ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Stream Analytics

Yannis Kotidis

http://pages.cs.aueb.gr/~kotidis/

Stream Data Challenges

- Conventional (static) algorithms assume that data is available when we want it
- In a (pure) stream processing scenario, data arrives in streams and if not processed immediately or stored, then it is lost forever
- Main challenges: number of streams * velocity
  - Data arrives so rapidly that it is not feasible to store it all in memory or in a database to query it in real time
  - Even if a single stream is slow, there can be thousands of such steams in a large-scale application

Example: Gas Turbines Monitoring
[Optique FP7]

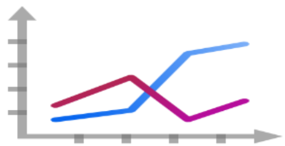
- 950 power generating turbines located across the globe
  - 100K sensors installed
  - Hundreds of TB worth of readings
- Detect in real-time undesirable patterns
  - Single-stream processing
  - Multi-stream processing
  - Live stream + archived stream correlation

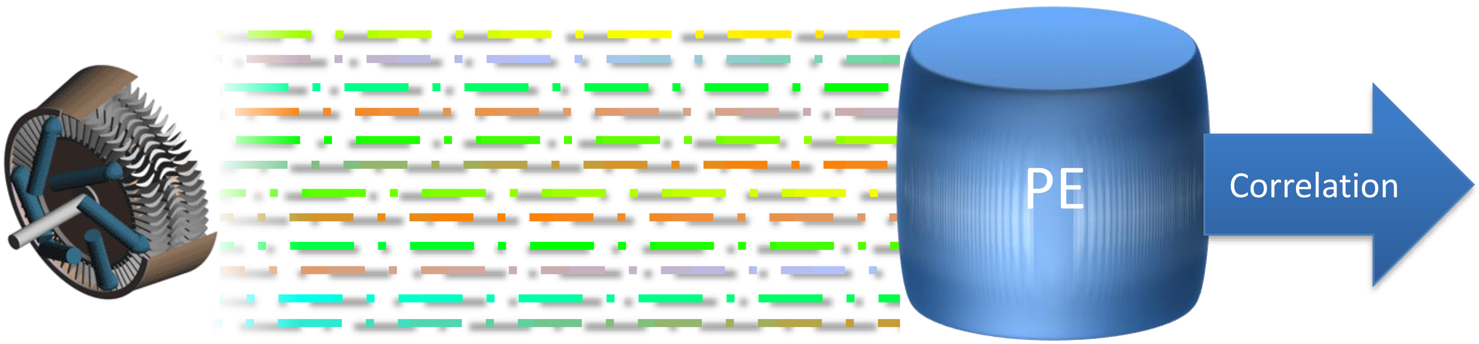
Optique

Turbine monitoring

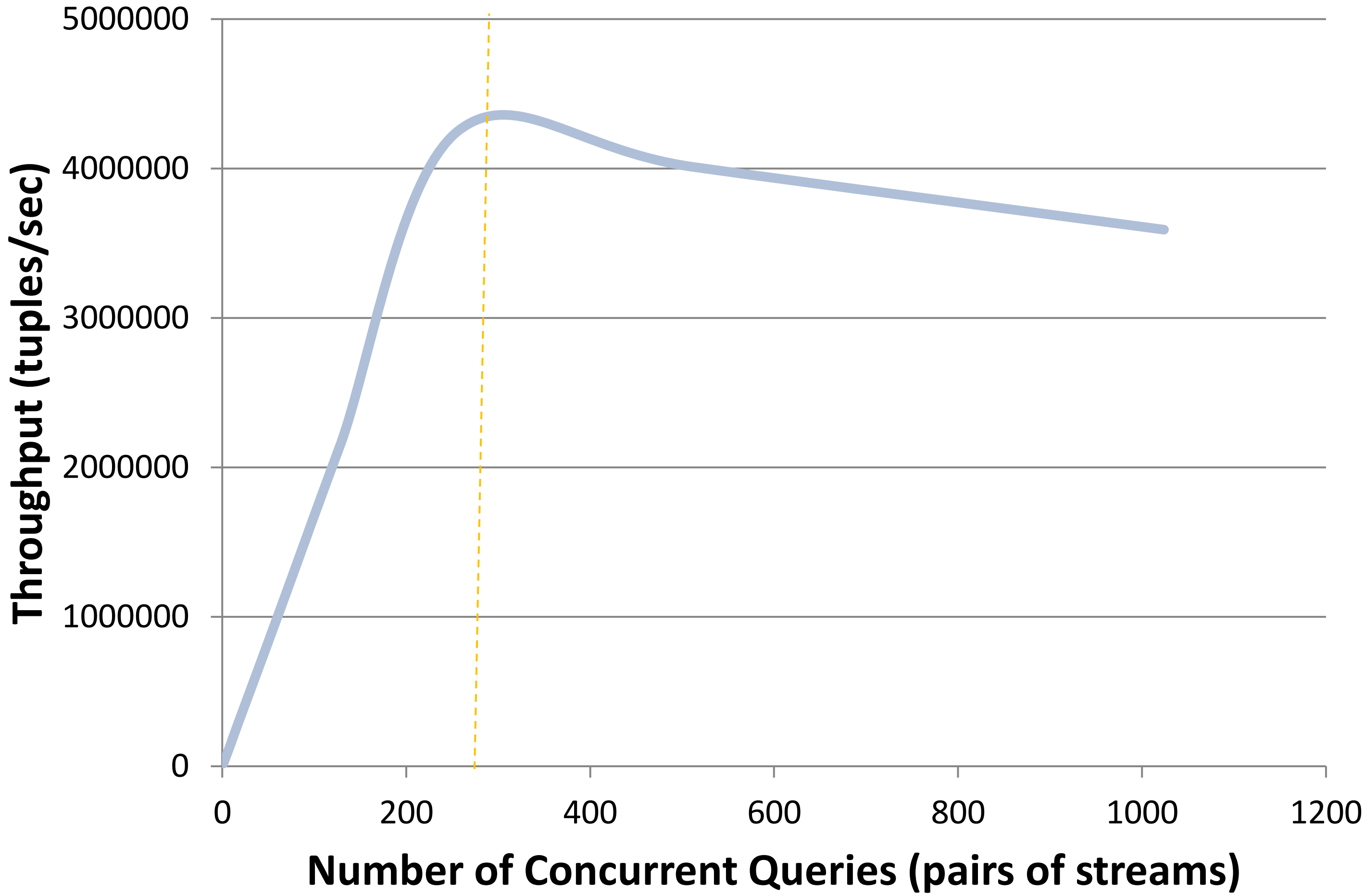
- Each Correlation query:
  - Intercepts two streams
  - Groups measurements over specified windows
  - Joins streams, computes Pearson coefficient:

$$Pearson(u_i, u_j) = cov(u_i, u_j) / (\sigma_{u_i} * \sigma_{u_j})$$





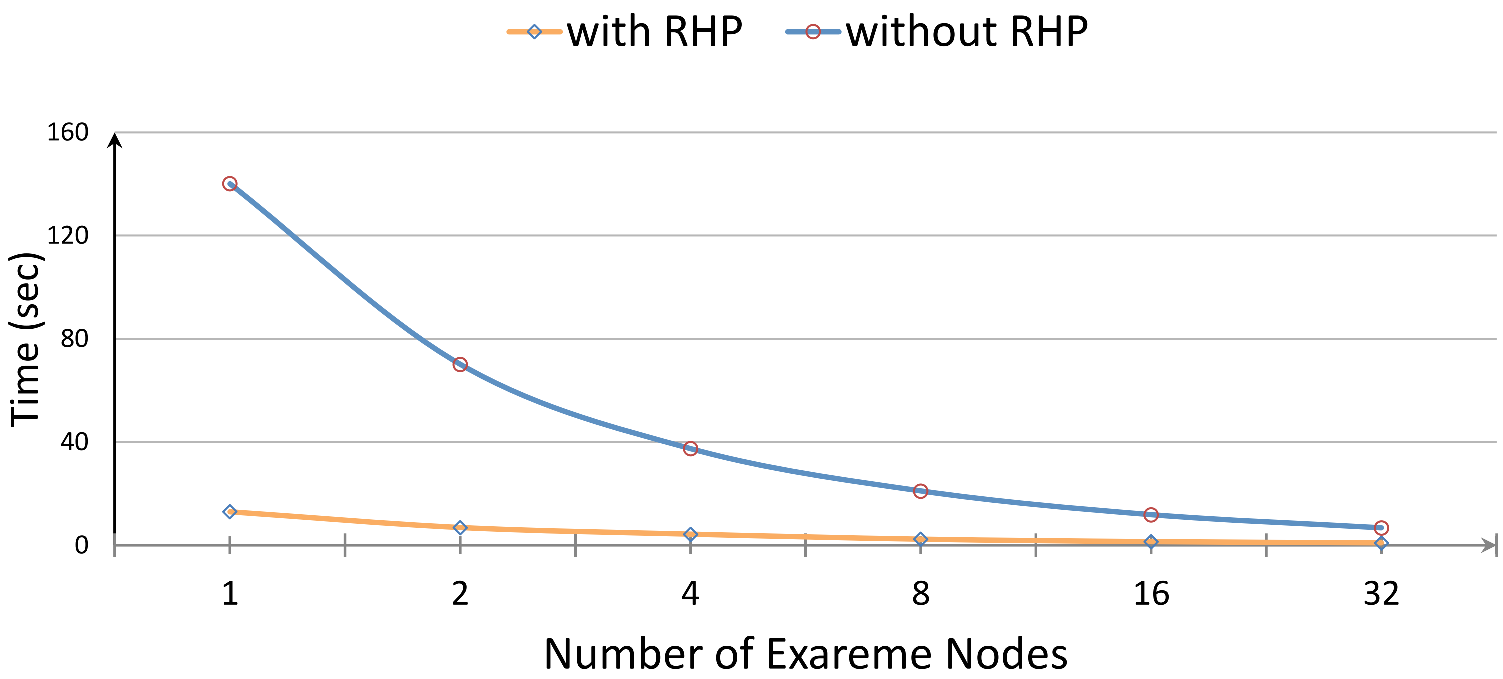
Throughput on a 256-core Exareme* cluster



