

# Data Streams

soego nga ralp#hoendaaa s lo#rde:uia# #k#ptia tlinri uje osapi lo i.r#bi#ssu d  
hd#;nin e p#adta splitaa cre l r#isatclo# #raids#iteseter r#ialeiiflounicetaape  
e blie o i la n br.egl intesnan ereeaiun uoble arl rd .ls#t#gae.ei#rrrosce#lh  
esi . egqi e#ia#fyasofauageearsttdoayd tr otaes n#f . aafaa.r#ruei#aiisi l#met  
o.me ilofo p#og#e#es c#d#i#l#er#id#ux# ed#r a. tm r d#r#p#iee o#o#o#ab a#ia.s eaano  
eea tesu i i poim#n#nu sodeatsaariat.ip#ssel m#f#s#at#ry#aoon#uy#er#ox#ai a#o#j#c h  
uc#f#t#b#ic ns ia ns#n#a#i#e#l#t#se#h oereu sy oon r#E#s#a#e#n#d#o#c#s#n#p#o#; ed oc rri aerci  
ng#o#q#i#s s#n#j m#o#st et#i#o#g a#o#l#o#r#:# h or#e.e#e#p#o#e#j#r#r eoc e#o#o#v#e#i n#u#y#j#e#ar#i u emid#l  
d#n c#e#r#d#aaa au ree#s#o#s c#e#o#l#e#i#e#e#e#o#o d r e#s#a#r#i#e#e#n#j t meca il#s#o#s i b lu#t#s#n#e#o  
t pc .c#p#l#i#n#f#q#r#e#u#d r s#e#i#o#t#i#s#u#e uro #l#r i#u#m da ar#t#r#a,i#a so#h ad:#o tade#ar#l#u  
o#e o q#e e#o n i#y#i#c#e#r#n#a xl l#r#e#t#e#s#t c#i#t#i#e#r#al r#o#e#g#e#s#n r n#e#n#d#i#e#u#r n#f#i t r#l  
s#d#o#a#a#u .e#s#f.u t#t#a#a n#n#u c#i#e#n#o#i#r r s#u#e#i#o#l#n#r#m#l t#e#s#l#i#f#n,t#r#o#l#r o#n#s#e n#o#v#e l  
o#o n#e#u#e#o#c#e#n#u#p#at j#l#e n#d#o#f#a#e d#e#i#s#o n c#p#a#f#t#d#i#u#s#s e#o c a#e#j#o#u#d#o#e#n#l i#u.#o#a#d#e#t#u  
l t#i#s#e o#n u#e#b#e#o#a.l#e#e#c#o#d#m#e#r#a#c#i#a#i#b#t#a#e#r s s#l#g#f#o#r#t#i#c#n#a#n#a u#d#u#o#e#i s#o#a#e#r l#n#t#l#f  
a (e#e#d#a#c c#r#e#e#o#u#t#s#t#e#y#o .s#e#l#o#u j#t c#s i#n#r j#o#o#n#t.#p#t#s#e#e#o#q#l o#e#a#a# c c#z#i#o#n#p#v#e  
n r i#o n#q#e#e b u#e#e r#l j#n#a#d#i u#d#r.s#b#r#n#i#u#ab o#s r g#u#r#v#o l#i s#e#o#s#t#e#r#a#l#i#o#u#n#g#o#e#d#r#e  
o#t#d#y#a#s#o#s#u#e#a#i#g#y#i#e#n#e#g#a e#d#a#i#e#l#i#a#n#r#i#t q#l#d#s#u#e#e#s#d a#e#u#s#o#e#l v#a#s e#e#s u r#n#o#e#a  
t#o#s#r#r#n o n#a#h e r#l#u#e#e#n#s.e#r#g#d#o#s c#a#e#o#d#o#c#e#n#l a#g#a#o#r#e i#e#o#l#l#n#o a#l#o#c#t#s#t#a#h l#u#n#i#r#r#n  
l#i#a a.n#o#a#u#n#a#d#i#e#f i#r#t#e#i#r a#e#c r#e#n#t#e#a#l#o e#n#r c#a#n#a#f#e#s#s#o#e#t#s o#r#r#e n#a#i#d#a#c#c#r#d#p#u#o#d  
e#o#l#b#a#o,o#r#t#h#i#g#u#e#r,a#r #a#l#b#o#l#r#h#a#o,e#i#r#i#d#e#t#j#f#n#o#a#v E#n u l r#e#l#d#l. e#j#o#f q i#e#d#o#f#p#n  
a#a#o t#s a f#s#e#e#n#e#u#i#e#p#v#e#d i#e.r#e#e#a#s#b n#r#o#e#h#a e#l#b a#l#l#r#d#e#o#e#t#d#s#n#o#p#s#s#a#s#e#d#d#l b#l#e#u#d  
