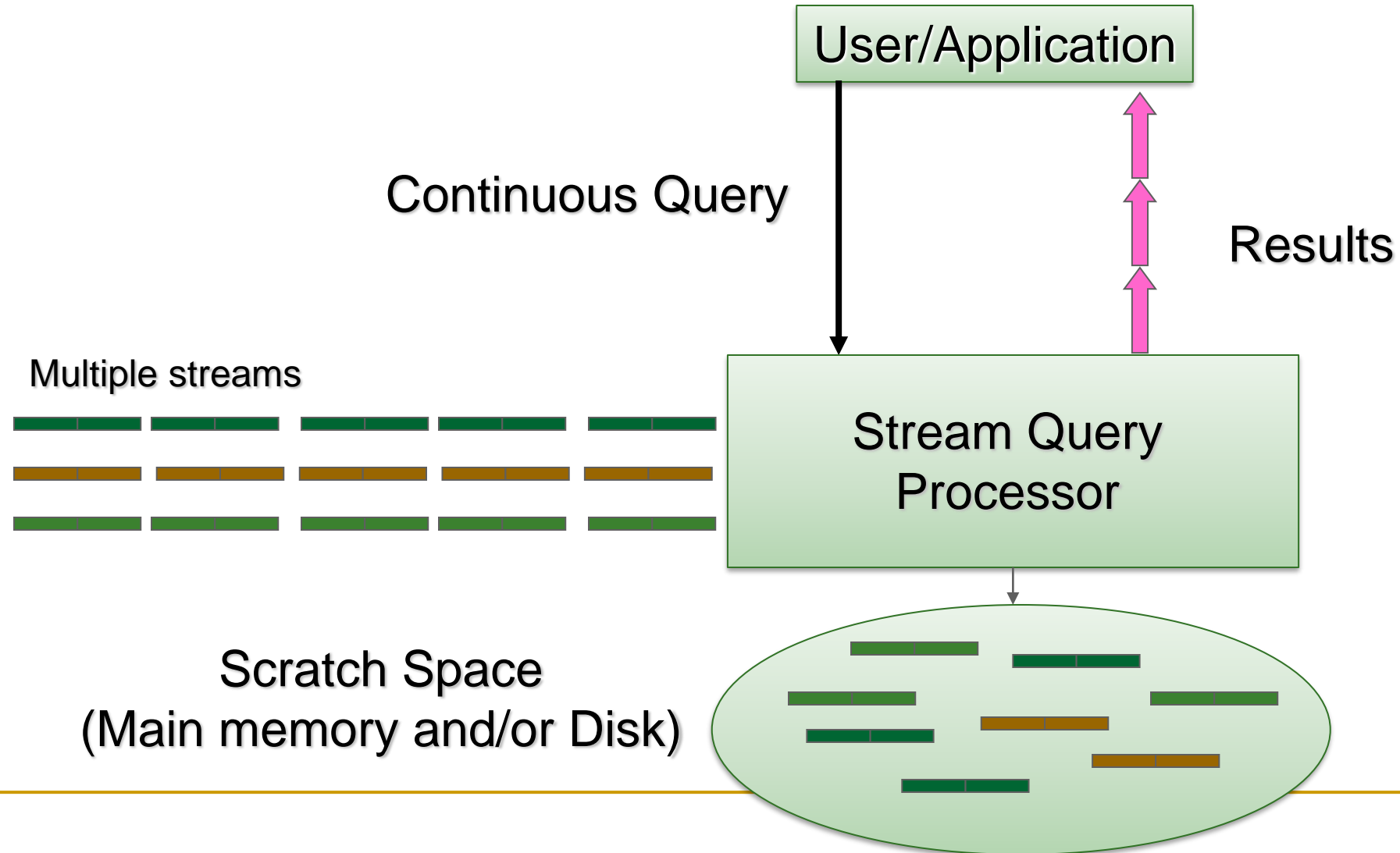
- Massive data, temporally ordered, fast changing and potentially infinite
- Satellite Images, Data from electric power grids
 - ❑ Streaming Data Characteristics
 - ❑ Architecture
 - ❑ Synopsis Structure – Sketches,....
 - ❑ Tilted Time Frame Models (3 models, critical layer, Materialization)
 - ❑ Frequent itemset approach
 - Lossy Counting Algorithm
 - ❑ Stream Classification techniques – 4 different types of models
 - ❑ Stream Clustering Techniques – STREAM, Clustream

Mining Data Streams

- **Streams** – Temporally ordered, fast changing, massive and potentially infinite.
- **Characteristics**
 - **Huge volumes** of continuous data, possibly infinite
 - **Fast changing** and requires fast, real-time response
 - Most stream data are at pretty **low-level** or multi-dimensional
 - Random access is expensive—**Single scan algorithms** are required
 - On-line, multi-level and multi-dimensional

Architecture: Stream Query Processing

DSMS (Data Stream Management System)



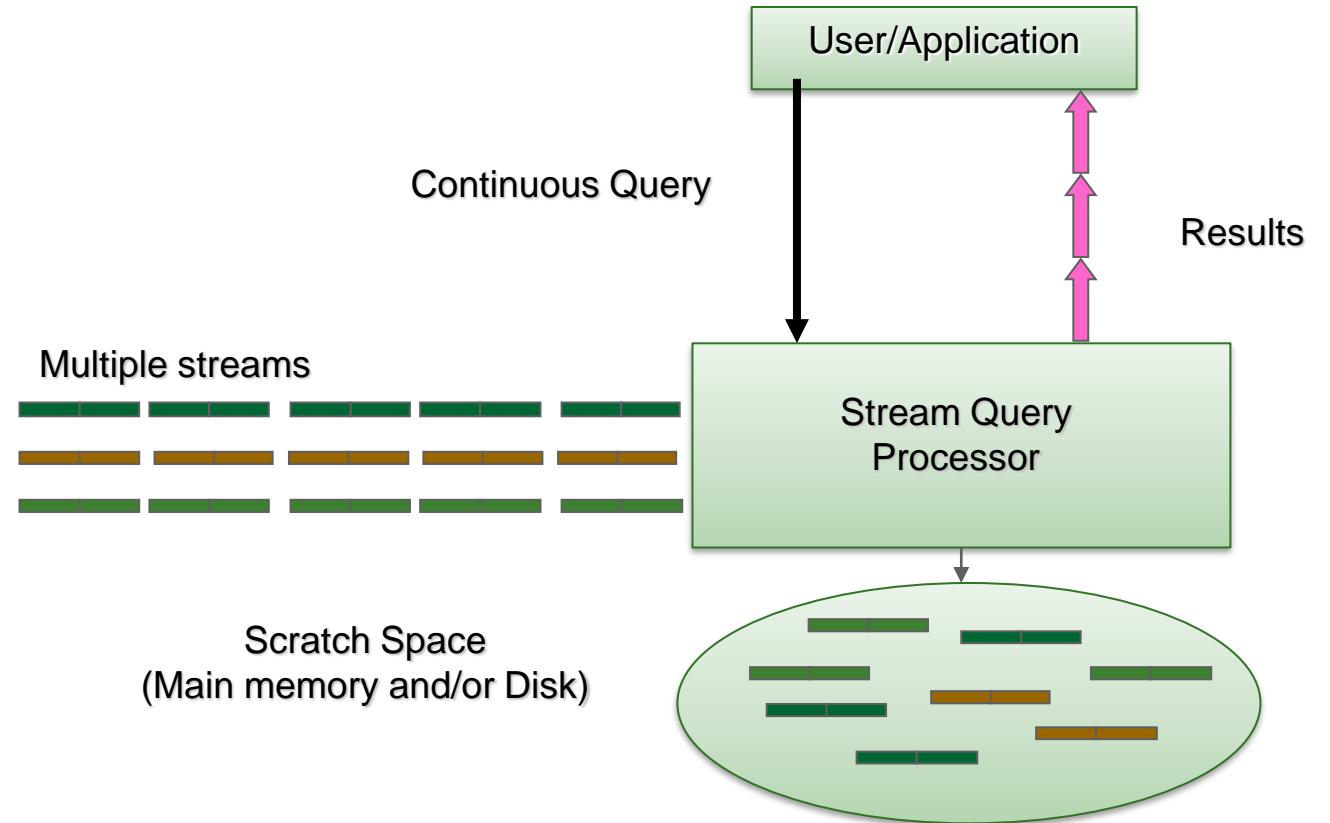
Architecture: Stream Query Processing

DSMS (Data Stream Management System)

In traditional database systems, data are stored in finite and persistent databases. However, stream data are infinite and impossible to store fully in a database.

In a Data Stream Management System (DSMS), there may be multiple data streams. They arrive on-line and are continuous, temporally ordered, and potentially infinite.