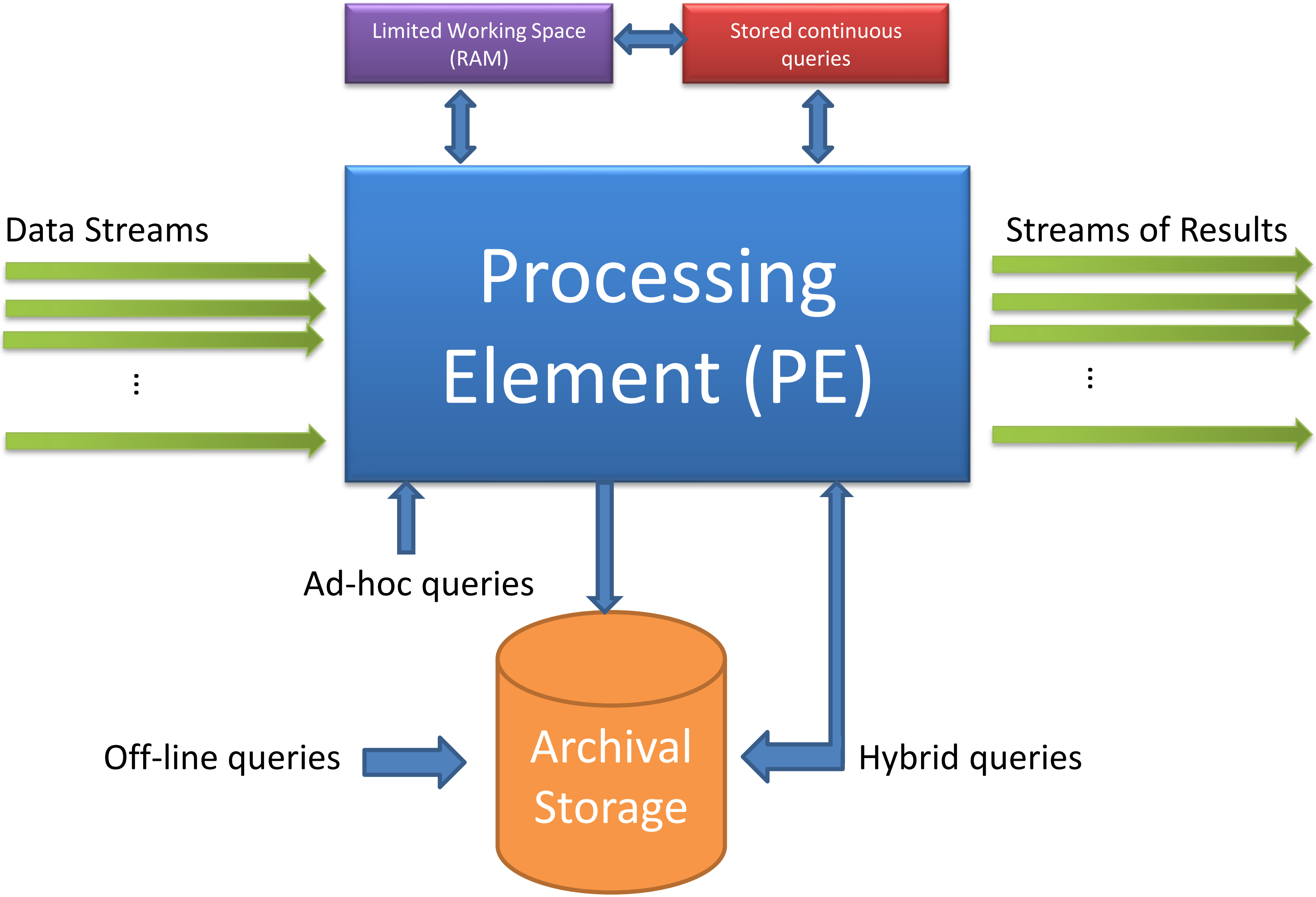
*http://madgik.github.io/exareme/

Speed-up via LSH

- Corr. between current window and 100K archived ones [ISWC 2016, BigData 2016]



Data Stream Processing



Static and stream data processing

- E.g. compute correlation between the *current state of a stream* and its *past states stored in archive storage*

Ad-hoc query example

- Queries on a search engine
  - Stream of tuples <user, term, timestamp>

- Simplification (for the shake of this running example): a user may ask the same query (term) once or twice

- Want to compute the fraction of duplicate queries issued by a typical user

Query Stream

(showing one user for simplicity)

- User makes 6 searches
  - 5 unique terms
- One duplicate term ("Real")
- fraction of duplicate queries: $\frac{1}{5}$

User	Term	Timestamp
user1	Barca	t1
user1	Real	t2
user1	Liverpool	t3
user1	Porto	t4
user1	Real	t5
user1	Panathinaikos	t6

Sampling from a data stream

- Keep a 10% sample of the stream
  - E.g. draw a random integer x in range (0..9). Then keep tuple if x = 0
- For a typical user, we want to compute the fraction of duplicate queries from the sample
- Assume a user make s one-time searches and d duplicate searches
  - Correct answer is $\frac{d}{s+d}$
  - $\frac{1}{4+1} = 20\%$ in the previous example

Using the sample

- Look at the sample to determine duplicates
  - Let s’ be the number of unique queries, for a user
  - Let d’ be the number of duplicates found, for a user
  - Report d’/(s’+d’)

- Is this correct?

Sample 50% of user searches (rows in green color)

- Real fraction of duplicate queries: $\frac{1}{5}$
- Estimate = ?

User	Term	Timestamp
user1	Barca	t1
user1	Real	t2
user1	Liverpool	t3
user1	Porto	t4
user1	Real	t5
user1	Panathinaikos	t6

Sampling unique queries

- Let s be the number of unique searches a user makes

- Prob. of keeping a query in the sample is $\frac{1}{10}$

- Thus, these unique searches appear $\frac{s}{10}$ times in the sample

Sampling duplicate queries

- Let d be the number of duplicate searches a user makes

- A duplicate search appears twice in the sample with probability: $\frac{1}{10} * \frac{1}{10} = \frac{1}{100}$

Sampling duplicate queries

- A duplicate search appears once in the sample with probability: $\underbrace{\frac{1}{10} * \frac{9}{10}}_{\text{Sample only 1st occurrence}} + \underbrace{\frac{9}{10} * \frac{1}{10}}_{\text{Sample only 2nd occurrence}} = \frac{18}{100}$

- A duplicate search does not appear in the sample with probability: $\frac{9}{10} * \frac{9}{10} = \frac{81}{100}$

In conclusion

- One-time queries in the sample
  - $s' = \frac{s}{10} + \frac{18d}{100} = \frac{10s+18d}{100}$
- Duplicate queries in the sample
  - $d' = \frac{d}{100}$
- Our estimate is d’/(s’+d’) = d/(10s+19d)
- Notice that this is vastly different than d/(s+d)

Under-estimation

s	d	Fraction d/(s+d)	Estimate d/(10s+18d)
95	5	5%	0.5%
90	10	10%	0.9%
85	15	15%	1.3%
80	20	20%	1.7%
75	25	25%	2.0%
. . .	. . .	. . .	. . .
5	95	95%	5.1%

Obtaining a Representative Sample

- As shown a random sample from all users is not representative of the average behavior

- Alternative idea: select 10% of the users and keep all their queries
  - Select these users at random
  - Do not store searches from users not in the sample

User selection

- Incoming stream tuple <user, term, time>
- Let h(x) be a hash function returning values in the range (0..9)
- Keep tuple if h(user) = 0

Maintaining fixed sample size

- In the previous example we keep about 10% of the searches
- Recall that stream is (in theory) infinite
  - Thus, the sample keeps growing
  - Also recall that we do not have control over the input stream. System may exhibit bursts of heavy usage
- How to keep the sample size memory bound?

Hashing to the rescue

- Let h(x) return values in the range (0..B-1) for some very large value B
- Keep <user,term,time> in the sample if h(user) ≤ k, for some constant k≤B,
  - Store <h(user),user,term,time> in memory
  - Possibly index by h(user)
- If memory is full, reduce value of k
  - discard samples with h(user)>k

STREAM FILTERING

Applying filters on streams

- Often the selection criterion can be calculated from the stream tuple
  - Does the query term contain > 5 characters?
    - Easy to compute: length(term) > 5
- In other cases the selection criterion involves lookup for membership in a set
  - Problem becomes hard when this set is very large
  - Is the query term a “bad” word

Membership Test: Motivational Example

- Have 1 billion bad URLs you would like to block ($n=10^9$)
  - each URL is ~50 characters long
  - Need >50GB to keep all in main memory
- Would like to block a URL request in real time if it belongs to the black list

Membership test: Bloom Filters

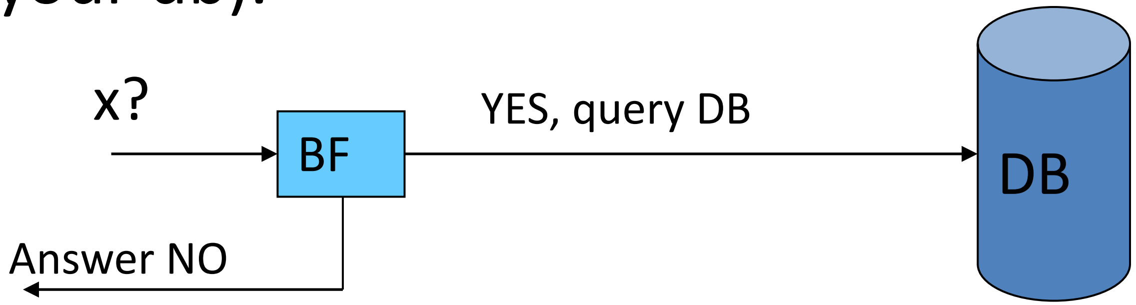
- Be able to quickly test where key value x is part of a set S
- Application: spam filtering
  - Have a set S of one billion valid email addresses (white list) for spam filtering
  - Assume 20 bytes per email address. S does not fit in memory
  - Want a memory resident data structure that will tell us whether an incoming email is spam or not

Spam Filtering

- Bloom filter will check whether an incoming email is from a valid email address in the white list
- If the answer is no then the email is guaranteed to be spam and is thus rejected
- If the answer is yes, the email is with high probability in the list
  - Cases where the filter says "yes" while the true answer is "no" are termed false positives

More applications of Bloom Filters

- Web-crawler: avoid visiting same page twice
- High-traffic on-line music store with millions of titles
  - only fetch song information when you know the song exists in your collection (minimize #queries to your db).



Problem Statement

- Have a very large set S
- Membership test: is x part of S?
- Want a data structure that
  - Is small (can fit in memory, when S cannot)
  - Requires a (small) constant time for look-ups
  - Guarantees no false negatives
  - Introduces a limited number of false positives
    - For those cases you can optionally look up x in S in a second step
      - This works only if answering "yes" happens infrequently

Bloom Filter

- Use bitmap of length m and k hash functions
  - Each $h_i(x)$ maps x to [0..m-1]
- Initially, all bits are zero

Initially Empty Bloom Filter (m=12)

0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Training (using 3 hash functions)

Insert “apples”

- h_1("apples") = 3
- h_2("apples") = 11
- h_3("apples") = 10

} set corresponding bits

BITMAP (after insertion of “apples”)

0	0	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Train with more data

BITMAP (apples)

0	0	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Now insert “oranges”

- h_1("oranges ") = 10
- h_2("oranges ") = 1
- h_3("oranges ") = 5

BITMAP (apples+oranges)

collision

0	1	0	1	0	1	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Querying: Membership test

- All bits indicated by $h_i(x)$ must be set
  - h_1(“bananas") = 10
  - h_2(“bananas") = 5
  - h_3(“bananas") = 7

Is “bananas” part of my data?