t#e#e#a o#u.n#e#s a.m#a#n#d#o#n#a#v#f o#s#r#r#e#s j#o#a q#h#a#r#e.i l#s f#e#d#o#A#e#r#m#f d#e#s#i#l#s#q#e a. u#o#a#l#e#u  
a r#m o f#s#i#o#d#c#r#a a#t#e#o#e#r#t#e#i#e#s#r#n#T#o#s#l s#a#i#l o#i#u#r#n#g#o#g#b#n#s.n#s#o#c r#o#d#s#i#a#y#i#b#z#i#t#e#t#a#d#a a  
o#a a n r#m#d#e#r#s#q#s i#t#e#t#o n#o ;b t#t#u#f#n#a#l n s#a#n#g#e#a#a#r u#s j i#e#g#a#q#n#e#e r#e s#a#g#b#d#f#i  
b#i#e#s#v#o#i.y#l#e#e#l#e.i#a.s#i e o n#i e#m#a#p#l#v#t#v#o#d#u#o#d#t#s#i#i#r t#i#s.j#i#o#p#r#a#o#o#i#n#t#s#a n#a#r#e#e c#o#d#e#g#o  
l#r#o,o#u#s q#e m#o#n#s#o#e#i#l#d#i#r#r#e#z#u#n#s#i#n#c e#a#b n t#o#n#t d#i l#s#q#e f#l#e#o#i#s#l#g#a a#l#l#r#e#a#d r#r l#o  
b#p#m#a#e#z#a#e#m#s#t i#g s a#e#a#o#s#e#r#p a#i s#e#t#o#p#r#o#q#t#s#y#p#s#o#e#s#o.a v#y t p#o#d#p#r#t#h#a h#o#n#p a#l#m#o#r#d  
c#t#n#s b#i#n#s#e#t#i e s#r. n s#e e#s o#s#e#c o#l#j#i o#a#v#o#s#a#a#e#t#n#o#s r#e#e#u#r#o#p#n#o#n#o i#n#d#o#o#r#r#h#a#n#p#z.l  
o#u c i d#e#d#e#t#a#l s#ci c#u#v#t#a#d#e t#e#a f s#s#t#e. e#a#c u#a#s#t#u#n#i h#u#s#i#s#o#r#e#j#s o#a#o#e#n#o n#e#a#r#o#d  
f#p#q#l m#a#n#t#e y#e l#r u#r#i e#u#l#a#b#e#a#e#a#n#o r n#r#s#a#a t#o c#o#a#e y#n#e d#e#l.e#r a#s#t#a#u n#e#m#e e#C#a#r#t#s  
B#j#i#o#l#l#e#a#i#u.i c#o#a#i#s a#o#o e#p#o#b#e#p#i#e#a#o#i t.e#e#f#v#m t#t r#o o#p#e#r#l#i#a#f#a#h#c#e#s#e#g#e#b r#i#t#n#i#c#r#u#t  
u#l#p l#a o#p#f#e#a#e i#q#e#o s m#a#i#r#e#e#s.s#r#e#d#r#u#p#u#r q#z#a#h n#a v#m#q d#f.e#e#l#t#s#a#n#g#e s l#n#o#p#o#i  
e#s#s . i#n#d#e#l#i n i#e#l#a#e#d r#u o#o#d n#s#b o d#f a#h#n i#a#l#a#n o#l l#i#n#g#l#i#l#e#d#e#p#z#a#e#i#r e#l n e#e  
a#i e m#e#c#r o#p n#r#o#o#l d i#o e#e#r#e#l#d c#a#t#o a#e#n#d#p#u#e t#o#e#i#t#a#r#n b#i#c#n#a o#a#u#i#a#l e#o#p#t#e#d#a#t  
g#o#e#i#a e#a n#o#l n#g#o#s#f#e#d#i#r#c#e#a#e.t#n#n#e#g l#i#n m-#E#a#i#o#s#l#e#r#n#i#s#l#t#p f.#u#i s b#u#e l#c#u g. e#a d#r  
a#m#u#n#d#u#b#e#e#p#e a B#g#o#g#o#.l#l#a#x#o#q#C m a#t#i n#a#u#a#a#n a#l#r d n#e#i s#u#i#t#d#e#c#s#n#g#e#f#o#a#l#e#i#d o#i#t#a#h  
e i#b#n#e#s#s b#t f o#c#r#e#s#o#s#o#b#d#i#e#e#n#b#l e e#u#m#e#i#r n#m#s.i#s.e#o#e#q#d o#e#i s#i o#i#a#i#h o#l#a#d#o#a#n#r  
c#e#y a#e#n#i#e#r o#e#l b#i a#n o#t#s#u o#e#s#e#o#b c#i#n.g u#l#u r#v#a#g i#l#e#a#p#a#l#p.t#m s#o#v#i#n#e.l#b#n#a#l#a  
n#l n e#a l#a#r s#s#a#l#r#m#d#u#p r#r#p#e#l t#r#o# i#o#a#t#u#r#v#e#l i e#r#e#b#i#o#l m#a i#r u#s#x l#u#i d s#i#t#i#o#d#o  
m#a#n#e#a#d#o#r#a#o#c#o#l v#s#s e#l#t o b#e#m#e#s#r#e#n#t n#o#a#s#o o#e#l#u#q#a#g a#s#s#n#y#o l#e#a#n#a#i#a#f#e.a e#p#t#n#p#d s#a