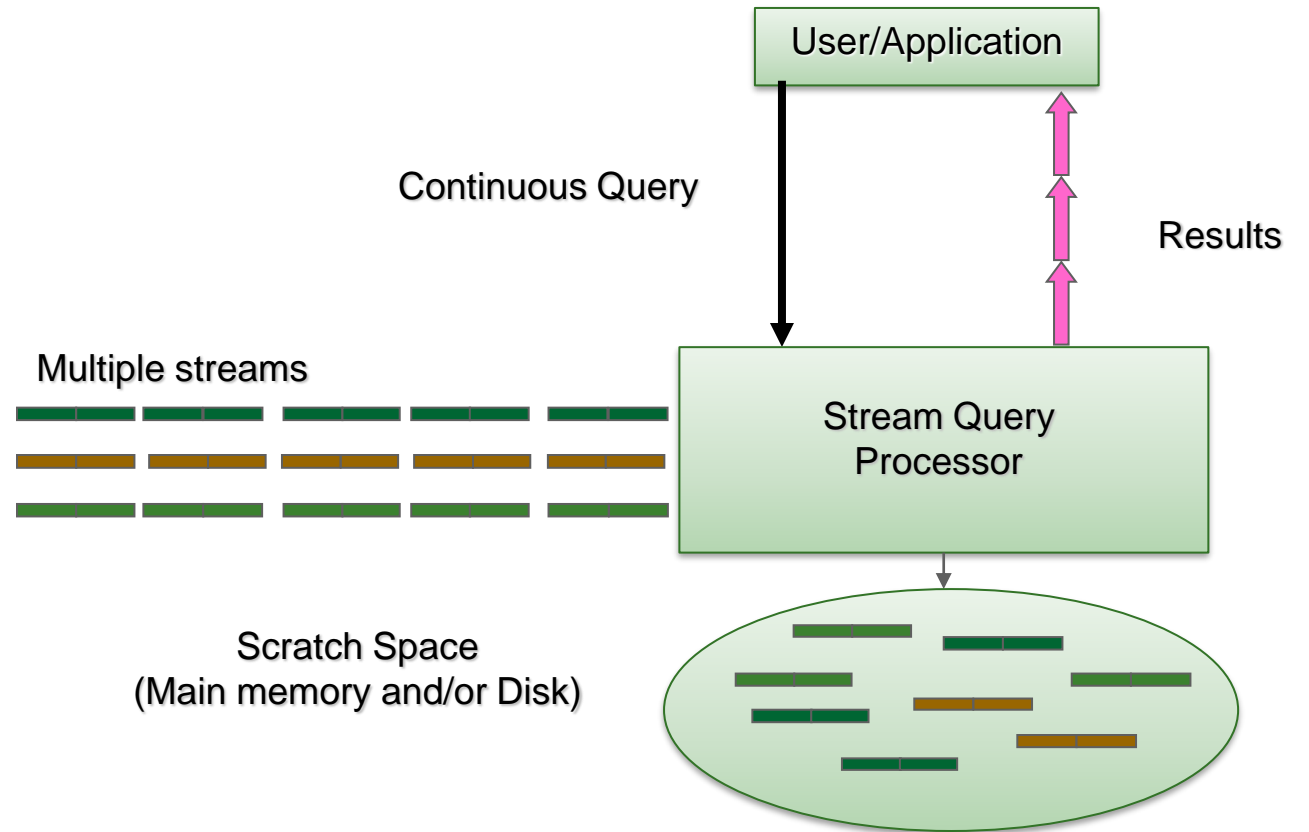
Once an element from a data stream has been processed, it is discarded or archived, and it cannot be easily retrieved unless it is explicitly stored in memory.



Architecture: Stream Query Processing

DSMS (Data Stream Management System)

A stream data query processing architecture includes three parts: *end user*, *query processor*, and *scratch space* (which may consist of main memory and disks). An end user issues a query to the DSMS, and the query processor takes the query, processes it using the information stored in the scratch space, and returns the results to the user.



Stream Queries

- Queries are often **continuous**
 - Evaluated continuously as stream data arrives
 - Answer updated over time
- Queries are often **complex**
 - Beyond element-at-a-time processing
 - Beyond stream-at-a-time processing
 - Beyond relational queries
- Query types
 - One-time query vs. **continuous query** (being evaluated continuously as stream continues to arrive)
 - **Predefined query** vs. ad-hoc query (issued on-line)
- Unbounded memory requirements

- Queries can be either *one-time queries* or *continuous queries*.
- A one-time query is evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user.
- A continuous query is evaluated continuously as data streams continue to arrive.
- The answer to a continuous query is produced over time, always reflecting the stream data seen so far.

Stream Query Processing Challenges

■ Unbounded memory requirements

- ❑ For real-time response, **main memory algorithm** should be used
- ❑ Memory requirement is unbounded due to future tuples

■ Approximate query answering

- ❑ With bounded memory, it is not always possible to produce exact answers
- ❑ **High-quality approximate answers** are desired
- ❑ Data reduction and synopsis construction methods
 - Sketches, random sampling, histograms, wavelets, etc.

Methodologies

- Impractical to scan more than once
- Even inspection is an issue
- Gigantic size rules out storage
- Size of Universe is also very large
 - Even generating summaries is not possible
- New data structures, techniques and algorithms are needed
- Tradeoff between Storage and accuracy
 - Synopses
 - Summaries of data - support approximate answers
 - Use Synopsis data structures
 - Algorithms must use poly-logarithmic space $O(\log^k N)$
 - Compute an approximate answer within a factor ε of the actual answer – As approximation factor goes down, space requirements increase

Synopsis Data Structure

- A *synopsis structure* for a dataset S is any summary of S whose size is substantively smaller than S
- Synopsis structures are small, often statistical summaries of a data set.
- The term serves as an umbrella for any summarization structure of sufficiently small size, such as random samples, histograms, wavelets, sketches, top-k summaries, etc.

Synopsis Data Structure

- Synopsis structures are most commonly used in conjunction with data streams.
- The goal is to construct, in one pass over the data stream, a synopsis structure that can be used to answer any query from a prespecified class of queries.
- At any point, a user may pose a query Q on the data stream thus far, and a (typically approximate) answer to Q must be produced using only the current synopsis structure.

Synopsis Data Structures and Techniques

- Random sampling
- Histograms
- Sliding windows
- Multi-resolution model
- Sketches
- Randomized algorithms

- Synopses techniques mainly differ by how exactly they trade off accuracy for storage.
- Sampling techniques and sliding window models focus on a small part of the data,
- whereas other synopses try to summarize the entire data, often at multiple levels of detail.
- Some techniques require multiple passes over the data, such as histograms and wavelets, whereas other methods, such as sketches, can operate in a single pass.

Synopsis Data Structures and Techniques

■ Random Sampling

- Without knowing the total length in advance
- Reservoir sampling
 - Maintain a set of s candidates in the reservoir, which form a true random sample of the element seen so far in the stream.
 - As the data stream flow, every new element has a certain probability (s/N) of replacing an old element in the reservoir.

■ Sliding Windows

- Make decisions based only on recent data of sliding window size w
- An element arriving at time t expires at time $t + w$

Synopsis Data Structures and Techniques

■ Histograms

- Approximate the frequency distribution of element values in a stream
- Partition data into a set of buckets
- Equal-width (equal value range for buckets) vs. V-optimal (minimizing frequency variance within each bucket)

■ Multi-resolution methods

- Data reduction through divide and conquer
- Balanced Binary tree – each level provides a different resolution
- Hierarchical clustering of trees
 - Micro clusters, Macro clusters and Summary Statistics
- Wavelets – For spatial and multimedia data
 - Builds a multi-resolution hierarchy structure of input signal

Synopsis Data Structures and Techniques

■ Sketches

- Sampling and Sliding window focus on small amount of data
- Histograms and Wavelets require several passes
- Sketches use **Frequency moments**
 - $U = \{1, 2, \dots, v\}$ $A = \{a_1, a_2, \dots, a_N\}$
 - F_0 – number of distinct elements
 - F_1 – length of sequence
 - F_2 – Self-join size / repeat rate / Gini's index of homogeneity
 - Frequency moments – give information about data and degree of skew or asymmetry
 - Given N elements and v values, sketches can approximate F_0, F_1, F_2 in $O(\log v + \log N)$ space
 - Each element is hashed onto $\{-1, +1\}$ – **Sketch Partitioning**
 - Random variable $X = \sum_i m_i z_i$ is maintained and can be used to approximate moments

Frequency Moments

Consider a stream $S = \{a_1, a_2, \dots, a_m\}$ with elements from a domain $D = \{v_1, v_2, \dots, v_n\}$. Let m_i denote the *frequency* (also sometimes called multiplicity) of value $v_i \in D$; i.e., the number of times v_i appears in S . The k^{th} frequency moment of the stream is defined as:

$$F_k = \sum_{i=1}^n m_i^k \tag{1}$$

Frequency Moments

To maintain the full histogram over the universe of objects or elements in a data stream, where the universe is $U = \{1, 2, \dots, v\}$ and the stream is $A = \{a_1, a_2, \dots, a_N\}$

for each value i in the universe, we want to maintain the frequency or number of occurrences of i in the sequence A . If the universe is large, this structure can be quite large as well. Thus, we need a smaller representation instead.

Consider the frequency moments of A . These are the numbers, F_k , defined as

$$F_k = \sum_{i=1}^v m_i^k,$$

where v is the universe or domain size (as above), m_i is the frequency of i in the sequence, and $k \geq 0$.

- In particular, $F0$ is the number of distinct elements in the sequence. $F1$ is the length of the sequence (that is, N , here).
- $F2$ is known as the *self-join size*, the repeat rate, or as Gini's index of homogeneity.
- The frequency moments of a data set provide useful information about the data for database applications, such as query answering.
- In addition, they indicate the degree of *skew* or asymmetry in the data which is useful in parallel database applications for determining an appropriate partitioning algorithm for the data.

Sketches

When the amount of memory available is smaller than v , we need to employ a synopsis. The estimation of the frequency moments can be done by synopses that are known as **sketches**.

