Week 3: Stream (1)

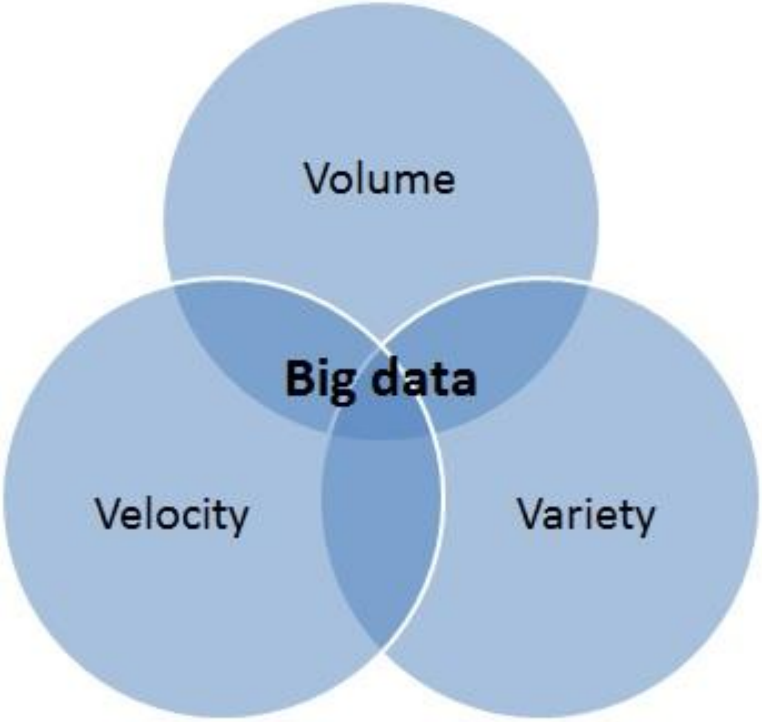
Instructor: Daejin Choi (djchoi@inu.ac.kr)

INCHEON
NATIONAL
UNIVERSITY

* Most slides are used from www.mmds.org with small updates

Static vs. Stream Data

- REMIND: Big data means not only "size", but "velocity"



Characteristics of Stream Data

- Infinite, Burst, Non-stationary
  - **We do not know the entire data set in advance**
- only **INSTANTLY** accessible
- **Stream Management** is important when the input rate is controlled **externally:**
  - Google queries
  - Twitter or Facebook status updates

Applications

- **Mining query streams**
  - Google wants to know what queries are more frequent today than yesterday
- **Mining click streams**
  - Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour
- **Mining social network news feeds**
  - E.g., look for trending topics on Twitter, Facebook
- **Sensor Networks**
  - Many sensors feeding into a central controller
- **IP packets monitored at a switch**
  - Gather information for optimal routing
  - Detect denial-of-service attacks