# What is data stream?

- Large Data Set:
  - Continuous
  - Massive
  - Unbounded
  - Possibly infinite
- Fast changing and requires fast, real-time response
- Example: Traffic to popular website (Facebook, Google, Amazon)



# Data stream management

- Essential for many applications such as:
  - Financial applications
  - Network monitoring

# Problems?


- Stream is hard to:
  - **Store** - Data is *continuously growing* faster than our ability to store or index it
  - **Process**
  - **Transfer**



# Difficult to Process

Unbounded number of elements



					
<b>Count</b>	4	5	3	4	3

...but we want finite space, depending on the problem

# Difficult to Process

- Short processing time for each element in the stream
- Only one single pass at the data

# Why stream processing is important?

Answering queries about this sort of data requires clever observation techniques and data compressing methods



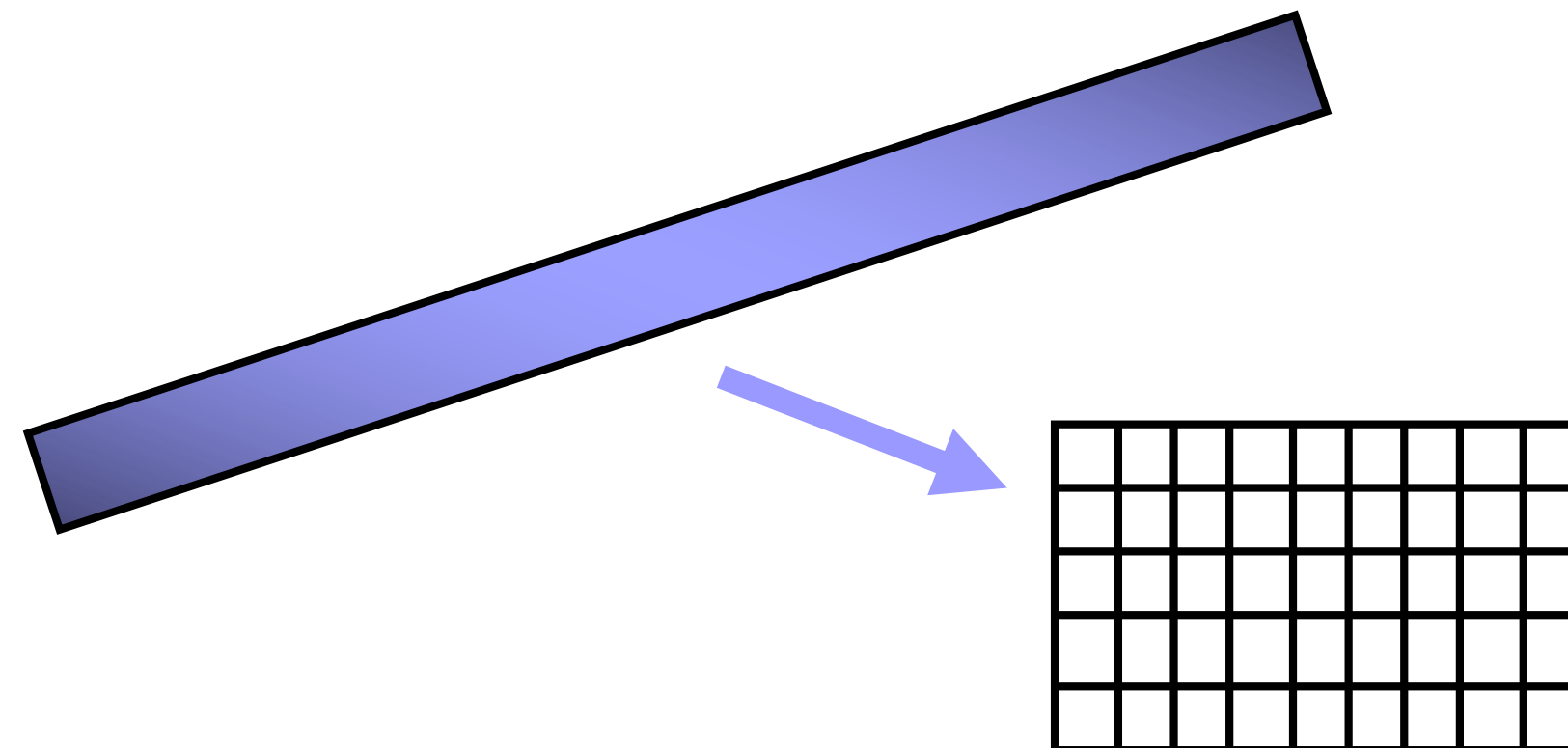
# Generic solution

Stream processing algorithms often build compact approximate sketches of the input stream



# Sketch

- Try to build a small data-structure to represent the data you want to obtain from the stream
- The smaller the data structure, the less accurate the results