These build a small-space summary for a distribution vector (e.g., histogram) using randomized linear projections of the underlying data vectors

Sketches provide probabilistic guarantees on the quality of the approximate answer

The basic idea is to hash every element uniformly at random to either $z \in \{-1, +1\}$, and then maintain a random variable,

$$X = \sum_i m_i z_i.$$

Synopsis Data Structures and Techniques

■ Randomized Algorithms

- Las Vegas – right answer but running time varies
- Monte Carlo – time bound; may not always return the correct answer
- When random variables are used, its deviation from the expected value must be bounded

■ Chebyshev's inequality:

- Let X be a random variable with mean μ and standard deviation σ

$$P(|X - \mu| > k) \leq \frac{\sigma^2}{k^2}$$

■ Chernoff bound:

- Let X be the sum of independent Poisson trials X_1, \dots, X_n , δ in $(0, 1]$
- The probability decreases exponentially as we move from the mean

$$P[X < (1 - \delta)\mu] < e^{-\mu\delta^2/4}$$

Stream OLAP and Stream Data Cubes

- Accommodating Stream data completely in a data warehouse is challenging
- Data is low-level so multi-dimensional analysis has to be done– **OLAP Analysis is needed**
- **Example:** Power Supply Stream data – fluctuation of power usage will be analyzed at higher levels such as by city/district.. / hourly...
 - Can be viewed as a Virtual data cube
 - Efficient methods are needed for systematic analysis
- Impossible to store complete stream data and compute a fully materialized cube
 - **Compression Techniques** are needed

Stream OLAP - Compression

- **A tilted time frame**

- Different time granularities
 - second, minute, quarter, hour, day, week, ...

- **Critical layers**

- Minimum interest layer (m-layer)
- Observation layer (o-layer)
- User: watches at o-layer and occasionally needs to drill-down down to m-layer

- **Partial materialization of stream cubes**

- Full materialization
- No materialization
- Partial materialization

Time Dimension with Compressed Time Scale

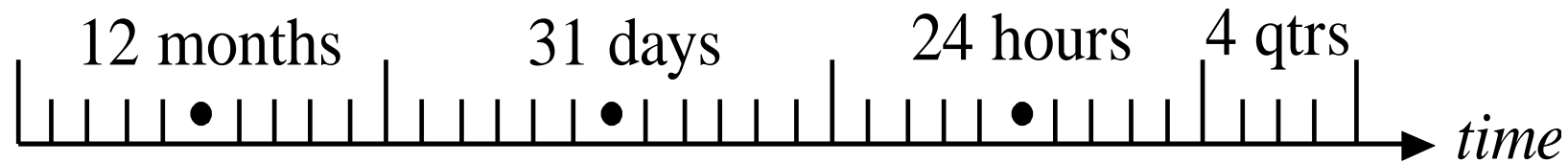
■ Tilted time Frame

- Recent changes are recorded at a fine scale and long-term changes at coarse scale
- Level of coarseness depends on application requirements and on how old the time point is
- Time Frame models
 - Natural tilted time frame model
 - Logarithmic tilted time frame model
 - Progressive logarithmic tilted time frame model

Tilted Time Frame

- Natural tilted time frame:

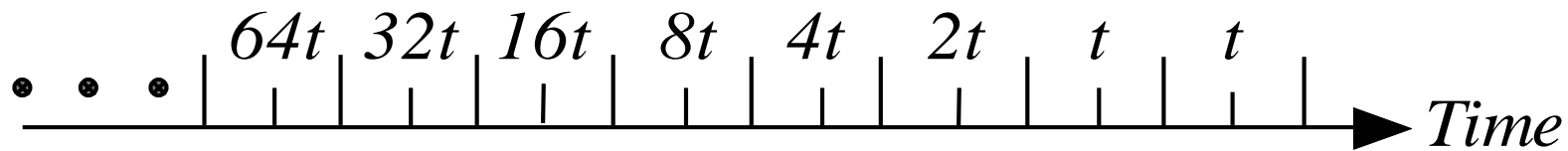
- Minimal: quarter, then 4 quarters → 1 hour, 24 hours → day, ...
- This model registers only $4 + 24 + 31 + 12 = 71$ units of time for a year instead of $365 \times 24 \times 4 = 35,040$ units



- The time frame is structured in multiple granularities : the most recent 4 quarters (15 minutes), followed by the last 24 hours, then 31 days, and then 12 months (the actual scale used is determined by the application).
- Based on this model, frequent itemsets can be computed in the last hour with the precision of a quarter of an hour, or in the last day with the precision of an hour.

Tilted Time Frame

- **Logarithmic tilted time frame:** (time frame is structured in multiple granularities according to a logarithmic scale)
 - Minimal: 1 quarter, then 1, 2, 4, 8, 16, 32, ... growing at an exponential rate
 - Records $\log_2(365 \times 24 \times 4) + 1 = 16.1$ units / year instead of $365 \times 24 \times 4 = 35,040$ units
 - Only 17 time frames will be needed to store the compressed information



Tilted Time Frame

■ Progressive Logarithmic tilted time frame

- The third method is the progressive logarithmic tilted time frame model, where snapshots are stored at differing levels of granularity depending on the recency
- Each snapshot is represented by its timestamp.
- The rules for insertion of a snapshot t (at time t) into the snapshot frame table
 - If $(t \bmod 2^i) = 0$ but $(t \bmod 2^{i+1}) \neq 0$, t is inserted into frame number i if $i \leq \text{max_frame}$ else into max_frame
 - Example:
 - Suppose there are 5 frames and each takes maximal 3 snapshots
 - Given a snapshot number N , if $N \bmod 2^d = 0$, insert into the frame number d . If there are more than 3 snapshots, remove the oldest one

Frame no.	Snapshots (by clock time)
0	69 67 65
1	66 62 58
2	68 60 52
3	56 40 24
4	48 16
5	64 32

Let T be the clock time elapsed since the beginning of the stream. Snapshots are classified into different *frame numbers*, which can vary from 0 to *max frame*, and *max capacity* is the maximal number of snapshots held in each frame.

Tilted Time Frame

The rules for insertion of a snapshot t (at time t) into the snapshot frame table

Frame no.	Snapshots (by clock time)
0	69 67 65
1	66 62 58
2	68 60 52
3	56 40 24
4	48 16
5	64 32

$t = 64$

$(64 \bmod 2^6) = 0$ and $(64 \bmod 2^7) \neq 0$,

$i = 6$

t is inserted into *frame number* i if $i \leq \text{max frame}$; otherwise (i.e., $i > \text{max frame}$), t is inserted into *max frame*;

Since this value of i exceeds *max frame*, 64 was inserted into frame 5 instead of frame 6.

$t = 70$

$(70 \bmod 2^1) = 0$ but $(70 \bmod 2^2) \neq 0$,

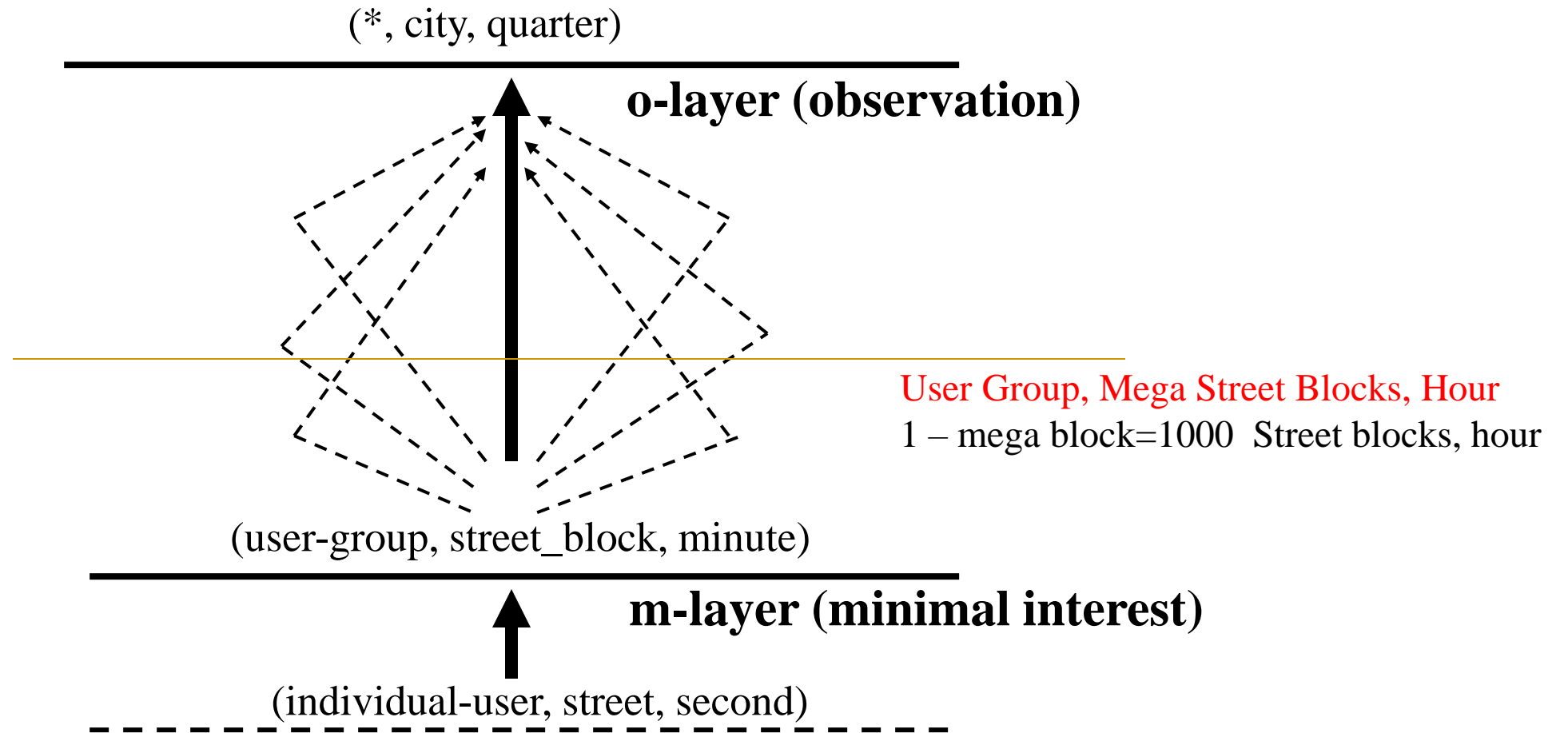
Each slot has a *max capacity*. At the insertion of t into *frame number* i , if the slot already reaches its *max capacity*, the oldest snapshot in this frame is removed and the new snapshot inserted

Insert 70 into *frame number* 1. This would knock out the oldest snapshot of 58, given the slot capacity of 3.