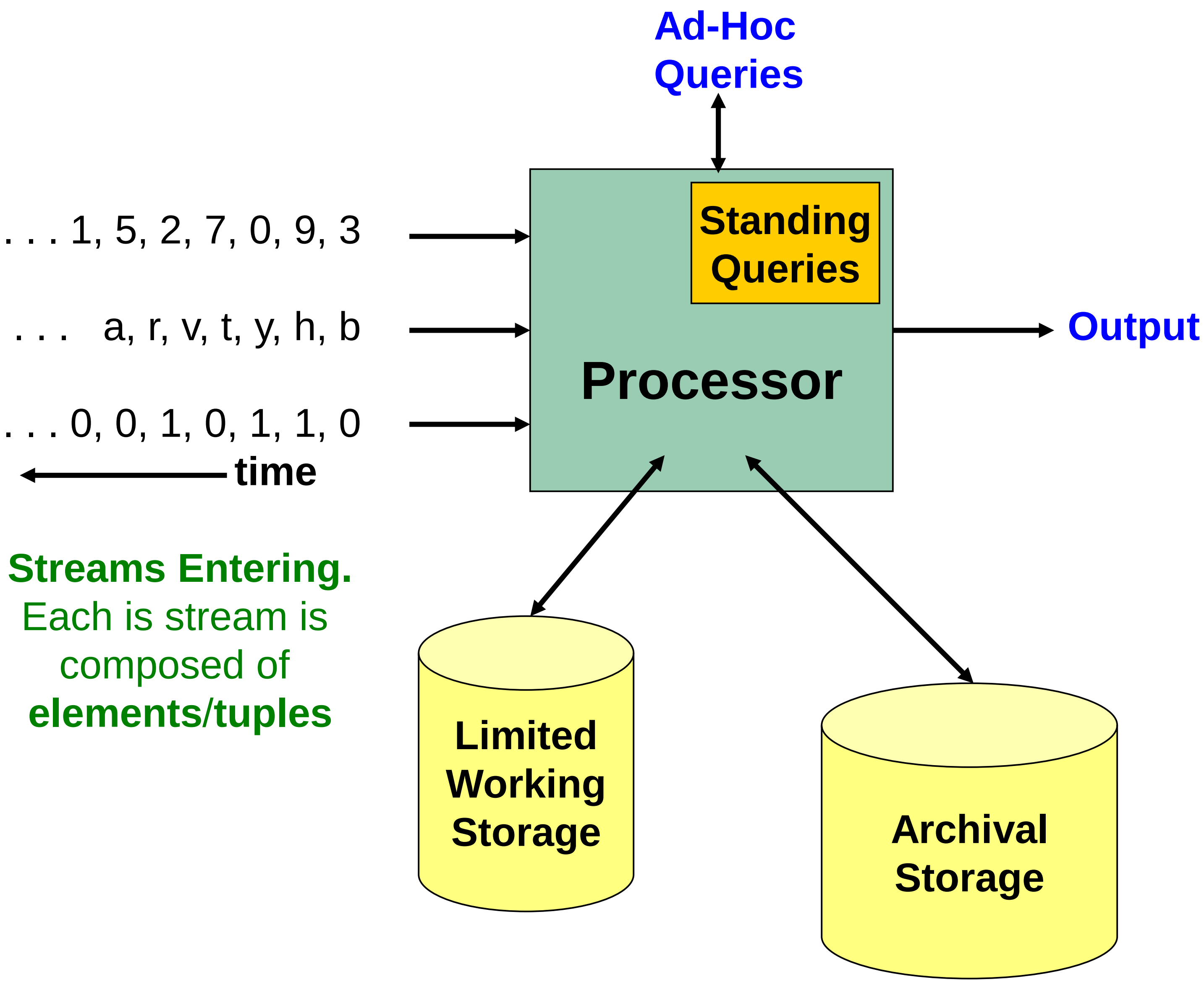
The Stream Model

- Input **elements** enter at a *rapid* rate, at one or more input ports (i.e., **streams**)
  - **We call elements of the stream tuples**

- The system **cannot** store the entire stream

- **Q:** How do you make critical calculations about the stream using a **limited amount of (secondary) memory?**

General Stream Processing Model



Streams Entering.
Each is stream is composed of
elements/tuples

Side note: Streaming Alg. in Machine Learning?

- **Online Learning** enables a machine learning model to continuously learn from the **recent data stream**
  - An algorithm to learn from it and slowly adapt to the changes in data

- **Example: Stochastic Gradient Descent (SGD)**

- **Idea: Do slow updates to the model**
  - **SGD** (SVM, Perceptron) makes small updates
  - **So:** First train the classifier on training data.
  - **Then:** For every example from the stream, we slightly update the model (using small learning rate)

Operations on Data Streams

- In conclusion, we have to choose **a subset of input streams**
  - **Sampling data from a stream**
    - Construct a random sample
  - **Queries over sliding windows**
    - Number of items of type ***x*** in the last ***k*** elements of the stream
  - **Filtering a data stream**
    - Select elements with property ***x*** from the stream
  - **Counting distinct elements**
    - Number of distinct elements in the last ***k*** elements of the stream
  - **Estimating moments**
    - Estimate avg./std. dev. of last ***k*** elements
  - …

Week 3

Week 4

Let's start from sampling!

- Sample a **fixed proportion** of elements in the stream (say 1 in 10)
- Maintain a **random sample of fixed size** over a potentially infinite stream
  - At any "time" ***k*** we would like a random sample of ***s*** elements
    - **What is the property of the sample we want to maintain?**
      : For all time steps ***k***, each element seen so far has equal prob. of being sampled

Sampling from a Data Stream: Sampling a fixed proportion

Sampling a Fixed Proportion

- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Answer questions such as: How often did a user run the same query in a single day**
  - Have space to store **1/10th** of query stream

- **Naïve solution:**
  - Generate a random integer in **[0..9]** for each query
  - Store the query if the integer is **0**, otherwise discard