BITMAP

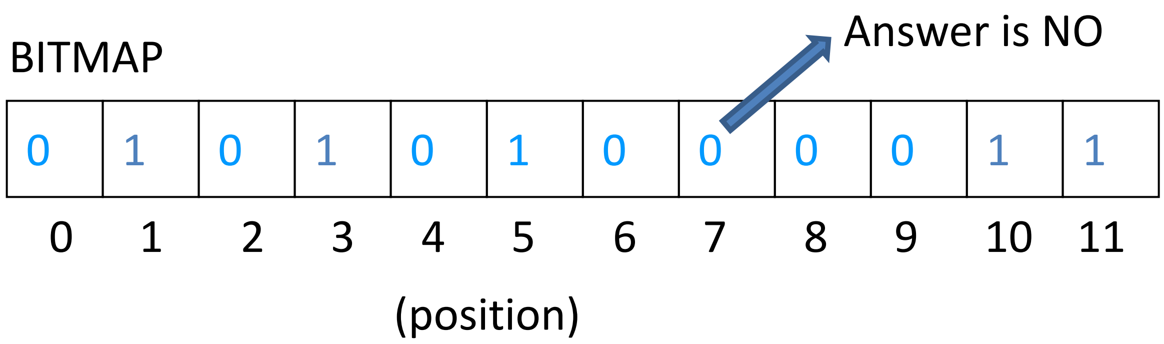
0	1	0	1	0	1	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Querying: Membership test

- All bits indicated by $h_i(x)$ must be set
  - h_1(“bananas") = 10
  - h_2(“bananas") = 5
  - h_3(“bananas") = 7

Is “bananas” part of my data?



What can we guarantee?

- No false negatives (why?)
- Small probability of false positives

$$(1-(1-1/m)^{kn})^k$$

- False positive when all k bits are set for an item we have not seen
  - A bit is set with probability 1/m assuming ideal hash function
  - $(1-1/m)^k$ = probability a bit is not set after one insertion
  - $(1-1/m)^{kn}$ = probability that a bit is not set after n insertions

Running Example

- Have 1 billion bad URLs you would like to block ($n=10^9$)
  - each URL is ~50 characters long
  - Need >50GB to keep all in main memory
- Use a bitmap of 8 billion entries ($m=8*10^9$)
  - hash table takes 1GB of memory
- For k=6, probability of false positives = $(1-(1-1/(8*10^9))^{6*10^9})^6$ =2.1%

Dependency on k

more hash functions

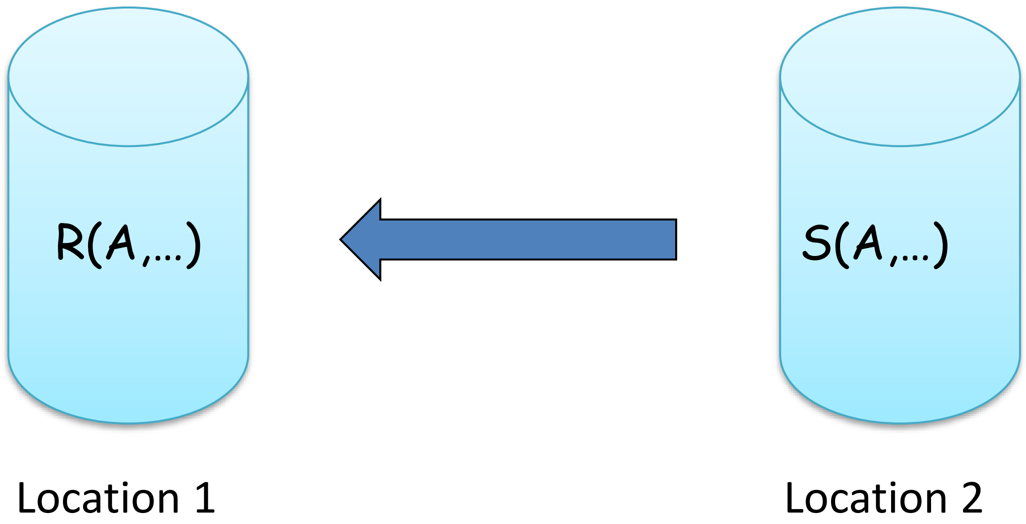
k	False positives Probability
1	12%
2	5%
3	3%
4	2.4%
5	2.2%
6	2.1%
7	2.3%
8	2.5%
9	3%

Bloom Filters in Distributed Databases

- Suppose we want to join two tables R(A,...) and S(A,...) that reside on two distant locations
  - Join result can be computed at either location

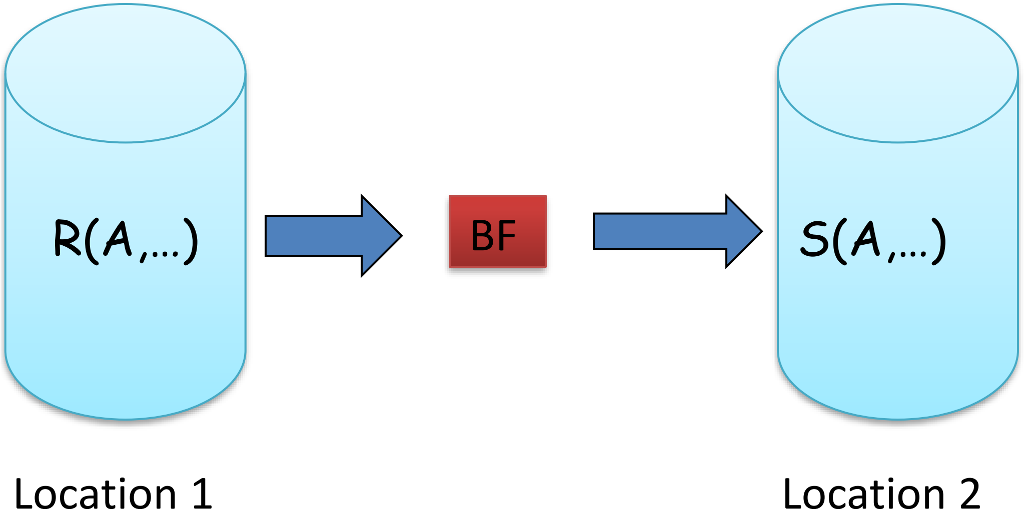
Idea 1: Ship smallest relation to the other side

- Suppose S is smaller
- Communication Cost = size(S)
- Can we do better?



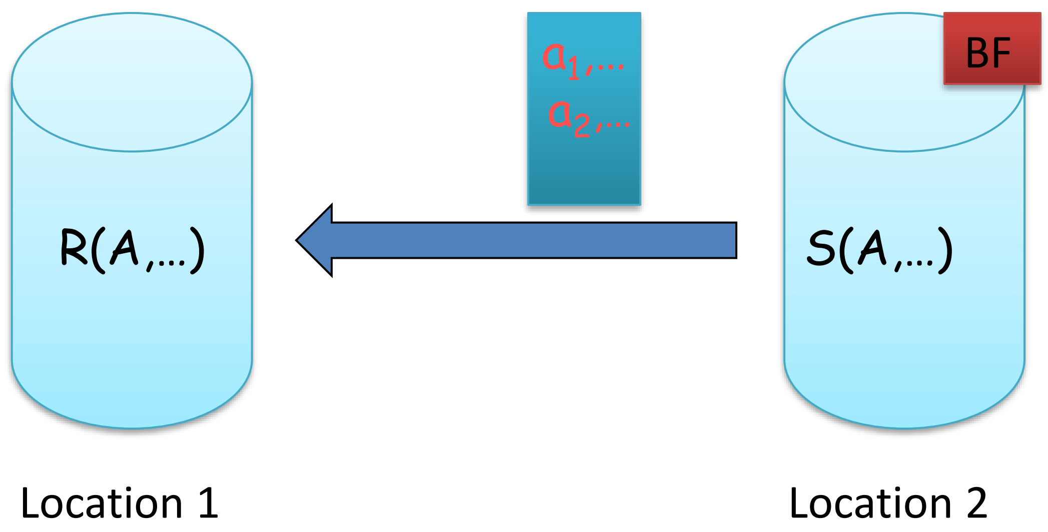
Idea 2: Step 1

- Build BF on the values of R.A
- Ship BF to location 2
  - Recall that size(BF) << size(R)



Idea 2: Step 2

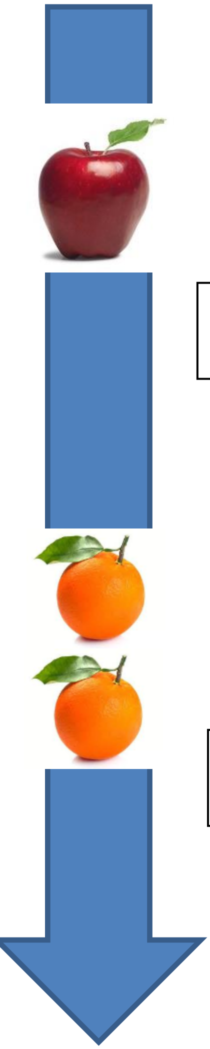
- For each S.A value a test using BF whether a exists in R.A column
- Ship to Location 1 those records that pass the BF test
  - If a value S.A does not pass the BF test, then S.A does not join for sure (why?)
  - But we may ship a few records that will not join (false positives)
  - Final result is always correct!



Extensions

- Support insertions/deletions/multi-set semantics
- Have a grocery store and the following list of transactions
  - Buy apple from supplier
  - Buy apple from supplier
  - Sell apple to buyer
  - Buy apple from supplier
  - Sell apple to buyer
- Do I have apples left in my store?

Intuition: maintain counters within buckets



Stream

BITMAP (after insertion of 1 apple)

0	0	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

BITMAP (after insertion of 2 oranges)

0	2	0	1	0	2	0	0	0	0	3	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Neat Implementation: Count-Min sketch

APPROXIMATE COUNTING

Applications of Count-Distinct

- Suppose stream elements are chosen from some universal set
- We would like to know how many different elements have appeared in the stream
  - Number of distinct (src,dest) pairs in traffic that flows through my routers?
  - How many different users visited Facebook/Twitter this week?
- Also useful when data is locally available for quick approximate answers
  - How many customers with at least one purchase?
  - How many people have visited my web-site?

Document Crawling

- While crawling documents from a web-site we count the number of different words that appear in them
  - Too low or too large may indicate artificial pages/spam

Distinct Value Counting: Flajolet-Martin Sketch

- **Problem:** Estimate the number of distinct items in a stream of values from [0,…, n-1]

Data stream:	3	0	5	3	0	1	7	5	1	0	3	7

Number of distinct values: ?

Number of Distinct Values?