Critical Layers

- In many applications, it is beneficial to dynamically and incrementally compute and store two critical cuboids (or layers)
- Compute and store only **mission-critical cuboids** of the full data cube
 - **Minimal Interest layer** – not feasible to examine minute details
 - The first layer, called the minimal interest layer, is the minimally interesting layer that an analyst would like to study
 - **Observation layer** – layer at which analyst studies the data
 - The second layer, called the observation layer, is the layer at which an analyst (or an automated system) would like to continuously study the data.
 - Example: **Power Supply Stream data Cube**
 - Raw data layer – Individual_user, Street_address, second
 - Minimal Interest layer – user_group, street_block and minute
 - Observation layer – *(all_user), city, quarter

Critical Layers



Partial Materialization

“What if a user needs a layer that would be between the two critical layers?”

■ On-line materialization

- ❑ Materialization takes precious space and time
 - Only incremental materialization (with tilted time frame)
- ❑ Only materialize “cuboids” of the critical layers
 - Online computation may take too much time
- ❑ Preferred solution:
 - **Popular-path approach:** Materializing those along the popular drilling paths
 - **H-tree (Hyperlink tree) structure:** Such cuboids can be computed and stored efficiently using the H-tree structure
 - ❑ Aggregate Cells in non-leaf nodes
 - ❑ Based on popular path – all cells are cuboids on popular path
 - non-leaf nodes of H-tree; space and computation overheads are reduced

Materializing a cube at only two critical layers leaves much room for how to compute the cuboids in between. These cuboids can be precomputed fully, partially, or not at all (i.e., leave everything to be computed on the fly).

Frequent Pattern Mining in Data Streams

- Existing FP Mining algorithms require to scan the complete data set
- In data streams an infrequent item may become frequent and vice versa
- Approach I – Keep track of only limited set of items / itemsets
- Approach II – Approximate set of answers
 - Example: a router is interested in all flows:
 - whose frequency is at least 1% (σ) of the entire traffic stream seen so far and feels that 1/10 of σ ($\epsilon = 0.1\%$) error is comfortable
 - All frequent items with a support of atleast min_support will be output; some items with a support of min_support – ϵ will also be output

Lossy Counting Algorithm

- **Input:** min_support threshold σ and error bound ϵ
 - Incoming stream is divided into buckets of width $w = \lceil 1/\epsilon \rceil$
 - N – Current stream length
- **Frequency list data structure** is maintained
 - For each item – approximate frequency count f , and maximum possible error Δ are maintained
 - If the item is from the b^{th} bucket, the **maximum possible error Δ** on the frequency count of the item is $b-1$
 - When a bucket boundary is reached, an item entry is deleted if $f + \Delta \leq b$ the current bucket number – Frequency list is kept small
 - The frequency of an item can be under-estimated by at most ϵN
 - $f + \Delta \leq b$; $b \leq N/w$ i.e $b \leq N\epsilon$ so $f + \Delta \leq N\epsilon$
 - **All frequent items and some sub-frequent items with frequency $\sigma N - \epsilon N$ will be output**

Lossy Counting Algorithm

■ Properties

- No false Negatives (all frequent items are output)
- False positives have a frequency of at least $\sigma N - \epsilon N$
- Frequency of a frequent item is under-estimated by at most ϵN

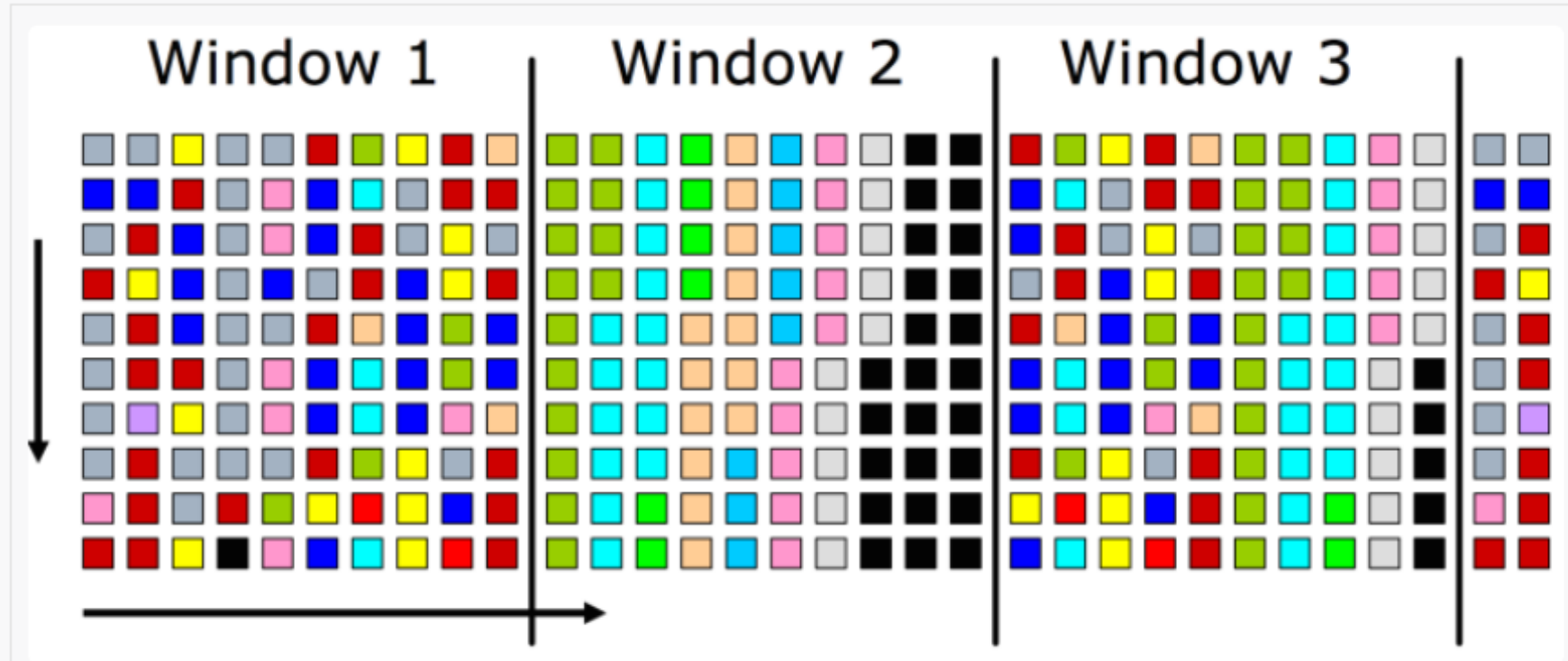
■ Reasonable Space Requirements – $7/\epsilon$