Problem with Naïve Approach

- **Simple question: What fraction of queries by an user are duplicates?**
  - Suppose each user issues ***x*** queries once and ***d*** queries twice (total of ***x*+2*d*** queries)
    - **Correct answer: *d*/(*x*+*d*)**
  - **Proposed solution: We keep 10% of the queries**
    - Sample will contain ***x*/10** of the singleton queries and **2*d*/10** of the duplicate queries at least once
    - But only ***d*/100** pairs of duplicates
      - **d/100 = 1/10 · 1/10 · d**
    - Of ***d*** "duplicates" ***18d/100*** appear exactly once
      - **18d/100 = ((1/10 · 9/10)+(9/10 · 1/10)) · d**
  - **So the sample-based answer is** $\frac{\frac{d}{100}}{\frac{x}{10}+\frac{d}{100}+\frac{18d}{100}} = \frac{d}{10x+19d}$

How to solve?

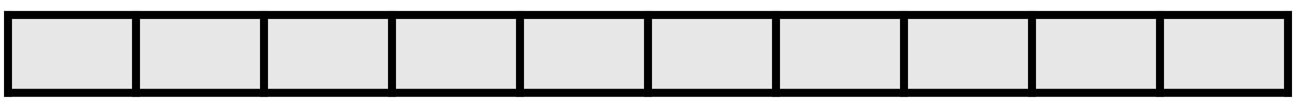
- Where the error comes from? → "Probability of selection"

- A possible solution? → Sampling user, not query
  - Pick **1/10th** of **users** and take all their searches in the sample
  - Use a hash function that hashes the user name or user id uniformly into 10 buckets

Generalized Solution

- **Stream of tuples with keys:**
  - Key is a subset of each tuple's components
    - e.g., tuple is (user, search, time); key is **user**
  - Choice of key depends on application

- **To get a sample of *a/b* fraction of the stream:**
  - Hash each tuple's key uniformly into ***b*** buckets
  - Pick the tuple if its hash value is at most ***a***



Hash table with **b** buckets, pick the tuple if its **hash** value is at most **a.**
How to generate a 30% sample?
Hash into b=10 buckets, take the tuple if it hashes to one of the first 3 buckets
(Assumption: Hash distributes users "uniformly")

Sampling from a Data Stream: Sampling a fixed-size tuples

Maintaining a fixed-size sample

- **Suppose we need to maintain a random sample *S* of size exactly *s* tuples**
  - E.g., main memory size constraint

- **Why?** Don't know length of stream in advance

- **Suppose at time *n* we have seen *s* items**
  - **Goal: Each item is in the sample *S* with equal prob. *s/n***

Illustration (s=2)

- **Stream:** a x c y z k c d e g…

- At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.
- At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.
- **Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random**

Solution: Reservoir Sampling

- **Algorithm**
  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the ***n****th* element arrives (***n* > *s***)
    - With probability ***s/n***, keep the ***n****th* element, else discard it
    - If we picked the ***n****th* element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random

- **Claim:** This algorithm maintains a sample ***S*** with the desired property:
  - After ***n*** elements, the sample contains each element seen so far with probability ***s/n***

Proof of Reservoir Sampling By Induction

- Assume that after ***n*** elements, the sample contains each element seen so far with probability ***s/n***
- We need to show that after seeing element ***n+1*** the sample maintains the property
  - Sample contains each element seen so far with probability ***s/(n+1)***
- **Base case:**
  - After we see **n=s** elements the sample **S** has the desired property
    - Each out of **n=s** elements is in the sample with probability ***s/s = 1***

Proof (cont'd)

- **Inductive hypothesis:** After ***n*** elements, the sample ***S*** contains each element seen so far with prob. ***s/n***
- **Now element *n+1* arrives**
- **Inductive step:** For elements already in ***S***, probability that the algorithm keeps it in ***S*** is:

$$\underbrace{\left(1-\frac{s}{n+1}\right)}_{\text{Element } \mathbf{n+1} \text{ discarded}}+\underbrace{\left(\frac{s}{n+1}\right)}_{\text{Element } \mathbf{n+1} \text{ not discarded}}\underbrace{\left(\frac{s-1}{s}\right)}_{\text{Element in the sample not picked}}=\frac{n}{n+1}$$

- So, at time ***n***, tuples in ***S*** were there with prob. **s/n**
- Time ***n→n+1***, tuple stayed in ***S*** with prob. **n/(n+1)**
- So prob. tuple is in ***S*** at time ***n+1*** $= \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

Queries over a (long) Sliding Window

Sliding Windows

- A useful model of stream processing is that queries are about a ***window*** of length ***N***, the ***N*** most recent elements received
- **Interesting case:** ***N*** is so large that the data cannot be stored in memory, or even on disk
  - Or, there are so many streams that windows for all cannot be stored
  - The estimation is required. We will finally cover later!