# Generic solution

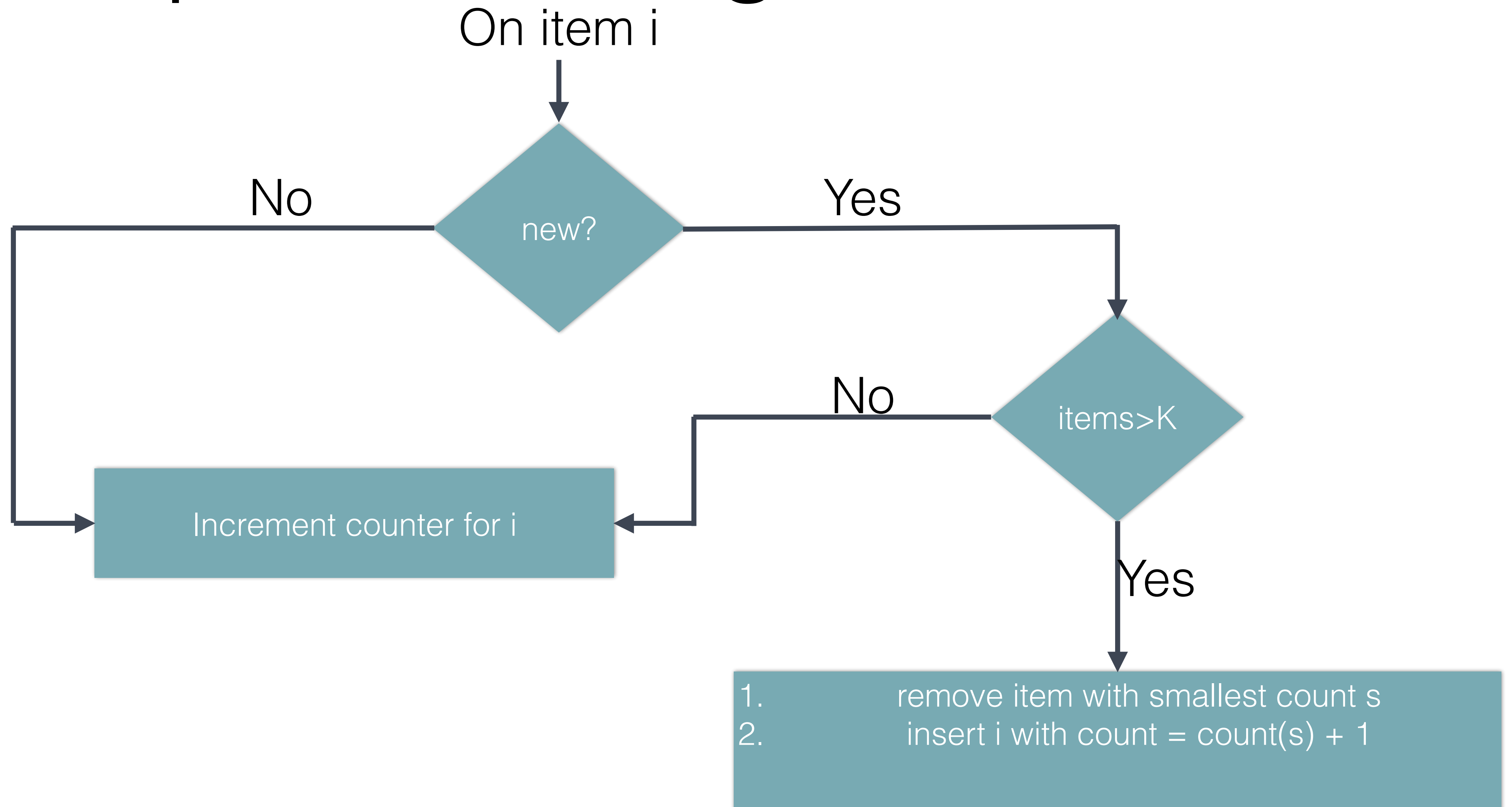
Almost all algorithms are approximate, answer with error and guarantee a bound on the error