53 36 37 41 41 60 7 38 45 82 21 53 32 93 62 73 73 92 65 6 54 1 96 52 18 79 0 36 30 5
33 24 66 61 83 71 45 97 91 25 48 67 22 7 7 83 49 56 16 80 90 23 70 25 57 64 55 9 25
25 3 68 19 21 60 73 33 5 64 36 96 97 11 46 95 81 9 12 63 9 2 89 30 99 51 78 46 3 65 12
51 96 80 57 60 46 34 22 82 95 57 54 95 52 34 60 65 24 26 59 94 67 71 30 55 45 75 35
82 52 27 42 73 77 93 36 50 10 8 80 87 48 55 76 91 26 99 3 20 45 1 40 85 71 99 8 56 49
88 58 14 84 35 15 92 85 21 40 66 11 59 65 12 10 88 33 92 65 70 10 89 4 88 80 69 14 92
13 65 75 94 81 60 42 35 31 54 14 44 14 86 0 32 28 47 89 81 61 84 18 77 19 46 48 9 51
63 69 83 15 7 53 58 39 15 64 3 57 79 2 87 85 71 3 29 26 0 51 39 17 60 59 34 77 26 70
91 20 68 50 93 39 38 55 27 3 89 53 15 5 39 34 82 81 36 59 7 73 18 43 65 1 26 72 76 44
75 36 18 60 79 14 85 13 66 34 14 25 1 39 72 1 77 22 54 99 62 19 46 29 52 27 57 80 60
76 48 92 47 33 23 7 85 45 67 59 31 17 15 41 44 51 41 40 16 1 35 41 49 51 64 4 21 11
85 45 81 8 22 79 80 24 31 17 74 80 86 49 60 78 90 39 79 43 16 37 98 9 76 40 0 49 72
34 95 4 33 28 97 16 7 86 11 99 25 68 97 64 42 10 2 88 2 37 92 42 55 18 58 23 52 15 45
71 61 32 84 11 37 24 85 23 72 79 8 98 48 96 35 64 78 37 55 4 2 72 4 36 76 9 66 99 27
20 75 60 95 23 18 87 47 71 44 26 75 11 5 1 83 11 81 46 32 28 15 83 17 70 31 92 80 2
76 22 40 5 91 66 18 84 69 78 80 25 69 98 93 31 62 95 74 91 94 25 2 1 65 5 73 77 11 38
96 21 39 43 56 11 85 45 79 47 72 35 47 40 2 61 41 97 68 59 71 29 17 37 20 9 51 63 69
83 15 7 53 58 39 15 64 3 57 79 2 87 85 71 3 29 26 0 51 39 17 60 59 34 77 26 70 91 20
68 65 19 40 53 81 65 22 64 30 62 67 28 77 45 14 95 71 5 32 62 47 23 57 60 87 62 31 48
54 7 85 13 49 74 0 24 68 9 88 85 21 60 38 47 71 84 87 82 74 59 67 97 31 33 27 47 13 6
68 75 53 63 68 18 64 98 59 90 23 53 66 2 87 88 28 48 98 6 97 90 13 49 7 7 21 25 29 62
9 25 64 30 70 19 67 16 2 89 61 45 23 25 63 29 12 54 5 49 39 43 56 3 8

How hard is it?

- Naïve (but OK if enough memory): bit array B of size n
  - Upon seeing item i set B[i]=1
  - Answer is #1s in B[]
- Similar idea: store items in a hash-table
  - Upon seeing i, store i in location indicated by h(i)
- However, these solutions
  - Do not work for large domains
  - Do not generalize for distributed settings nor for group counting
- Examples
  - Count number of distinct source/dest IPs seen in a router
    - There are 2^{64} possible pairs. Impractical to maintain one bit of each one of them or one hash-table entry for each observed pair
  - For each of my web-pages count the number of different users/IP-addresses that have visited that page
    - Would also like to have an estimate for groups or pages and the web-site as a whole

Distinct Value Counting [FM85]

- BITMAP array of B of L= O(logn) bits initialized to zero
  - Recall n is the domain size (e.g. 2^{64})
- Hash function h(x) maps incoming values x in [0,n-1] *uniformly* across [0, 2^L-1]
- Example:
  - L=8 bits
  - Domain of h(x) is [0..255]

Distinct Value Counting [FM85]

- Let lsb(y) denote the position of the least-significant 1 bit in the binary representation of y (i.e. rightmost bit set)
  - A value x is mapped to lsb(h(x))
- Example

  7 2 1 0

  - lsb(00100100) = 2
  - lsb(01011101) = 0

Equivalently: lsb(y) = number of trailing zeros in the binary representation of y

- For each incoming value x
  - set BITMAP[lsb(h(x))] = 1

EXAMPLE

Data stream: 3 0 5 3 0 1 7 5 1 0 3 7

Number of distinct values: 5

x = 3 → h(3) = 101110 → lsb(h(3)) = 1

BITMAP

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

ASSUME

h(0) = 011010

h(1) = 101101

h(5) = 100011

h(7) = 001001

FINAL BITMAP?

BITMAP

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

How do we use it?

- What is the probability that BITMAP[0]=1?
  - Recall that x maps uniformly to h(x)
  - Bit 0 is set to 1 if h(x)= ……..1
  - This happens ~half of the times (for the other half ls-bit is zero)
  - BITMAP[0] is set d/2 times (on expectation)

(d is the number of distinct items that we are trying to figure out)

Next bit

- What is the probability that BITMAP[1]=1?
  - Bit 1 is set to 1 if h(x)= ……..10
  - BITMAP[1] is set (about) d/4 times during counting

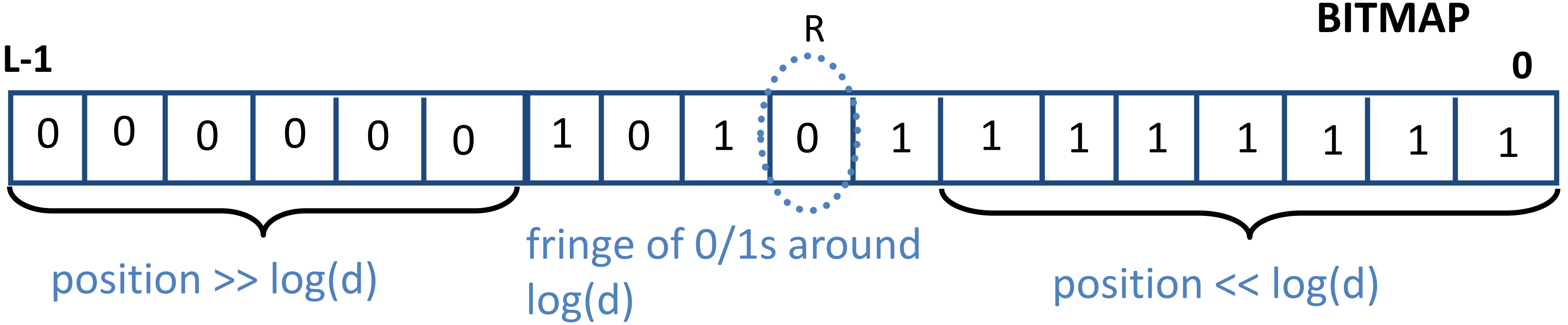
Next bits

- With similar arguments, the i^{th} rightmost bit in BITMAP is set with prob $1/2^i$ upon seeing a stream element

- Thus, we expect ~log(d) rightmost bits in BITMAP to be set with high probability

Estimate

- Let R = position of rightmost zero in BITMAP
  - FM show that $E[R]=\log(\phi d)$, $\phi=0.7735$
  - Thus, we estimate $d=(2^R)/\phi$



Back to our example

Data stream: | 3 0 5 3 0 1 7 5 1 0 3 7 |

Number of distinct values: 5

BITMAP

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

R=2

Estimate: d=(2^{2})/0,7735=5.17

WARNING

- Randomized algorithms have good *expected* behavior
  - But results may vary significantly between runs

What if

h(1) = 010100

NEW BITMAP

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

R=3

New estimate = 2^3/0.7735=10.3

Work around:

- use multiple BITMAPS, each with a different hash function

- combine estimates

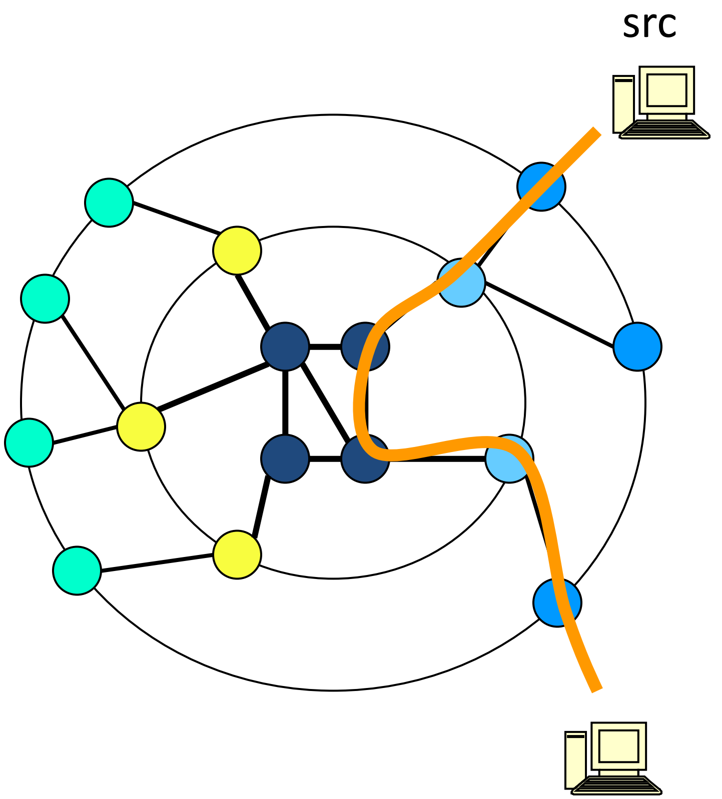
Median of averages

- Use k*l bitmaps, each with a different hash function
  - Consider them as k groups of l bitmaps
- From each group of l bitmaps take the average of their estimates
  - Occasionally, some of these averages will be affected by overestimation (previous example), or even underestimation
- Return the median of the produced k averages

Distributed Applications

- FM-sketches are composable
- How many distinct IPs transmit over our network?