EXAMPLE

<http://www.mathcs.emory.edu/~cheung/Courses/584/Syllabus/07-Heavy/Manku.html>

Lossy Counting

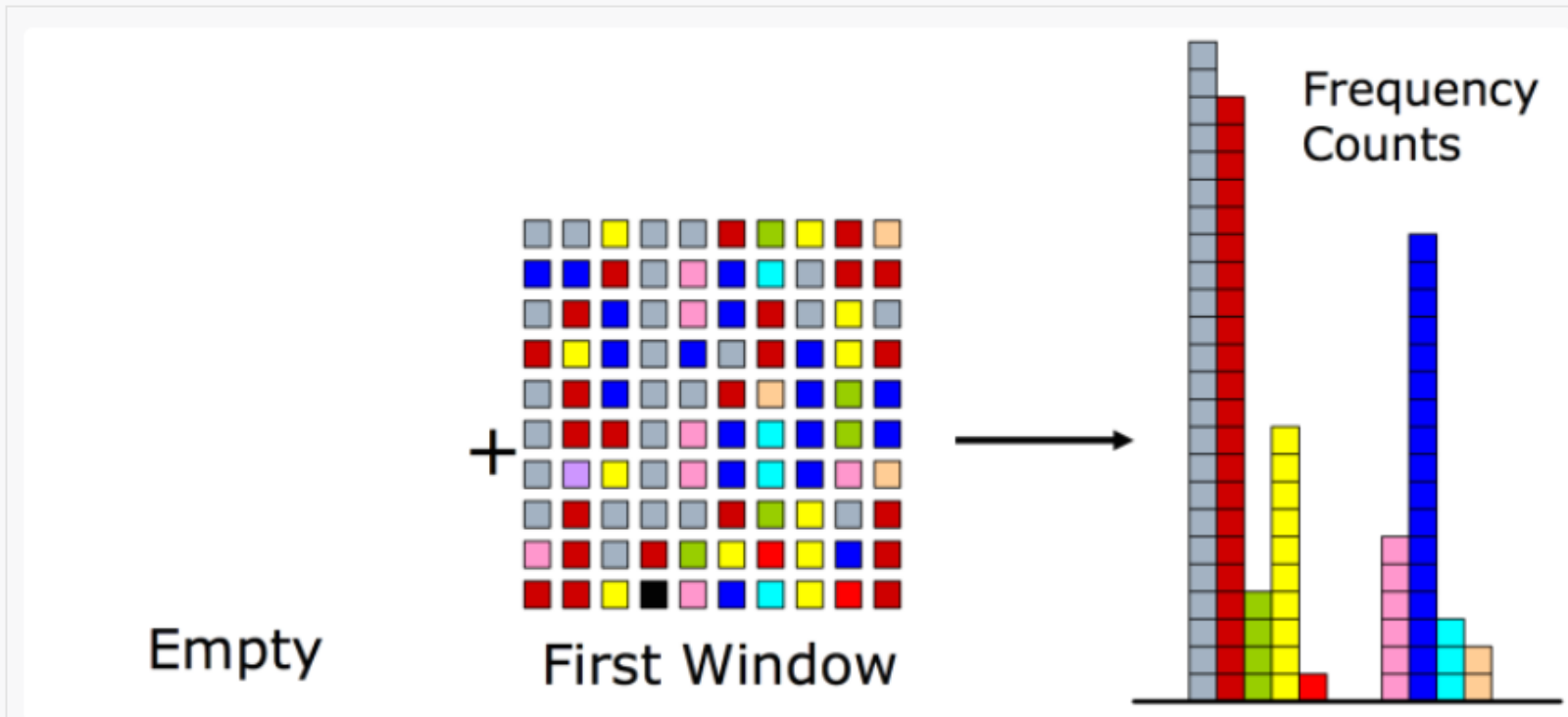
Step 1: Divide the incoming data stream into windows.



Split input stream into windows

Lossy Counting

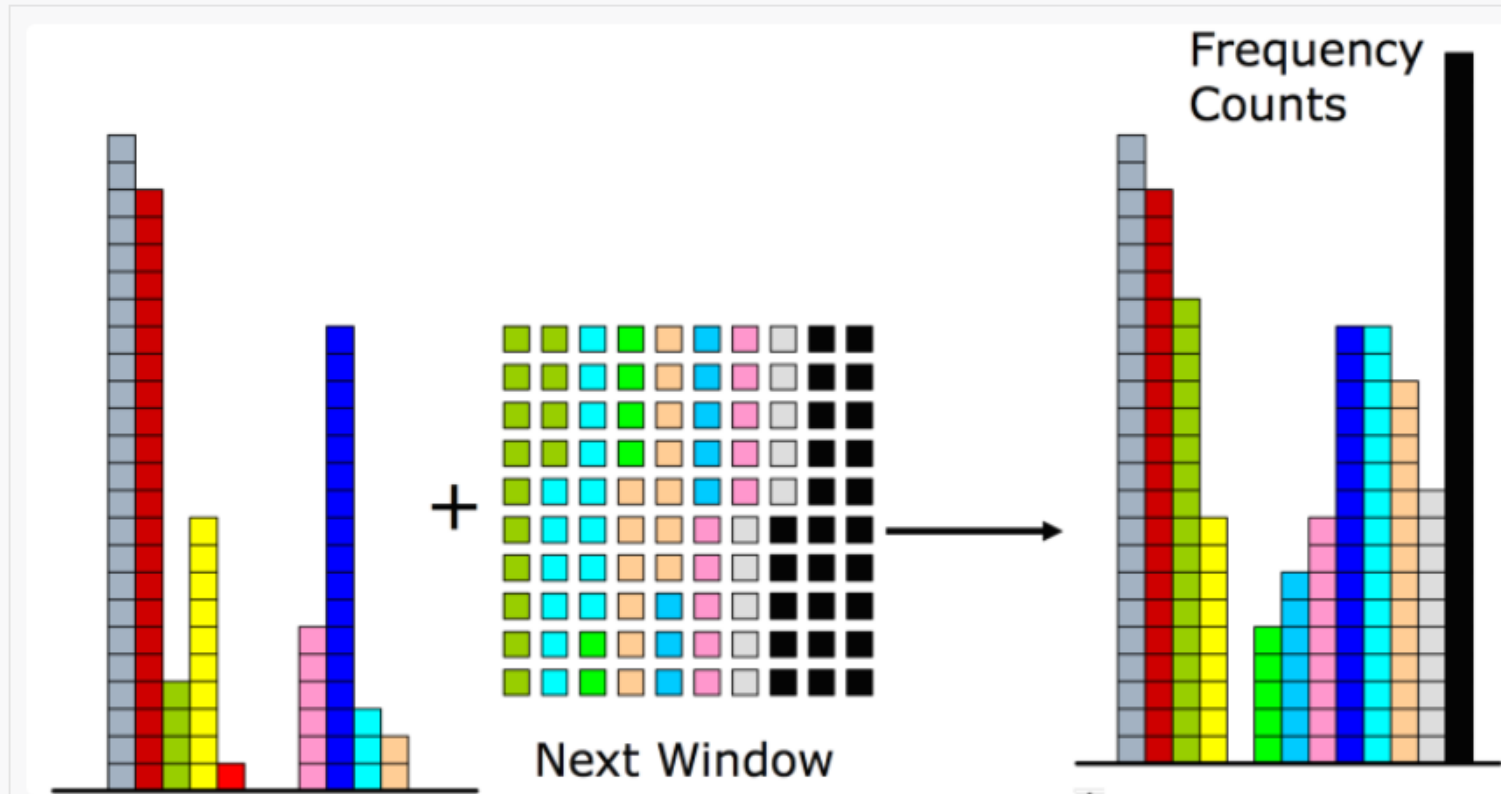
Step 2: Increment the frequency count of each item according to the new window values.
After each window, decrement all counters by 1.



Increment frequency counts – At window boundary adjust counts

Lossy Counting

Step 3: Repeat – Update counters and after each window, decrement all counters by 1.



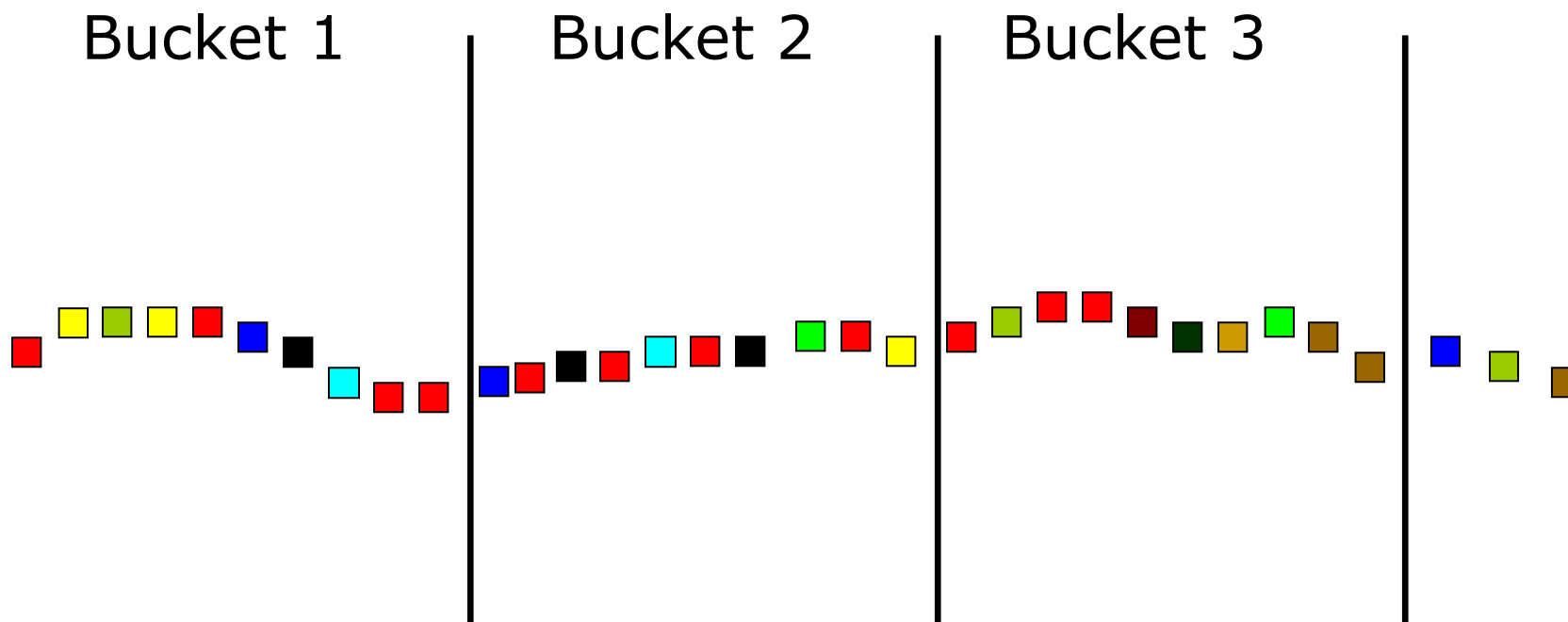
Update counters and at window boundary decrement all items by 1

Lossy Counting

The most frequently viewed items “survive”. Given a frequency threshold f , a frequency error e , and total number of elements N , the output can be expressed as follows:

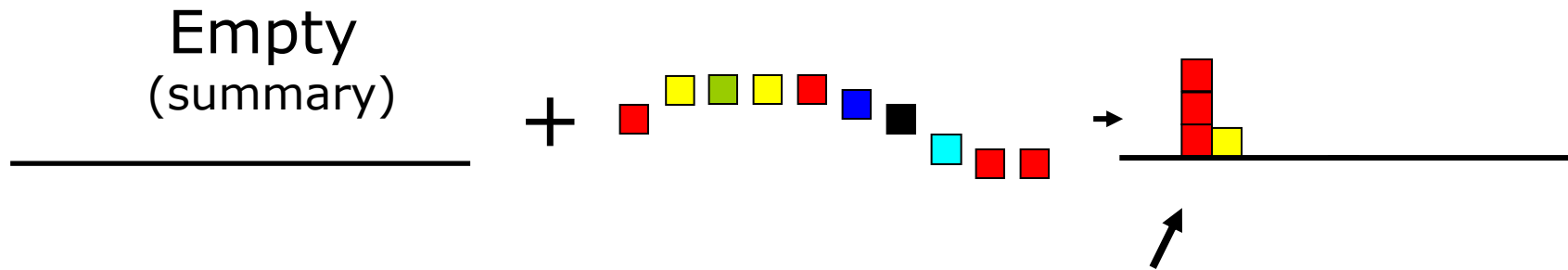
Elements with count exceeding $fN - eN$.

Lossy Counting Algorithm

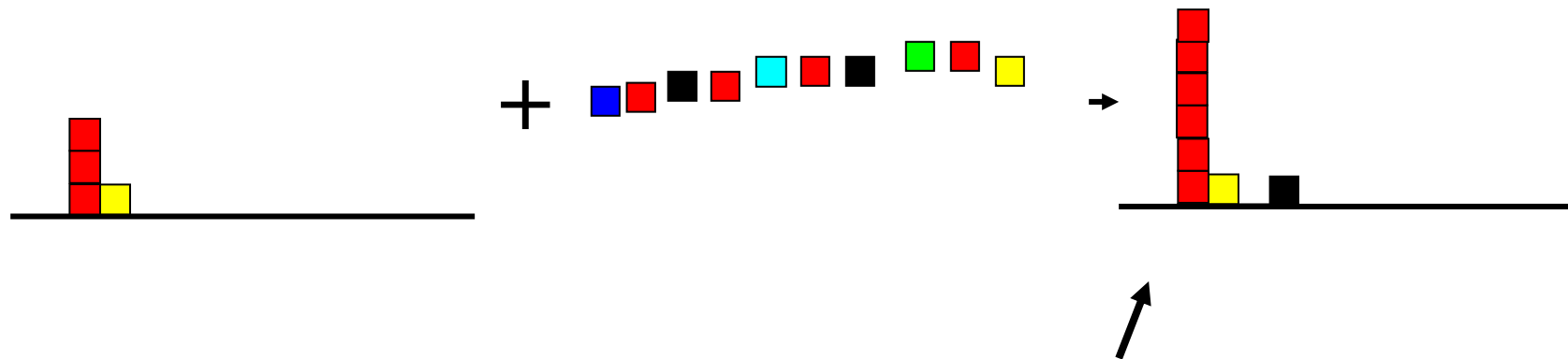


Divide Stream into 'Buckets' (bucket size is $1/\epsilon = 1000$)

Lossy Counting Algorithm



At bucket boundary, decrease all counters by 1



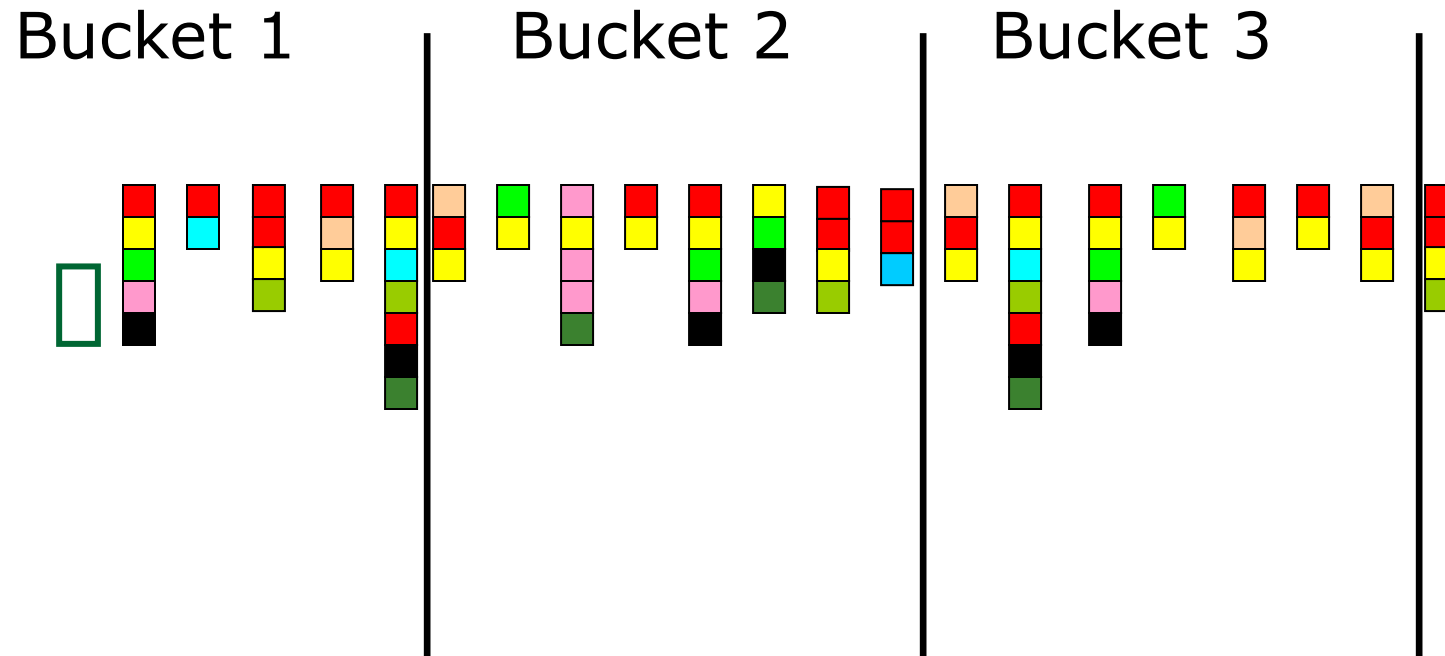
At bucket boundary, decrease all counters by 1

Lossy Counting Algorithm

- To find frequent itemsets
 - Load as many buckets as possible into main memory - β
 - If updated frequency $f + \Delta \leq b$ where b is the current bucket number entry can be deleted
 - If an itemset has frequency $f \geq \beta$ and does not appear in the list, it is inserted as a new entry with Δ set to $b - \beta$

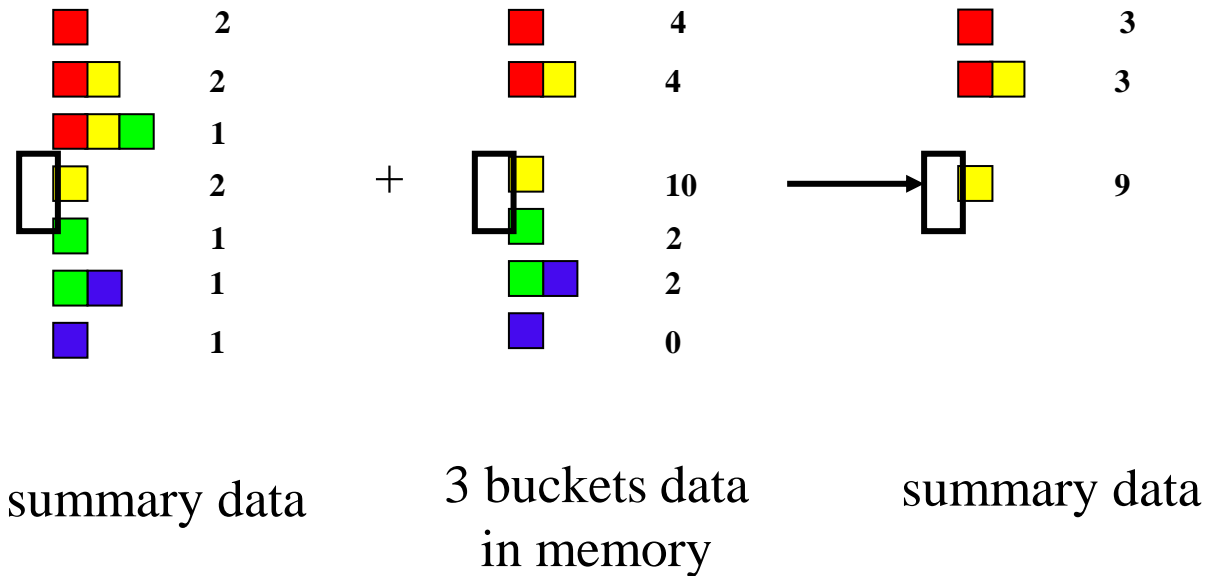
Lossy Counting Algorithm


Divide Stream into 'Buckets' as for frequent items
But fill as many buckets as possible in main memory one time



If we put 3 buckets of data into main memory one time,
Then decrease each frequency count by 3

Lossy Counting Algorithm



Itemset  is deleted.