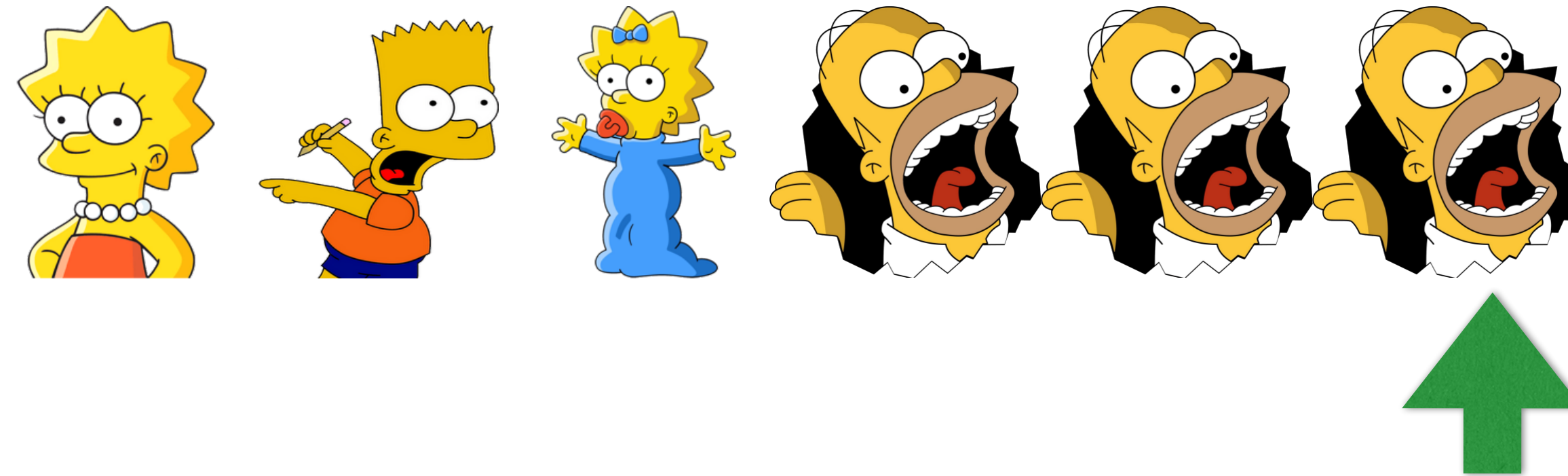
# Space saving algorithm

- Keep  $k$  items and counts initially zero
- Count first  $k$  distinct item exactly