- Compute FM sketch at each router
- Combine (by OR-ing) corresponding bitmaps



ESTIMATING MOMENTS

Generalized Counting Problem

- Computing "moments," involves the distribution of frequencies of different elements in the stream
- Let m_i be the number of occurrences of the i^{th} element
- The k^{th}-order moment (or just k^{th} moment) of the stream is the sum over all i of $(m_i)^k$

Examples

- Recall k^{th} moment = $\Sigma_i(m_i)^k$
- 0^{th} moment = #distinct elements in the stream
  - Solved with FM SKETCH
- 1^{th} moment = sum of stream elements
  - Easy, just a counter
- 2^{nd} moment = $\Sigma_i(m_i)^2$

Example of second moment

a, b, c, b, d, a, c, d, a, b, d, c, a, a, b

stream

$m_a = 5$

$m_b = 4$

$m_c = 3$

$m_d = 3$

2nd moment = 25+16+9+9=59

Second moment as a surprise index (skewed distributions)

a, b, c, b, d, a, c, d, a, b, d, c, a, a, b

stream 1

$m_a = 5$

$m_b = 4$

$m_c = 3$

$m_d = 3$

S=2nd moment = 25+16+9+9=59

a, b, a, a, d, a, c, a, a, a, a, a, a, a, a

stream 2

$m_a = 12$

$m_b = 1$

$m_c = 1$

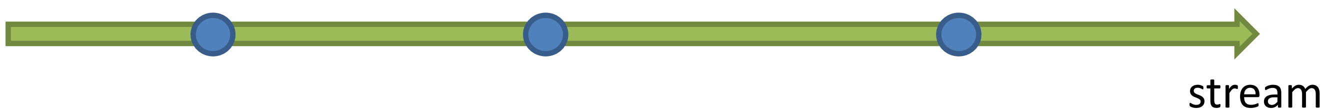
$m_d = 1$

S=2nd moment = 144+1+1+1=147

AMS technique

(by Alon, Matias and Szegedy, 1996)

a, b, c, b, d, a, c, d, a, b, d, c, a, a, b

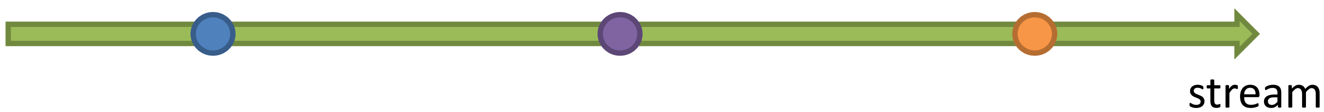


- Let X=(X.element,X.value) be a variable
- Pick a random position i in the stream
  - X.element = element at position i
  - X.value = a counter for item X.element from position i until the end of the stream

- E.g. for i=3, X.element = c, X.value = 3 at the end of the stream

Example with 3 variables

a, b, c, b, d, a, c, d, a, b, d, c, a, a, b



- Assume we pick locations 3,8 and 13
- At the end of the stream we have
  - X1=(c,3)
  - X2=(d,2)
  - X3=(a,2)

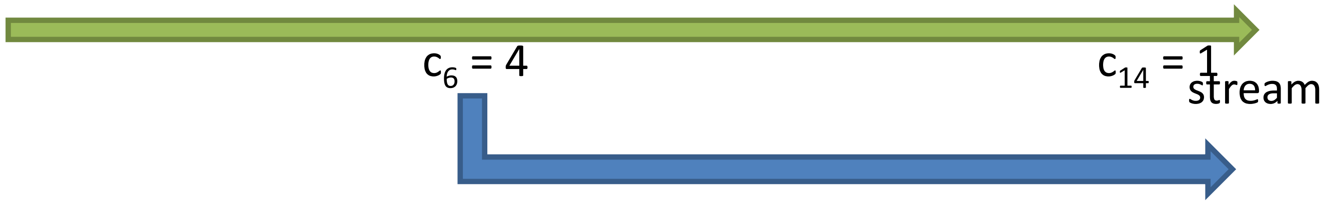
CLAIM: n(2X.value-1) is an estimate for 2nd moment S

X1 yields: 15(2*3-1)=75
X2 yields: 15(2*2-1)=45
X3 yields: 15(2*2-1)=45

AVG = 55
True S=59

Notation

a, b, c, b, d, a, c, d, a, b, d, c, a, a, b

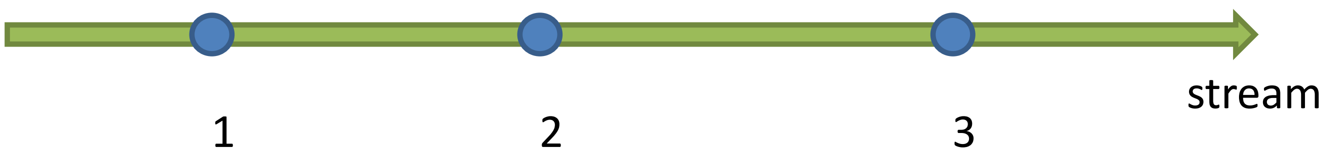


- Let c_t = number of times record at time t appears from that time on
- $c_1=m_a$, $c_2=m_b$, $c_4=m_b-1$,…, $c_6=m_a-1$, $c_{14}=1$

Observation

- Recall that for X we pick a random position i and start counting the observed element from that time on-ward

a, b, c, b, d, a, c, d, a, b, d, c, a, a, b

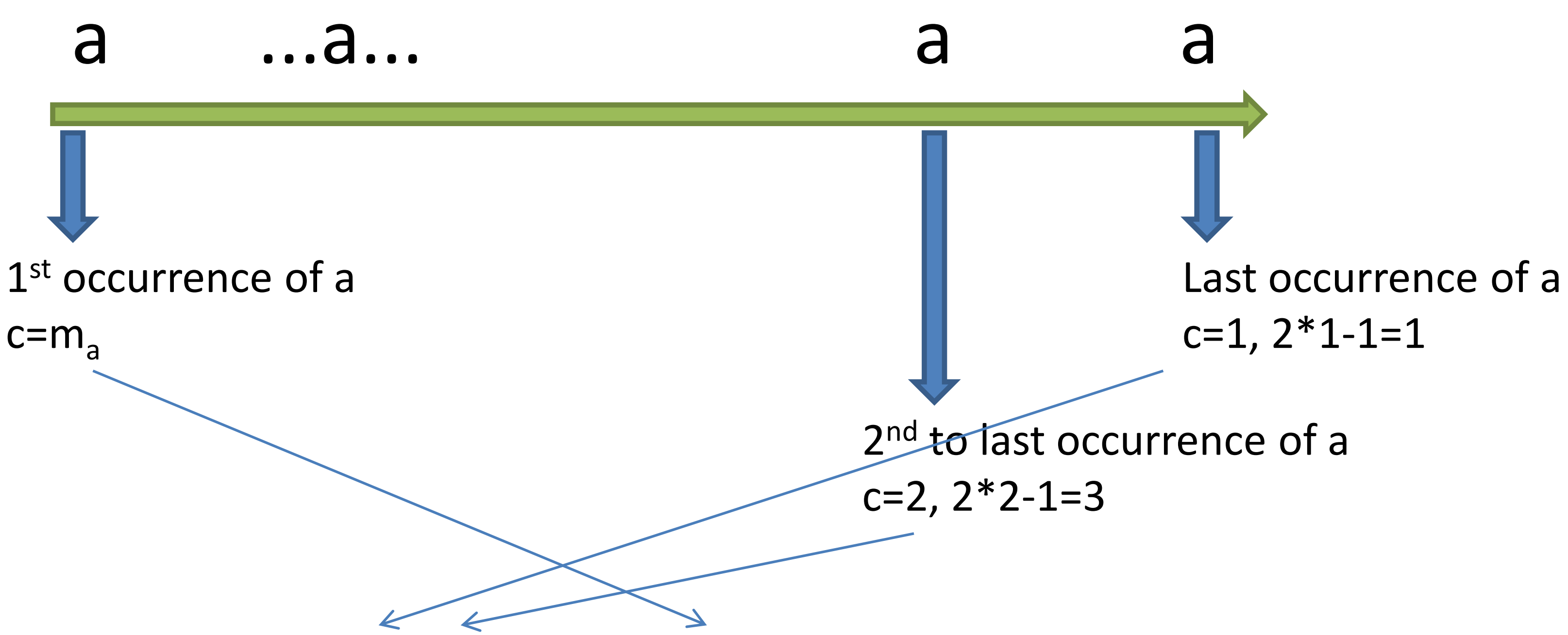


- Let Y=n(2X.value-1)
- Claim E[Y] = S

Proof

- Average over all possible positions i that can be used to initiallize X
- $E[Y] = 1/n * \Sigma_i [n(2c_i-1)] = \Sigma_i [(2c_i-1)]$
- We will rewrite the sum by iterating over all elements a,b,c,....

Consider some element a



- $\Sigma_a(2c-1) = \Sigma_a[1+3+5...+(2m_a-1)] = \Sigma_a(m_a)^2$
- Thus, $E[Y] = \Sigma_a (m_a)^2 = S$

Complication

- Since the stream is infinite, n keeps increasing
- How to maintain a random sample of size s (locations that define the X variables)?
  - If we pick locations too early, sample won't be representative of recent behavior
  - If we wait too long, then we will have few variables to answer queries
- Solution: reservoir sampling

Reservoir sampling

- Input n elements (n keeps increasing)
- Want a fixed size sample (assume size = s)
  - This is your "reservoir" of sampled items

- Solution
  - Choose the first s elements, keep them in memory
  - When the n^{th} element arrives ($n > s$), choose it with probability s/n
    - If chosen, throw away a random item from the sample

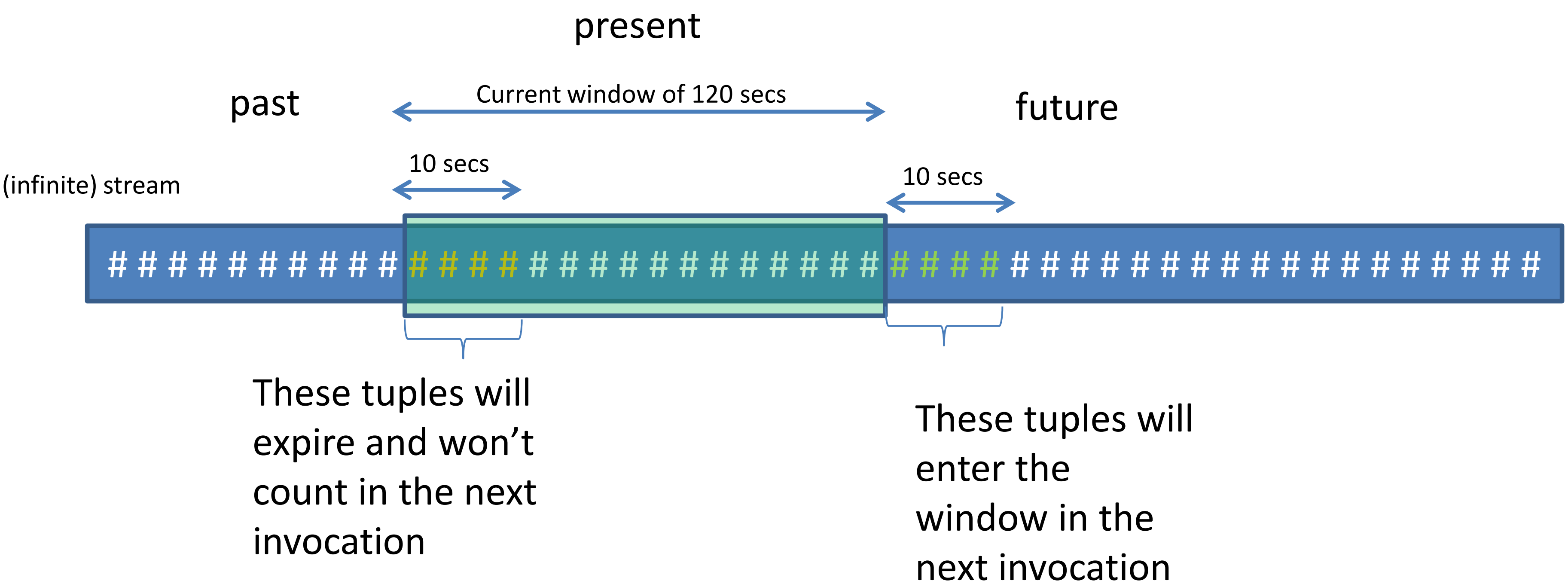
FREQUENT ITEM COUNTING

FREQUENT ITEM COUNTING

Example

- Example: given a stream of tweets, find the most popular hash tags
- Does not make sense to keep counters from a very distance past
- Mechanisms to concentrate on the most recent trends
  - Sliding windows
  - Exponential decay