Choosing a large number of buckets helps to delete more

Lossy Counting Algorithm

■ Strengths:

- A simple idea
- Can be extended to frequent itemsets

■ Weakness:

- Space Bound is not good
- For frequent itemsets, they do scan each record many times
- The output is based on all previous data. But sometimes, we are only interested in recent data

Classification of Dynamic Data Streams

- Traditional Classification techniques scan the training data multiple times – Off line process
 - Not possible in data streams
 - Example: Decision trees – at each node best possible split is determined by considering all the data, for all attributes
 - Concept drift occurs in streams
 - Changes in the classification model over time
- Stream Classification techniques
 - Hoeffding tree Algorithm
 - Very Fast Decision Tree (VFDT)
 - Concept-adapting Very Fast Decision Tree
 - Classifier Ensemble

Hoeffding Tree Algorithm

«Suppose we have made n independent observations of a variable r with domain R , and computed their mean \bar{r} . The Hoeffding bound states that, with probability $1 - \delta$, the true mean of the variable is at least $\bar{r} - \epsilon$ »

$$\epsilon = \sqrt{\frac{R^2 \ln(1 / \delta)}{2n}}$$

Hoeffding Tree Algorithm

- **Decision Tree Learning Algorithm**
 - Used originally to track Web click streams and predict next access
- Hoeffding trees – small sample is adequate to choose optimal splitting attribute
- Uses **Hoeffding bound** (or *additive Chernoff bound*)
 - r : random variable, R : range of r
 - n : # independent observations
 - Mean computed from sample – r_{avg}
 - True Mean of r is at least $r_{\text{avg}} - \epsilon$, with probability $1 - \delta$ (user-input)
$$\epsilon = \sqrt{\frac{R^2 \ln(1 / \delta)}{2n}}$$
 - Determines smallest number of samples needed at a node to split

Hoeffding Tree Algorithm

- **Hoeffding Tree Input**

S: sequence of examples

X: attributes

G(): evaluation function (information gain, gain ratio, Gini index, or some other attribute selection measure)

d: desired accuracy

- **Hoeffding Tree Algorithm**

for each node

 retrieve $G(X_a)$ and $G(X_b)$

 //two highest $G(X_i)$

 if ($G(X_a) - G(X_b) > \epsilon$)

 split on X_a

 recurse to next node

 break

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2N}}$$

- **Complexity** : $O(ldvc)$; l- maximum depth; d – number of attributes; v- maximum number of values for any attribute; c- number of classes

- Disagreement with traditional tree is at-most δ / p

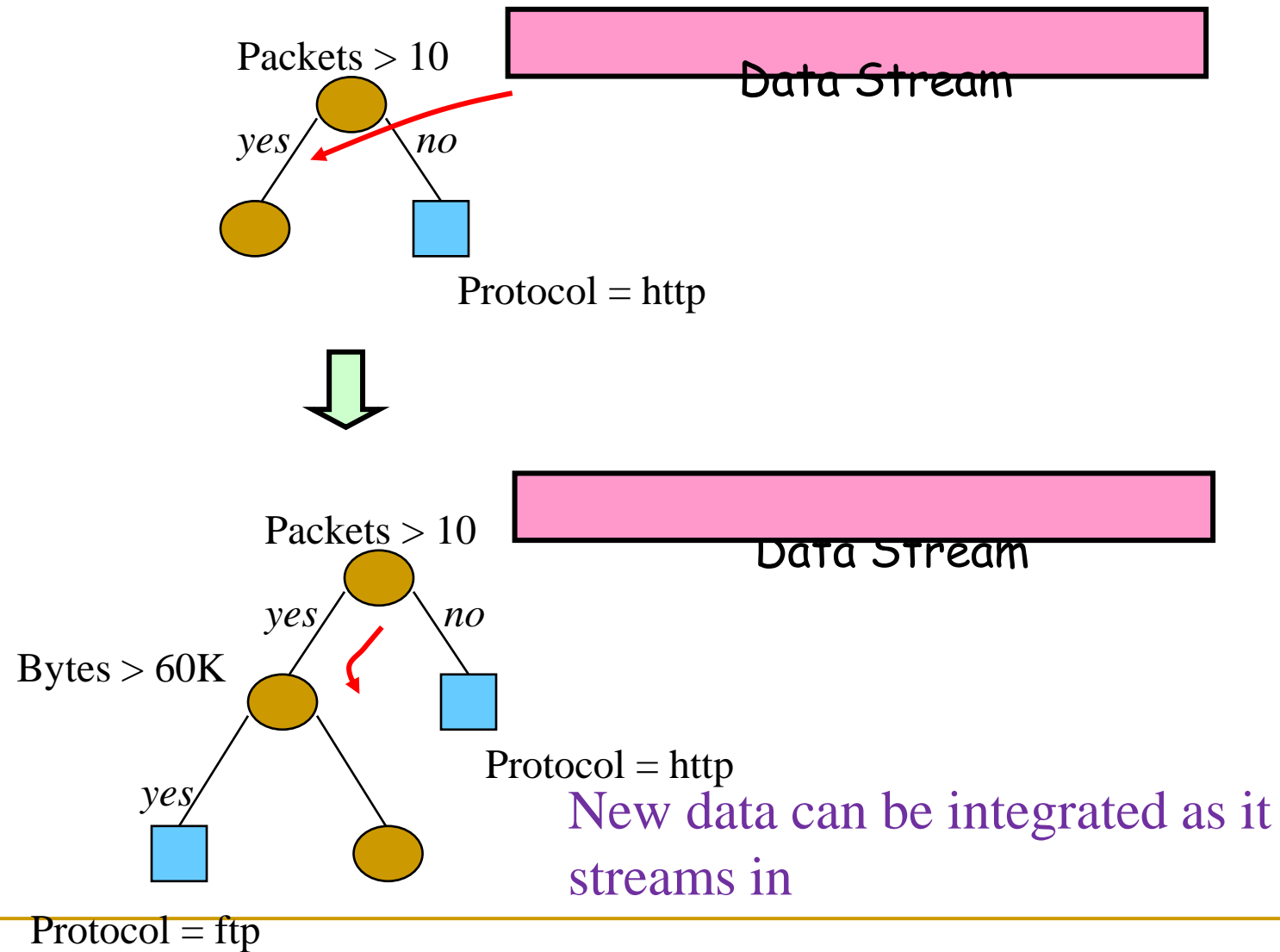
How many examples are enough?

- Let $G(X_i)$ be the heuristic measure of choice (*Information Gain, Gini Index*)
- X_a : the attribute with the highest attribute evaluation value after n examples
- X_b : the attribute with the second highest split evaluation function value after n examples
- We can compute

$$\Delta\bar{G} = \bar{G}(X_a) - \bar{G}(X_b) > \epsilon$$

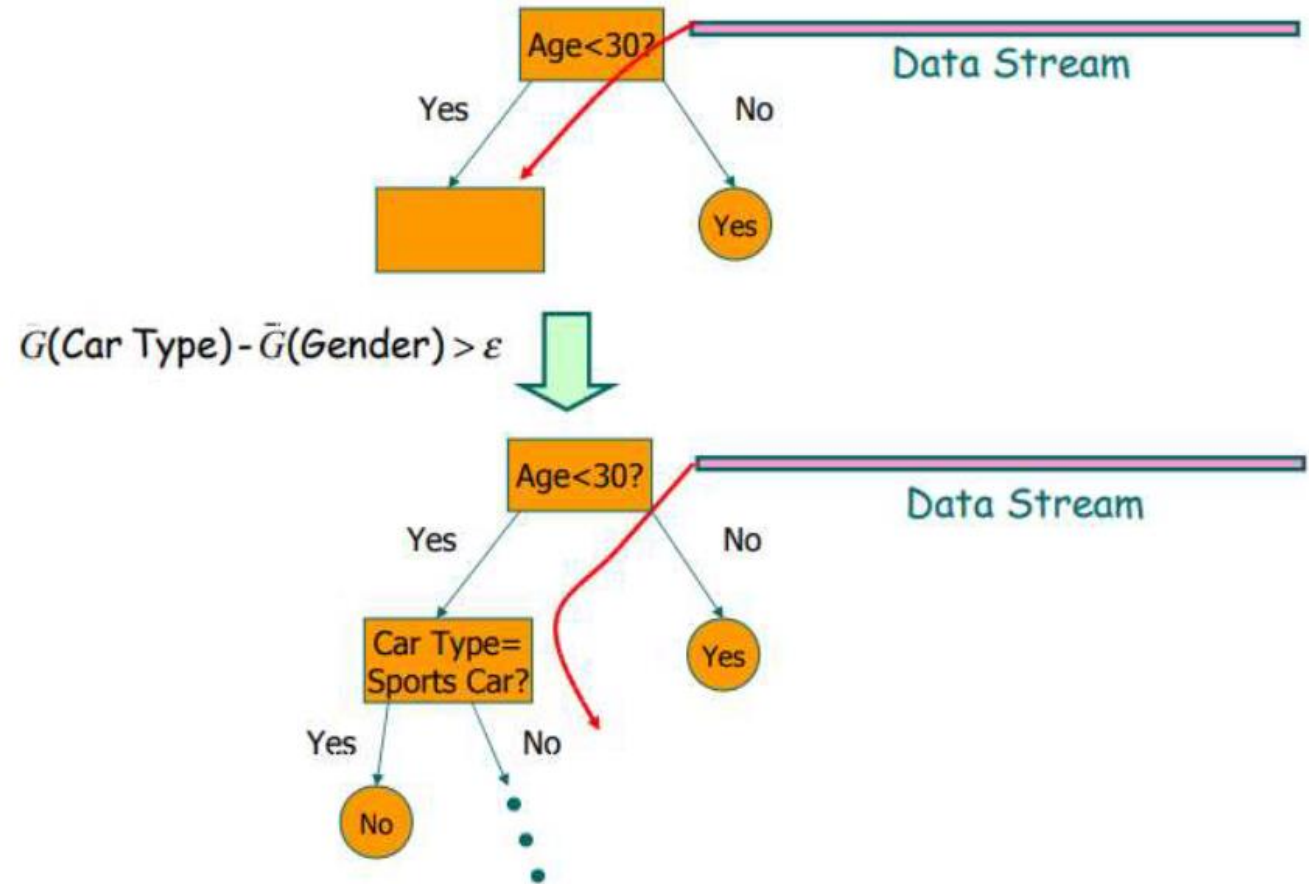
- Thanks to Hoeffding Bound, we can infer that:
 - $\Delta G \geq \Delta\bar{G} - \epsilon > 0$ with probability $1 - \delta$, where ΔG is the true difference in heuristic measure
 - This means that we can split the tree using X_a , and the succeeding examples will be passed to the new leaves (incremental approach)