Sliding Window: 1 Stream

- **Sliding window on a single stream:**

N = 6

q w e r t y u i o p a s d f g h j k l z x c v b n m

q w e r t y u i o p a s d f g h j k l z x c v b n m

q w e r t y u i o p a s d f g h j k l z x c v b n m

q w e r t y u i o p a s d f g h j k l z x c v b n m

← Past Future →

Example Problem: Counting Bits (1)

- **Problem:**
  - Given a stream of **0**s and **1**s
  - Be prepared to answer queries of the form **How many 1s are in the last *k* bits?** where ***k*** $\leq$ ***N***

- **Obvious solution:**
  Store the most recent ***N*** bits
  - When new bit comes in, discard the ***N*+1st** bit

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 1 1 0 1 1 0

Suppose N=6

← Past Future →

Example Problem: Counting Bits (2)

- **You CANNOT get an exact answer without storing the entire window**

- **Real Problem:**
  What if we cannot afford to store *N* bits?
  - **E.g.**, we're processing 1 billion streams and
    ***N* = 1 billion**

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 ~~1 1 0 1 1 0~~

← Past Future →

- But we are happy with **an approximate answer** inferred from **the summarized results**

An attempt: Simple solution

- **Q: How many 1s are in the last *N* bits?**
- A simple solution that does not really solve our problem: **Uniformity assumption**

$\longleftarrow$ ***N*** $\longrightarrow$

0 1 0 0 1 1 1 0 0 0 1 0 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 1 0 1 0

$\longleftarrow$ Past Future $\longrightarrow$

- **Maintain 2 counters:**
  - ***S***: number of 1s from the beginning of the stream
  - ***Z***: number of 0s from the beginning of the stream
- **How many 1s are in the last N bits?** $N \cdot \frac{S}{S+Z}$
- **But, what if stream is non-uniform?**
  - What if distribution changes over time?

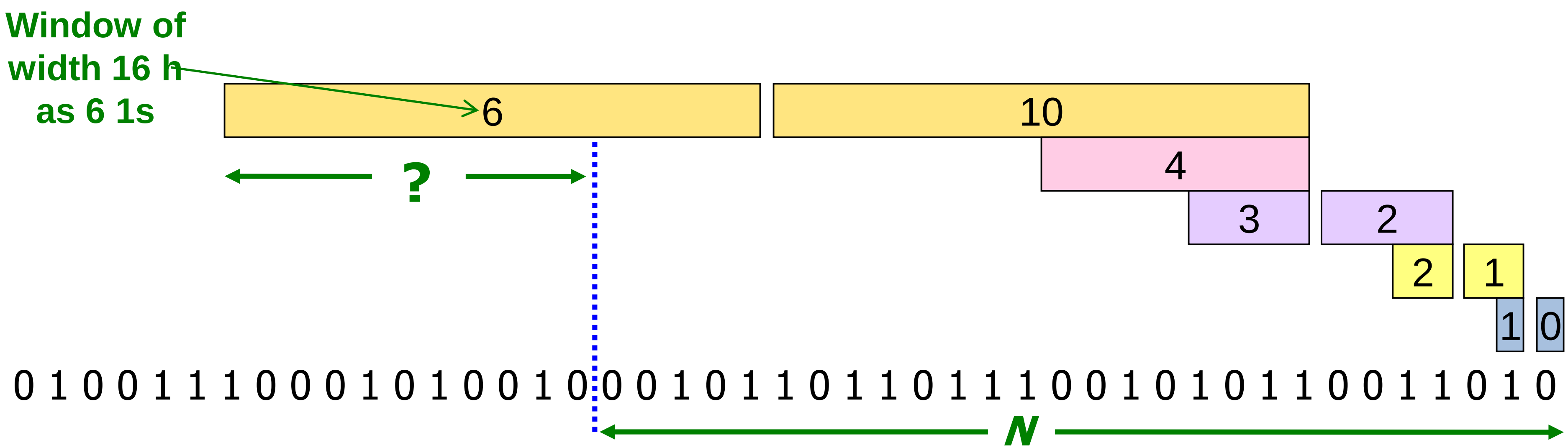
DGIM Method

[Datar, Gionis, Indyk, Motwani]