Sliding windows



Spark (brute-force) implementation

```
val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
//filter hashtags only
val hashtags = words.filter(w=>w.contains("#"))
//count all hashtags in the last 120 seconds
val winh = hashtags.window(Seconds(120))
//iterate over accumulated hashtags
 winh.foreachRDD { (rdd: RDD[String], time: Time) =>
     val spark = SparkSession.builder.config(rdd.sparkContext.getConf).getOrCreate()
      // Convert RDD[String] to RDD[case class] to DataFrame
     val wordsDataFrame = rdd.map(w => Record(w)).toDF()
    // Creates a temporary view using the DataFrame
    wordsDataFrame.createOrReplaceTempView("words")
    // Do word count on table using SQL and print it
    val wordCountsDataFrame =
       spark.sql("select word, count(*) as total from words group by word order by total DESC")
    println(s"========= $time =========")
    wordCountsDataFrame.show(20,false)
```

This computation is repeated for all RDD data accumulated within a window

Sample Output

```
========= 1537816150000 ms =========
+-------------------+-----+
|word               |total|
+-------------------+-----+
|#NBAMEDIADAY       |77   |
|#DUBNATION         |6    |
|#NBA               |4    |
|#TRAININGCAMPTIPOFF|2    |
|#CLIPPERS          |2    |
|#PLAYGROUNDS2      |2    |
|#GOSPURSGO         |2    |
|#TRUETOATLANTA     |1    |
|#GSW               |1    |
|#JOINTHEREVOLUTION |1    |
|#SANANTONIOSPURS   |1    |
|#MEMORABILIA       |1    |
|#STACKED           |1    |
|#TORONTORAPTORS    |1    |
|#COLT45            |1    |
|#SPORTS            |1    |
|#CLAMPCITY         |1    |
|#1ON1              |1    |
|#STEPHENCURRY      |1    |
|#NBADRAFTROOM      |1    |
+-------------------+-----+
only showing top 20 rows
```

```
========= 1537816160000 ms =========
+-------------------+-----+
|word               |total|
+-------------------+-----+
|#NBAMEDIADAY       |79   |
|#NBA               |6    |
|#DUBNATION         |5    |
|#TRAININGCAMPTIPOFF|2    |
|#CLIPPERS          |2    |
|#GOSPURSGO         |2    |
|#GSW               |1    |
|#JOINTHEREVOLUTION |1    |
|#TORONTORAPTORS    |1    |
|#MEMORABILIA       |1    |
|#1ON1              |1    |
|#STEPHENCURRY      |1    |
|#SANANTONIOSPURS   |1    |
|#COLT45            |1    |
|#NBADRAFTROOM      |1    |
|#KAWHILEONARD      |1    |
|#STACKED           |1    |
|#PLAYGROUNDS2      |1    |
|#SPORTS            |1    |
|#CLAMPCITY         |1    |
+-------------------+-----+
only showing top 20 rows
```

Issues with this scheme

- Assume window = 1 week
- Recall our example of counting hash tags
- The number of hash tags in all tweets made worldwide is too large
- We are only interested in frequent hashtags
- It is not memory-friendly to keep counters for all hash-tags seen in the current window (especially for the infrequent ones)

Decaying windows

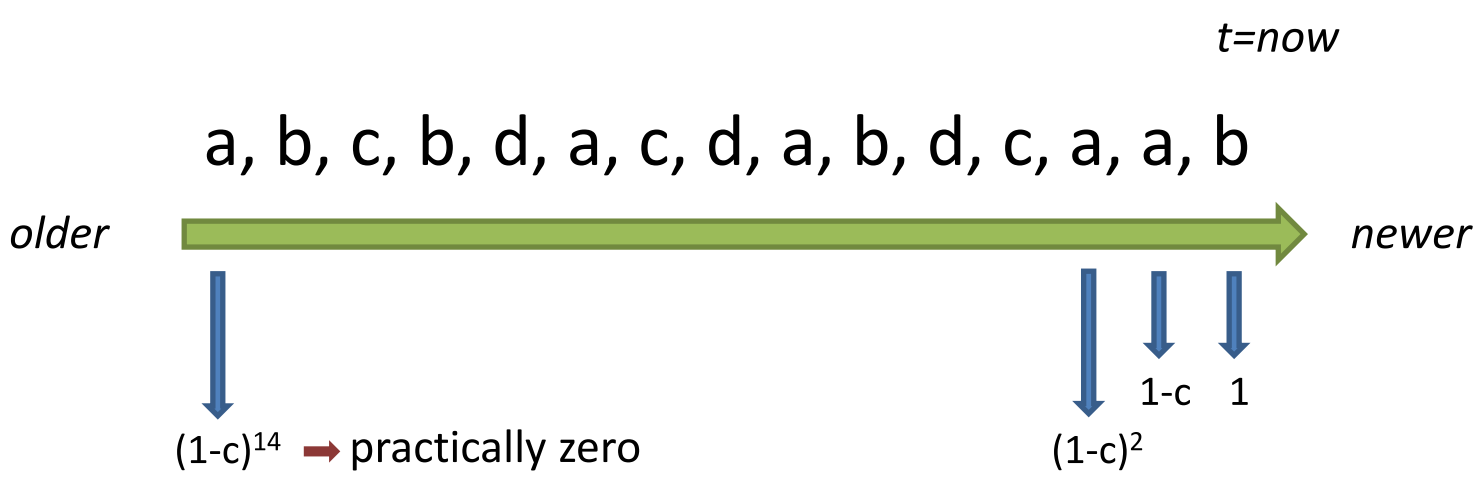
- Sliding windows make sharp distinction between recent elements and those in the distant past
  - weight = 1, if recent (within specified window)
  - weight = 0, otherwise
- Decaying windows
  - weigh recent elements more heavily
  - older elements receive monotonically smaller weights
    - Recent history is more important than distant past

Exponentially Decaying Windows

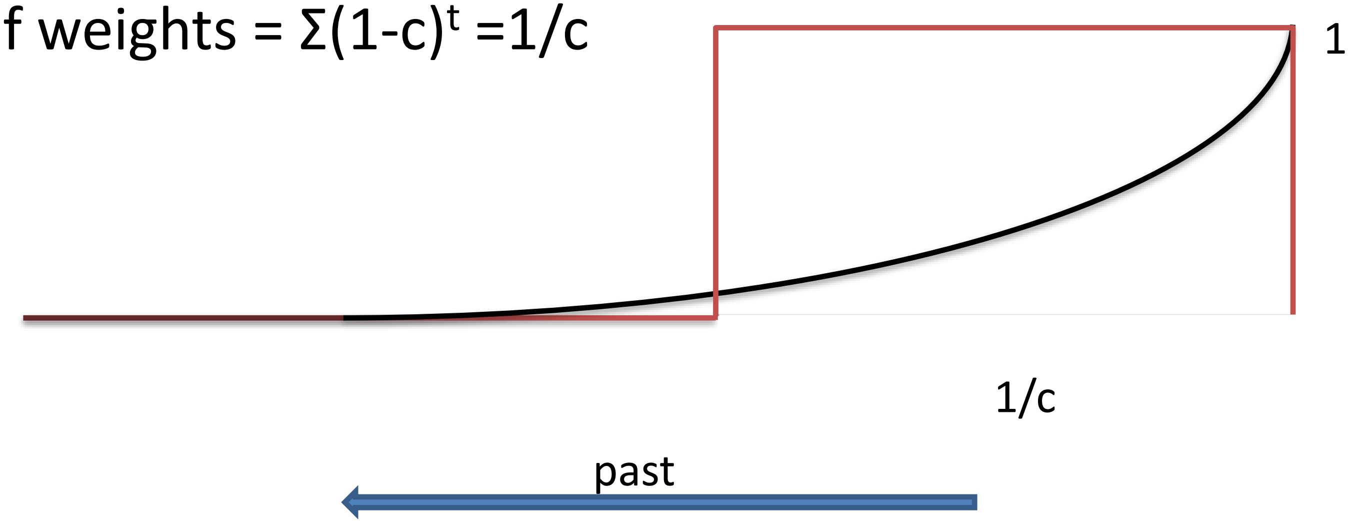
- Given a stream of numerical items $a_1, a_2, \ldots. a_t$
- Assume we would like to compute their SUM
- For a small constant c<<1, compute

$$\sum_{i=0}^{t-1} a_{t-i}(1-c)^i$$

Spread of weights



Sum of weights = $\Sigma(1-c)^t = 1/c$



Counting using Decaying Windows

- Keep a counter for each item seen
  - We will discard counters for infrequent items later-on

- Upon seeing item a
  - Multiply counters for all items by (1-c)
  - Then, add 1 to the counter for a
    - If no such counter exists, initialize it

Example (c=0.05, 1-c=0.95)

Input stream: a(t=1), b(t=2), a(t=3), d(t=4)

- Upon seeing a(1)
  - count(a) = 1
- Upon seeing b(2)
  - count(a) = 0.95
  - count(b) = 1
- Upon seeing a(3)
  - count(a) = $0.95^2 + 1 = 1.9025$
  - count(b) = 0.95
- Upon seeing d(4)
  - count(a) = 1.9025 * 0.95 = 1.807
  - count(b) = 0.9025
  - count(d) = 1

Example (c=0.001, 1-c=0.999)

Input stream: a(t=1), b(t=2), a(t=3), d(t=4)

- Upon seeing a(1)
  - count(a) = 1
- Upon seeing b(2)
  - count(a) = 0.999
  - count(b) = 1
- Upon seeing a(3)
  - count(a) = $0.999^2 + 1 = 1.998$
  - count(b) = 0.999
- Upon seeing d(4)
  - count(a) = 1.99801 * 0.999 = 1.996
  - count(b) = 0.998
  - count(d) = 1

Pruning

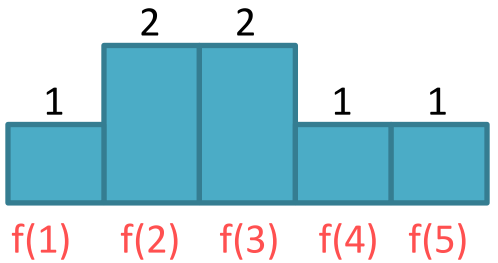
- Say we want frequent items with counts > s
  - drop counters smaller than s
- Recall that weights sum to 1/c
- There can be at most 1/sc counters exceeding the threshold
- E.g. for s=1/2, c=1/1000, there can be at most 2000 counters in use

LINEAR PROJECTIONS

Linear-Projections

- Seek *to* build a small-space summary for distribution vector f(i) (i=1,..., N) seen as a stream of i-values

Data stream: 3, 1, 2, 4, 2, 3, 5, . . .