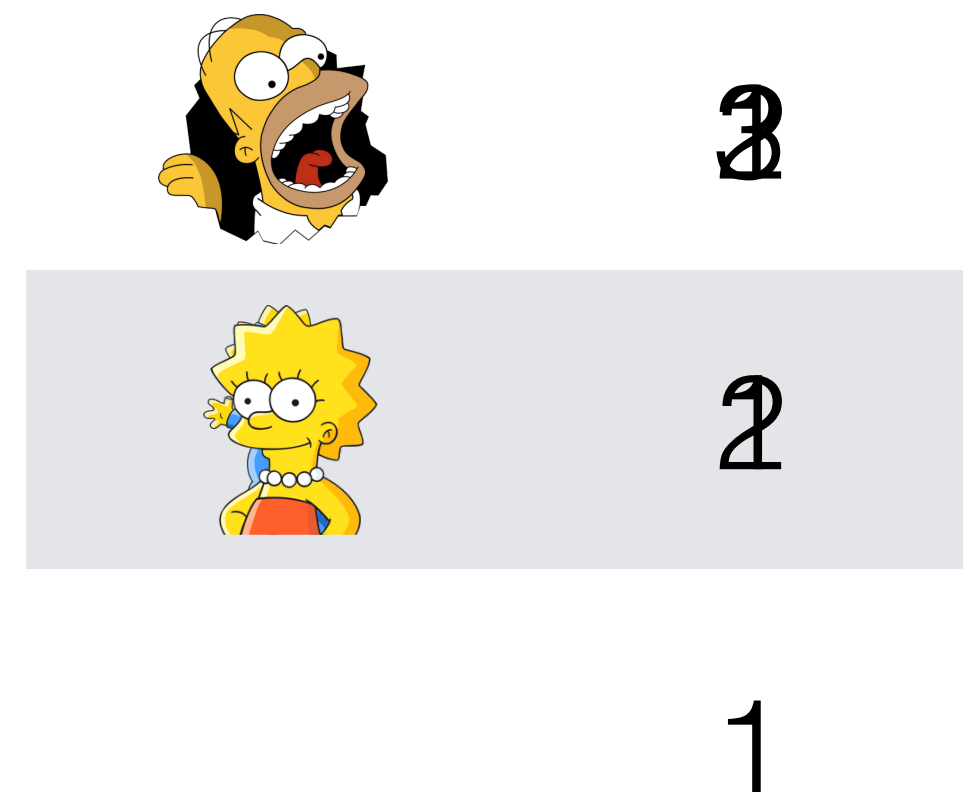
# Space saving model



# Space saving algorithm



K=3





# Space saving Guarantees

- When

$$k = 1/\epsilon$$