- **DGIM solution that does <u>not</u> assume uniformity**
- We store $O(\log^2 N)$ bits per stream
- **Solution gives approximate answer, never off by more than 50%**
  - Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits

Beginning Idea: Exponential Windows

- Summarize **exponentially increasing** regions of the stream, looking backward
- Drop small regions if they begin at the same point as a larger region



We can reconstruct the count of the last ***N*** bits, except we are not sure how many of the last **6 1s** are included in the ***N***

What's Good?

- **Stores only O(log²*N*) bits**
  - $O(\log N)$ counts of $\log_2 N$ bits each

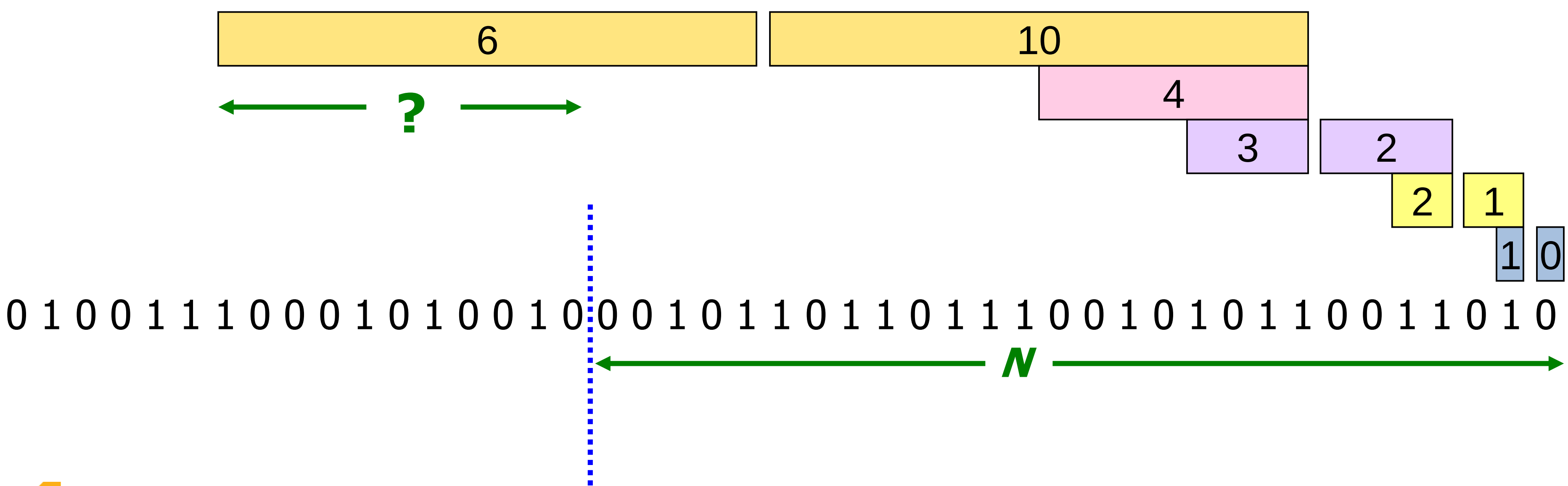
    Window counts   Window size

- **Easy update as more bits enter**

- Error in count no greater than the number of **1s** in the "**unknown**" area

What's Not So Good?

- As long as the **1s** are fairly evenly distributed, the e rror due to the unknown region is small – **no more than 50%**
- But it could be that all the **1s** are in the unknown ar ea at the end
- In that case, **the error changes w.r.t. N!**



Fixup: DGIM method

[Datar, Gionis, Indyk, Motwani]

- **Intuition:** We do not have to count both 0s and 1s

- **Idea:** Instead of summarizing fixed-length blocks, summarize **blocks with specific number of 1s**:
  - Let the block ***sizes*** (number of **1s**) increase exponentially

1001010110001011 0 1010101010101011 0 1010101010101110 1010101 110101 000 101 1 00 1 0

N

- **When there are few 1s in the window, block sizes stay small, so errors are small**

DGIM: Timestamps

- Each bit in the stream has a ***timestamp***, starting **1**, **2,** …

- Record timestamps modulo ***N*** (**the window size**), so we can represent any **relevant** timestamp in $O(log_2 N)$ bits
  - Store ***N*** = 4 using 2 bits