- ***Basic Construct:*** *Randomized Linear Projection of f()* = project onto inner/dot product of f-vector

$$< f, \xi >= \sum f(i)\xi_i$$

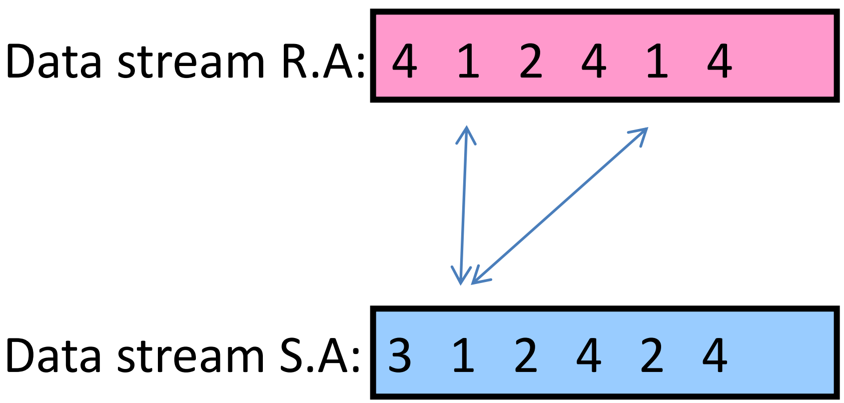
Where ξ = vector of random values from an appropriate distribution

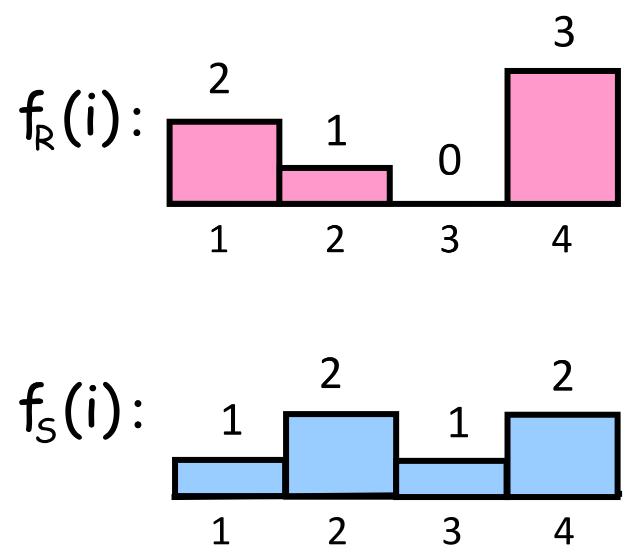
Data stream: 3, 1, 2, 4, 2, 3, 5, . . .

$$\xi_1 + 2\xi_2 + 2\xi_3 + \xi_4 + \xi_5$$

Example: Binary-Join COUNT Query

- Problem: Compute answer for the query COUNT(R $\bowtie_A$ S)
- Example:





$$COUNT(R \bowtie_A S) = \sum_i f_R(i) \cdot f_S(i)$$

$$= 10 \quad (2 + 2 + 0 + 6)$$

- Exact solution: too expensive, requires O(N) space!
  - N = sizeof(domain(A))

AMS Sketching Technique [AMS96]

- Key Intuition: Use randomized linear projections of f() to define random variable X such that:
  - X is easily computed over the stream (in small space)
  - E[X] = COUNT($R \bowtie_A S$)
  - Var[X] is small

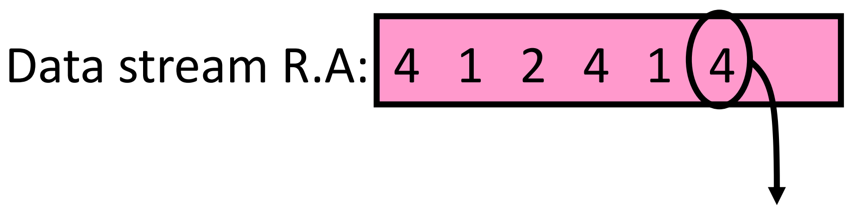
Used to provide probabilistic error guarantees
(e.g., actual answer is 10±1 with probability 0.9)

- Basic Idea:
  - Define a family of 4-wise independent {-1, +1} random variables ξ_i : i=1..,N
  - Pr[ξ_i = +1] = Pr[ξ_i = -1] = 1/2
    - Expected value of each ξ_i, E[ξ_i] = 0
  - Variables ξ_i are 4-wise independent
    - E[$\xi_1 \xi_2 \xi_3 \xi_4$]=E[ξ_1]*E[ξ_2]*E[ξ_3]*E[ξ_4]=0 (expected value of product of 4 distinct ξ_i s is zero)

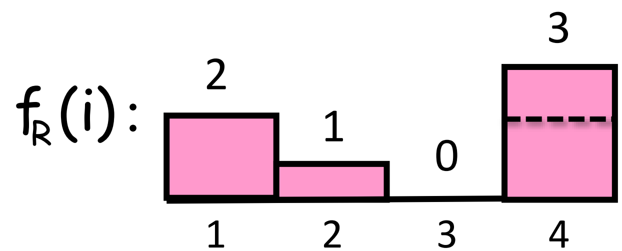
- Variables ξ_i can be generated using pseudo-random generator using only O(log N) space (for seeding)!

Summary Construction

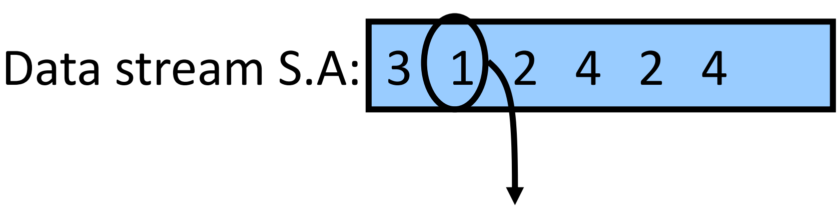
- Compute random variables: $X_R = \sum_i f_R(i)\xi_i$ and $X_S = \sum_i f_S(i)\xi_i$
  - Simply add ξ_i to X_R (resp. X_S) whenever the i-th value is observed in the R.A (resp. S.A) stream
- Define $X = X_R X_S$ to be estimate of COUNT query
- Example:



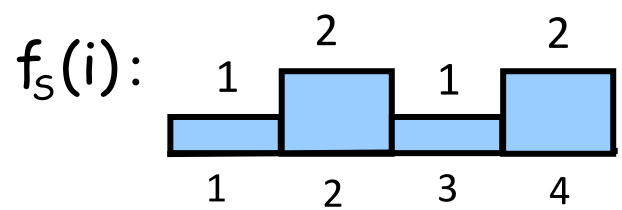
$X_R = X_R + \xi_4$



$X_R = 2\xi_1 + \xi_2 + 3\xi_4$



$X_S = X_S + \xi_1$



$X_S = \xi_1 + 2\xi_2 + \xi_3 + 2\xi_4$

Binary-Join AMS Sketching Analysis

- Expected value of X = COUNT(R $\bowtie_A$ S)

$$
\begin{aligned}
E[X] &= E[X_R \cdot X_S] \\
&= E[\textstyle\sum_i f_R(i)\xi_i \cdot \sum_i f_S(i)\xi_i] \\
&= E[\textstyle\sum_i f_R(i) \cdot f_S(i)\xi_i^2] + E[\sum_{i \neq i'} f_R(i) \cdot f_S(i')\xi_i \xi_{i'}] \\
&= \textstyle\sum_i f_R(i) \cdot f_S(i)
\end{aligned}
$$

1

0

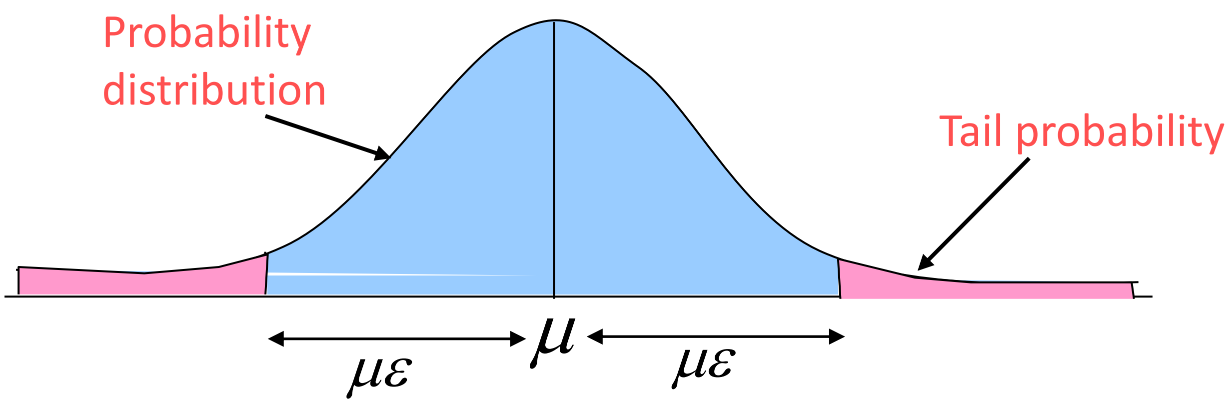
- Using 4-wise independence, possible to show that

$$Var[X] \leq 2 \cdot SJ(R) \cdot SJ(S)$$

- Where $SJ(R) = \sum_i f_R(i)^2$ is *self-join size of R (2nd moment)*

Tail Inequalities

- General bounds on *tail probability* of a random variable (that is, probability that a random variable deviates far from its expectation)



- <u>Basic Inequalities:</u> Let X be a random variable with expectation μ and variance Var[X]. Then for any ε>0 it holds thank

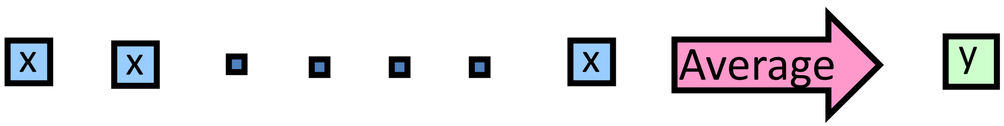
Chebyshev: $$\Pr(|X-\mu| \geq \mu\varepsilon) \leq \frac{Var[X]}{\mu^2\varepsilon^2}$$

Boosting Accuracy

- Chebyshev's Inequality:

$$\Pr(|X - E[X]| \geq \varepsilon E[X]) \leq \frac{Var[X]}{\varepsilon^2 \, E[X]^2}$$