- We denote the overall number of insertion by  $Z$

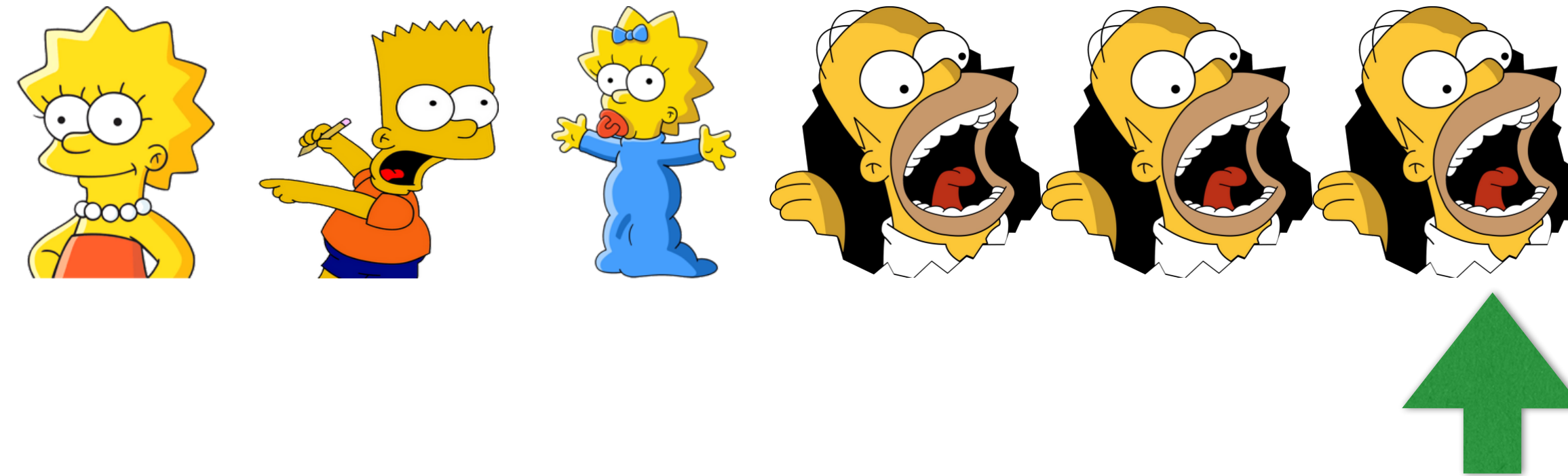
- The minimal counter is at most  $Z$

- The estimation error is  $Z \epsilon$