DGIM: Buckets

- A ***bucket*** in the DGIM method is a record consisting of:
  - **(A) The timestamp of its end [O(log *N*) bits]**
  - **(B) The number of 1s between its beginning and end [O(log log *N*) bits]**

- **Constraint on buckets:**
  Number of **1s** must be a power of **2**
  - That explains the **O(log log *N*)** in **(B) above**

1001010110001011010101010101011101010101010111010101011101010001011 0010

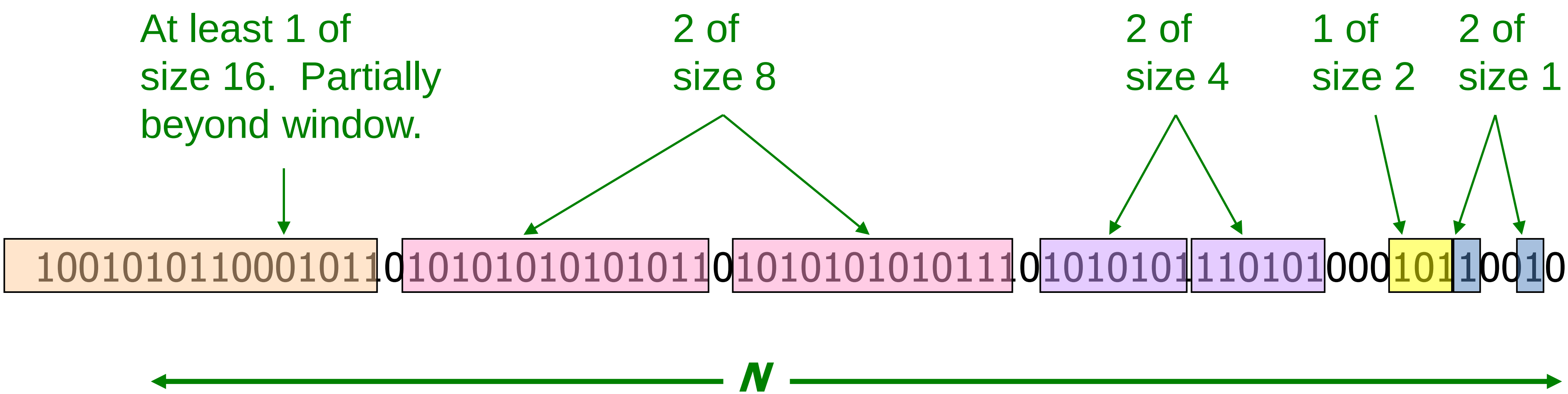
← ***N*** →

INU 인천대학교 INCHEON NATIONAL UNIVERSITY

More Constraints

- Either **one** or **two** buckets with the same **power-of-2 number** of **1s**
- **Buckets do not overlap in timestamps**
- **Buckets are sorted by size**
  - Earlier buckets are not smaller than later buckets
- Buckets disappear when their end-time is **>** ***N*** time units in the past

Example: Representing a Stream by Buckets



Three properties of buckets that are maintained:

- Either **one** or **two** buckets with the same **power-of-2** number of **1s**
- Buckets do not overlap in timestamps
- Buckets are sorted by size

Updating Buckets (1)

- When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to ***N*** time units before the current time

- **2 cases:** Current bit is **0** or **1**

- **If the current bit is 0:**
  no other changes are needed

Updating Buckets (2)

- **If the current bit is 1:**

(1) Create a new bucket of size **1**, for just this bit
  - **End timestamp = current time**

(2) If there are now **three buckets of size 1**, **combine the oldest two into a bucket of size 2**

(3) If there are now **three buckets of size 2**, **combine the oldest two into a bucket of size 4**

(4) And so on …

Example: Updating Buckets

Current state of the stream:

1001010110001011 0 10101010101011 0 10101010101110 1010101 110101 000 101 100 10

Bit of value 1 arrives

0010101100010110 10101010101011 0 10101010101110 1010101 110101 000 101 100 101

Two orange buckets get merged into a yellow bucket

0010101100010110 10101010101011 0 10101010101110 1010101 110101 000 101 1001 01

Next bit 1 arrives, new orange bucket is created, then 0 comes, then 1:

0101100010110 10101010101011 0 10101010101110 1010101 110101 000 101 1001 01 1 **01**