- Boost accuracy to **ε** by averaging over several (=s) independent copies of X (reduces variance)



$$s = \frac{8 \cdot (2 \cdot SJ(R) \cdot SJ(S))}{\varepsilon^2 \, COUNT^2} \quad \text{copies} \qquad E[Y] = E[X] = COUNT(R \bowtie S)$$

$$Var[Y] = \frac{Var[X]}{s} \leq \frac{\varepsilon^2 \, COUNT^2}{8}$$

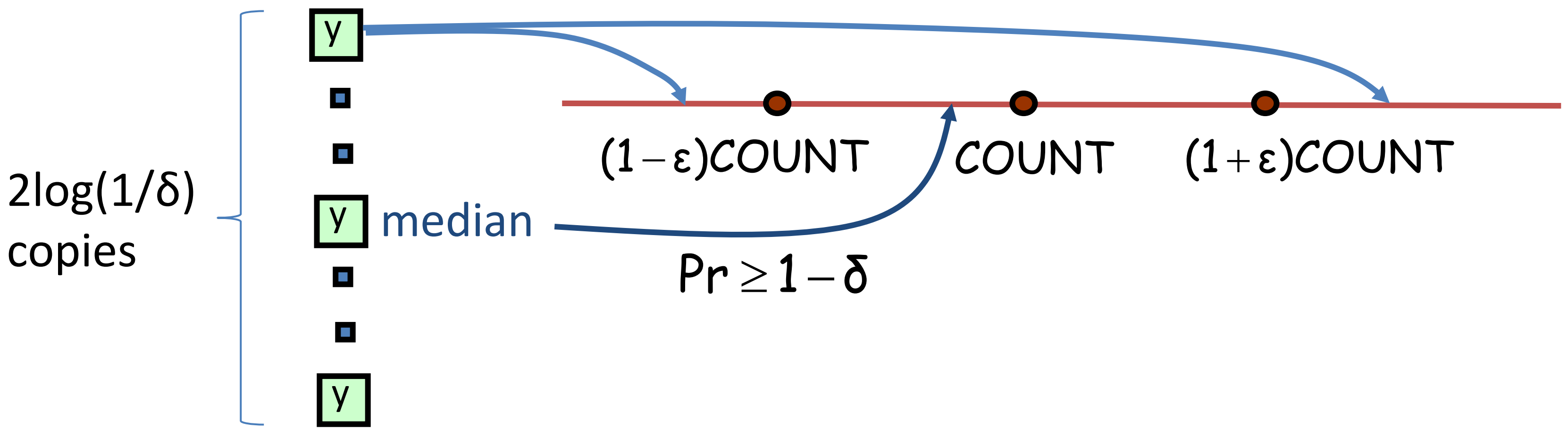
- By Chebyshev:

$$\Pr(|Y - COUNT| \geq \varepsilon \cdot COUNT) \leq \frac{Var[Y]}{\varepsilon^2 \, COUNT^2} \leq \frac{1}{8}$$

Boosting Confidence

- Boost confidence to 1-δ by taking median of 2log(1/δ) independent copies of Y
- Each Y = Bernoulli Trial that fails with probability ≤ 12.5%. With 87.5% it succeeds to provide estimate within (1± ε)

"FAILURE": $\Pr \le 1/8$



Pr[|median(Y)-COUNT|≥ ε*COUNT]

= Pr[# failures in 2log(1/ δ) trials >= half of the trials = log(1/δ)] ≤ δ by Chernoff Bound

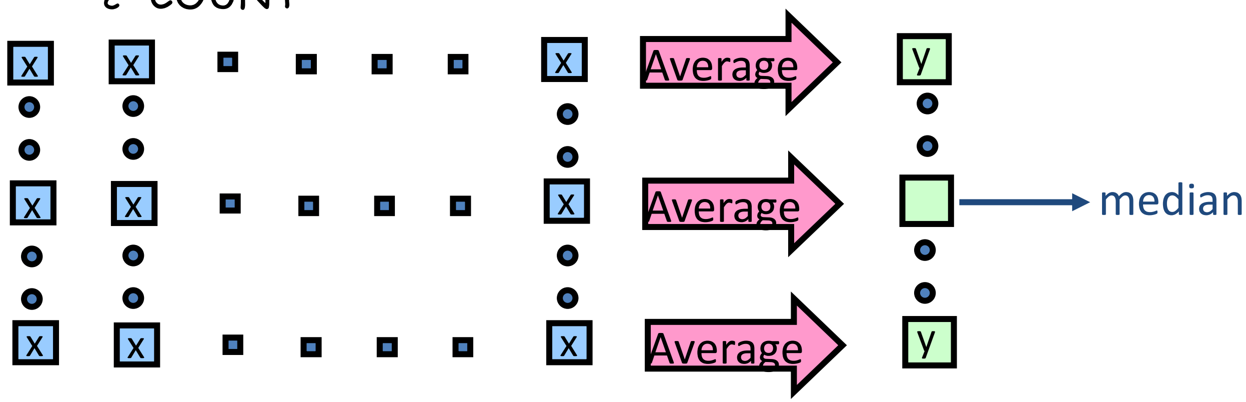
E.g. probability that more than half of Y's are out of the (1±ε) range is smaller than δ

Summary of Binary-Join AMS Sketching

- <u>Step 1:</u> Compute random variables: $X_R = \sum_i f_R(i)\xi_i$ and $X_S = \sum_i f_S(i)\xi_i$
- <u>Step 2:</u> Define X= $X_R X_S$
- <u>Steps 3 & 4:</u> Average independent copies of X; Return median of averages

$\frac{8 \cdot (2 \cdot SJ(R) \cdot SJ(S))}{\varepsilon^2 \ COUNT^2}$ copies

$2\log(1/\delta)$ copies



- <u>Main Theorem (AGMS99):</u> Sketching approximates COUNT to within a relative error of ε with probability ≥1 –δ using space

$$O\left(\frac{SJ(R) \cdot SJ(S) \cdot \log(1/\delta) \cdot \log N}{\varepsilon^2 \ COUNT^2}\right)$$

A Special Case: Self-join Size (2nd moment)

- Estimate COUNT(R $\bowtie_A$ R) $= \sum_i f_R^{\,2}(i)$ *(original AMS paper)*
  - Second moment of data distribution

In this case, COUNT = SJ(R), so we get an (ε,δ)–estimate using space only

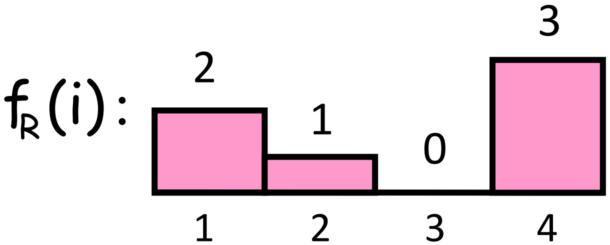
$$O\left(\frac{\log(1/\delta)\cdot \log N}{\varepsilon^2}\right)$$

Best-case for AMS streaming join-size estimation

Question

- Can I estimate some arbitrary portion of the distribution using these techniques?

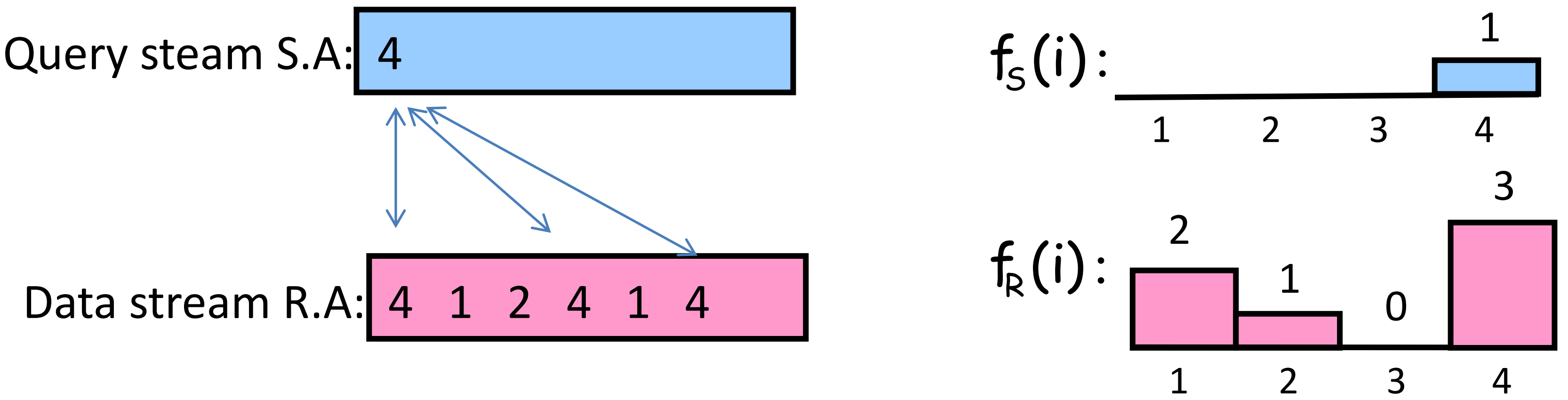
Data stream R.A: 4 1 2 4 1 4



- E.g. What is the value of f[4]?

Trick

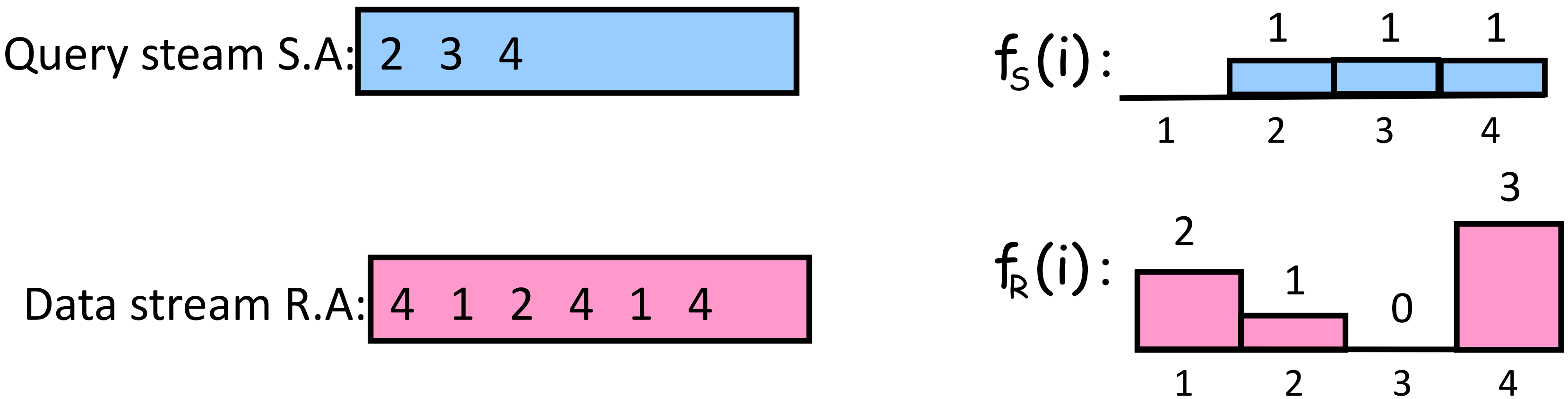
- Think of your query as a second distribution S we want to sketch



- Then answer = $f_R[4]$ = COUNT(R⋈S)

Same for Range Queries

- Think of your query as a second distribution S we want to sketch



- Then answer = $\Sigma_{i=2..4} f_R[\iota]$