Hoeffding Tree Algorithm



HT Algorithm

- Compute the heuristic measure for the attributes and determine the best two attributes
- At each node check for the condition
$$\Delta \bar{G} = \bar{G}(X_a) - \bar{G}(X_b) > \epsilon$$
- If *true*, create child nodes based on the test at the node; else, get more examples from stream.



Hoeffding Tree Algorithm

■ Strengths

- ❑ Scales better than traditional methods
 - Sub-linear with sampling
 - Very small memory utilization
- ❑ Incremental
 - Make class predictions in parallel
 - New examples are added as they come

■ Weakness

- ❑ Could spend a lot of time with ties
- ❑ Memory used with tree expansion
- ❑ Cannot handle concept drift

Very Fast Decision Tree

- The VFDT (Very Fast Decision Tree) algorithm makes several modifications to the Hoeffding tree algorithm to improve both speed and memory utilization.
- **Modifications to Hoeffding Tree**
 - Near-ties broken more aggressively
 - G computed every n_{\min} after a number of training examples
 - deactivating the least promising leaves whenever memory is running low,
 - dropping poor splitting attributes
- Compare to Hoeffding Tree: **Better time and memory**
- Compares extremely well to traditional classifiers in both speed and accuracy
 - Similar accuracy
 - Better runtime
- Still **does not handle concept drift**

Concept-adapting VFDT

- **Concept Drift** (*what can we do to manage concept drift?*)
 - Time-changing data streams
 - *Need a way to identify in a timely manner those elements of the stream that are no longer consistent with the current concepts.*
 - *A common approach is to use a sliding window.*
 - *The intuition behind it is to incorporate new examples yet eliminate the effects of old ones*
 - *Repeatedly apply a traditional classifier to the examples in the sliding window.*
 - *As new examples arrive, they are inserted into the beginning of the window; a corresponding number of examples is removed from the end of the window, and the classifier is reapplied.*
 - Sliding window – sensitive to window size (w)
 - *If w is too large, the model will not accurately represent the concept drift.*
 - *if w is too small, then there will not be enough examples to construct an accurate model.*
 - *It will become very expensive to continually construct a new classifier model.*

Concept-adapting VFDT

- To adapt to concept-drifting data streams, the VFDT algorithm was further developed into the Concept-adapting Very Fast Decision Tree algorithm (CVFDT).
- **CVFDT**
 - Uses Sliding window – but does not construct model from scratch each time
 - It updates statistics at the nodes by incrementing the counts associated with new examples and decrementing the counts associated with old ones
 - If there is a concept drift, some nodes may no longer pass the Hoeffding bound.
 - When this happens, an alternate subtree will be grown, with the new best splitting attribute at the root.
 - As new examples stream in, the alternate subtree will continue to develop, without
 - yet being used for classification. Once the alternate subtree becomes more accurate than the existing one, the old subtree is replaced.
- *Empirical studies show that CVFDT achieves better accuracy than VFDT with time-changing data streams.*
- *The size of the tree in CVFDT is much smaller than that in VFDT, because the latter accumulates many outdated examples*

Classifier Ensemble Approach to Stream Data Classification

- *Classifier ensemble - idea is to train an ensemble or group of classifiers (using, say, C4.5, or naïve Bayes) from sequential chunks of the data stream*
- Method (derived from the ensemble idea in classification)
 - train K classifiers from K chunks
 - for each subsequent chunk
 - train a new classifier
 - test other classifiers against the chunk
 - assign weight to each classifier
 - select top K classifiers
- Instead of discarding old examples – here **least accurate classifiers are discarded**

Clustering Data Streams

- Applications that require the automated clustering
 - Network intrusion detection
 - Analyzing Web clickstreams
 - Stock market analysis

The data stream model of computation requires algorithms to make a single pass over the data, with bounded memory and limited processing time, whereas the stream may be highly dynamic and evolving over time

Clustering Data Streams

For effective clustering of stream data, several new methodologies have been developed

■ Methodologies used:

- ❑ Compute and store summaries of past data
 - compute summaries of the previously seen data, store the relevant results, and use such summaries to compute important statistics when required.
- ❑ Apply a divide-and-conquer strategy
 - Divide data streams into chunks based on order of arrival
 - Compute summaries for these chunks, and then merge the summaries.
 - Larger models can be built out of smaller building blocks.
- ❑ Incremental Clustering of incoming data streams
 - Because stream data enter the system continuously and incrementally, the clusters derived must be incrementally refined.

Clustering Data Streams

- ❑ Methodologies used:
 - ❑ Perform micro-clustering as well as macro-clustering analysis
 - Stream clusters can be computed in two steps:
 - (1) compute and store summaries at the microcluster level (using hierarchical bottom-up clustering algorithm)
 - (2) compute macroclusters (using another clustering algorithm to group the microclusters) at the user-specified level.
 - This approach effectively compresses the data and often results in a smaller margin of error.
 - ❑ Explore multiple time granularity for the analysis of cluster evolution – data summaries at different time points
 - More recent data often play a different role from that of the older data in stream data analysis
 - Store snapshots of summarized data at different points in time

Clustering Data Streams

- ❑ Methodologies used:
 - ❑ Divide stream clustering into on-line and off-line (or independent) processes
 - On-line process is needed to maintain dynamically changing clusters (basic summaries of data snapshots should be computed, stored, and incrementally updated)
 - Off-line : A user may pose queries to ask about past, current, or evolving clusters.

STREAM

100 – MMem – 1 bucket – K cluster, $K=5$ grouped into 5 clusters

$C1 = 50, c2 = 10, c3, c4, c5=10$ – discard 100 points – Retain – only weighted centers

1000 Streams – 1000 Buckets – 5000 weighted centers

5000 weighted centers – clustered again – 5 clusters

- **Input:** Data stream of points $X_1, X_2 \dots X_N$ with timestamps $T_1, T_2 \dots T_N$
- **Single pass, k-medians approach**
 - Clusters data such that **Sum Squared Error (SSQ)** between the points and cluster center is minimized
 - Data streams are processed in **buckets** of m points (to fit into main memory)
- For each bucket – k clusters are formed, weighted centers are retained and other points are discarded (each cluster center being weighted by the number of points assigned to its cluster)
- When enough centers are collected, these are clustered - the weighted centers are again clustered to produce another set of $O(k)$ cluster centers.
- Clusters in limited time and space but does not handle time granularity
- Data stream clustering algorithm should also provide the flexibility to compute clusters over user-defined time periods in an interactive manner

CluStream: Clustering Evolving Streams

- **Design goal**
 - High quality for clustering evolving data streams with greater functionality
 - One-pass over the original stream data
 - Limited space usage and high efficiency
- **CluStream: A framework for clustering evolving data streams**
 - Divide the clustering process into online and offline components
 - Online component: periodically stores summary statistics about the stream data
 - Offline component: answers various user questions based on the stored summary statistics

CluStream

- Tilted time frame model adopted
- **Micro-cluster**
 - Statistical information about data locality
 - Temporal extension of the *cluster-feature vector*
 - Multi-dimensional points $X_1, X_2 \dots X_N$ with timestamps $T_1, T_2 \dots T_N$
 - Each point contains d dimensions
 - A micro-cluster for n points is defined as a $(2.d + 3)$ tuple
 $(CF2^x, CF1^x, CF2^t, CF1^t, n)$
- **Pyramidal time frame**
 - Decide at what moments the snapshots of the statistical information are stored away on disk

CluStream

- **Online micro-cluster maintenance**

- Initial creation of q micro-clusters
 - q is usually significantly larger than the number of natural clusters
- Online incremental update of micro-clusters
 - If new point is within max-boundary, insert into the micro-cluster or create a new cluster
 - May delete obsolete micro-cluster or merge two closest ones

- **Query-based macro-clustering**

- Based on a user-specified time-horizon h and the number of macro-clusters K , compute macroclusters using the k-means algorithm

CluStream

■ Evolution Analysis

- Given time horizon h and two clock times t_1, t_2 , cluster analysis examines data arriving between (t_2-h, t_2) and (t_1-h, t_1)
 - Check for new clusters
 - Loss of original clusters
 - How clusters have shifted in position and nature
 - Net snapshots of microclusters $N(t_1, h)$ and $N(t_2, h)$

■ CluStream – accurate and efficient; scalable; flexible