Buckets get merged…

0101100010110 10101010101011 0 10101010101110 1010101 110101 000 101 1001 01 1**01**

State of the buckets after merging

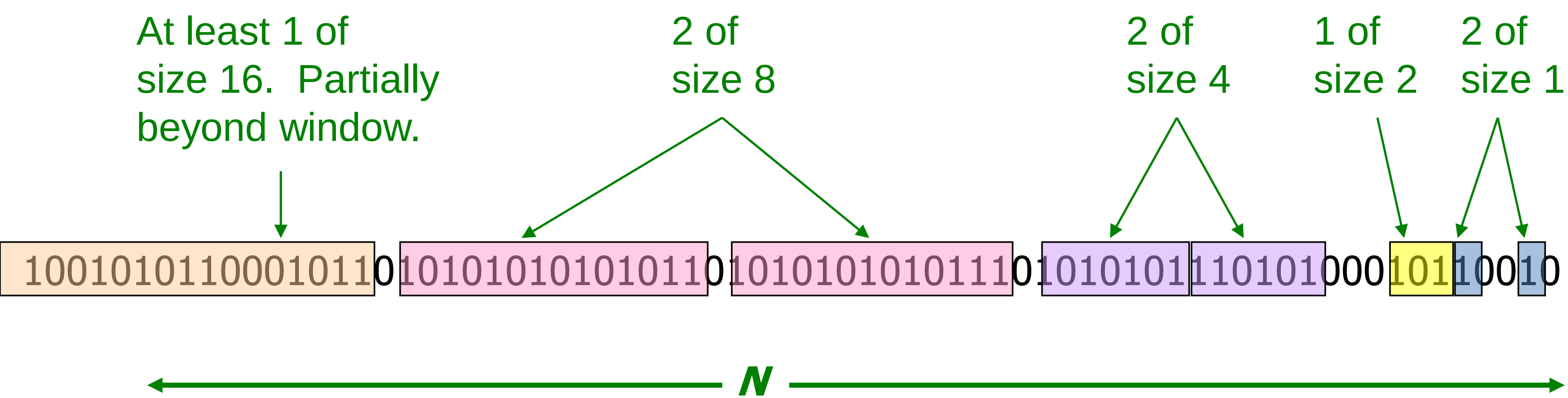
0101100010110 1010101010101110101010101110 1010101110101 000 1011001 0 110 1

Return to Our Original Question

- **To estimate the number of 1s in the most recent *N* bits:**
  1. **Sum the sizes of all buckets but the last**
     (note "size" means the number of 1s in the bucket)
  2. **Add half the size of the last bucket**

- **Remember:** We do not know how many **1s** of the last bucket are still within the wanted window

Example: Bucketized Stream



How can the error be bounded?

- **Why is error 50%? Let's prove it!**

At least 16 1s

11111111000000001 1101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0

N

- Suppose the last bucket has size $\mathbf{2^{r}}$
- Then by assuming $\mathbf{2^{r-1}}$ (i.e., half) of its **1s** are still within the window, we make an error of at most $\mathbf{2^{r-1}}$
- Since there is at least one bucket of each of the sizes less than $\mathbf{2^{r}}$, the true sum is at least
  $\mathbf{1 + 2 + 4 + .. + 2^{r-1} = 2^{r}-1}$
- Thus, error at most **50%**

Further Reducing the Error

- Instead of maintaining **1** or **2** of each size bucket, we allow either ***r*-1** or ***r*** buckets (***r* > 2**)
  - Except for the largest size buckets; we can have any number between **1** and ***r*** of those

- **Error is at most *O*(1/*r*)**

- By picking ***r*** appropriately, we can tradeoff between **number of bits we store** and **the error**

Extensions

- Can we use the same trick to answer queries **How many 1's in the last *k*?** where ***k*** **<** ***N***?
  - **A:** Find earliest bucket **B** that at overlaps with ***k***. Number of **1s** is the **sum of sizes of more recent buckets + ½ size of B**

10010101100010110101010101010110101010101010111010101011101010001011001 0

k

Summary

- **Sampling a fixed proportion of a stream**
  - Sample size grows as the stream grows
- **Sampling a fixed-size sample**
  - Reservoir sampling
- **Counting the number of 1s in the last N elements**
  - Exponentially increasing windows → DGIM method

Thank you!

Instructor: Daejin Choi (djchoi@inu.ac.kr)