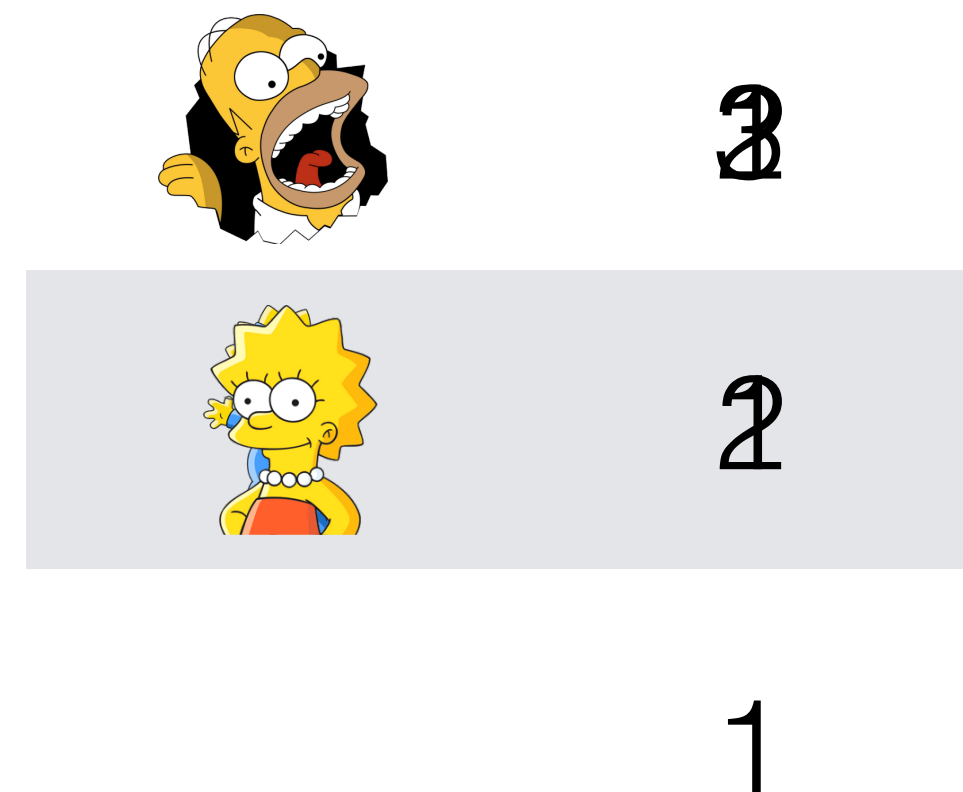
$\epsilon$

$\epsilon$

# Space saving algorithm

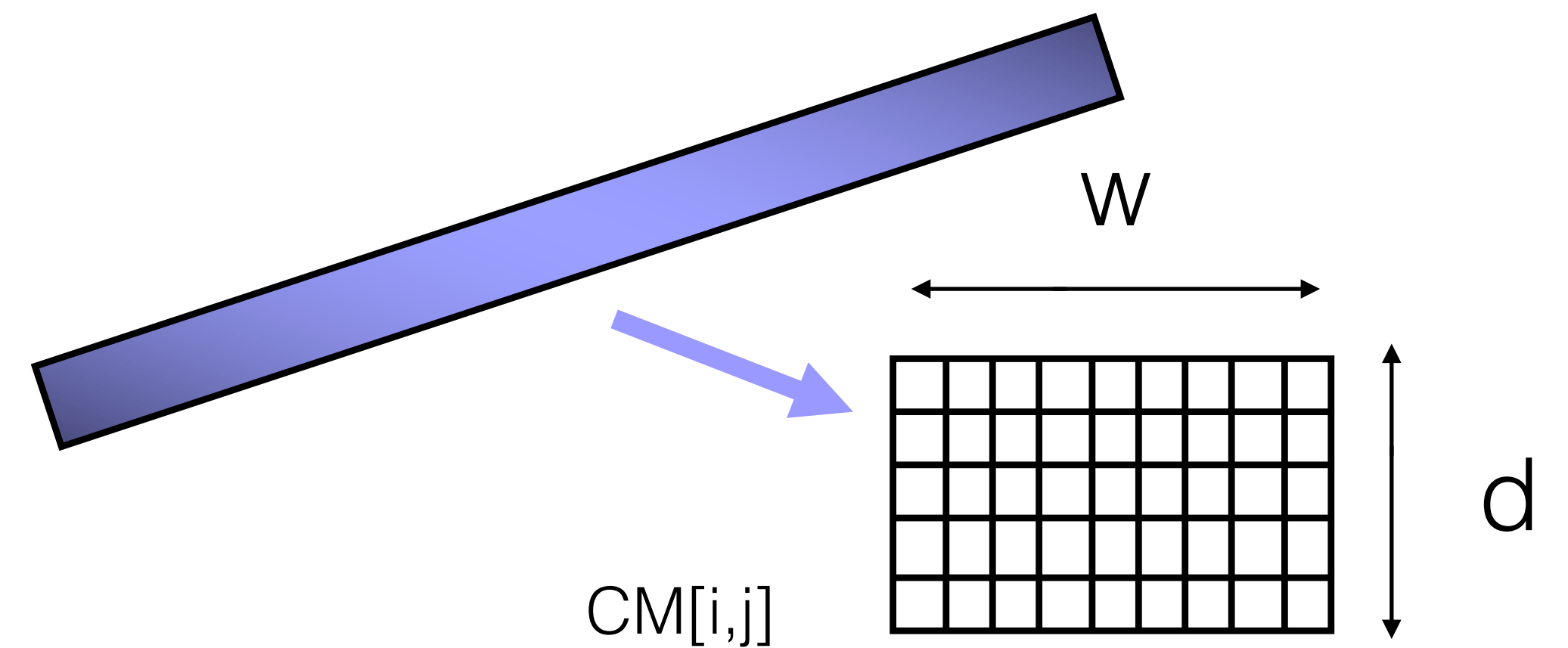


K=3



# Count min sketch

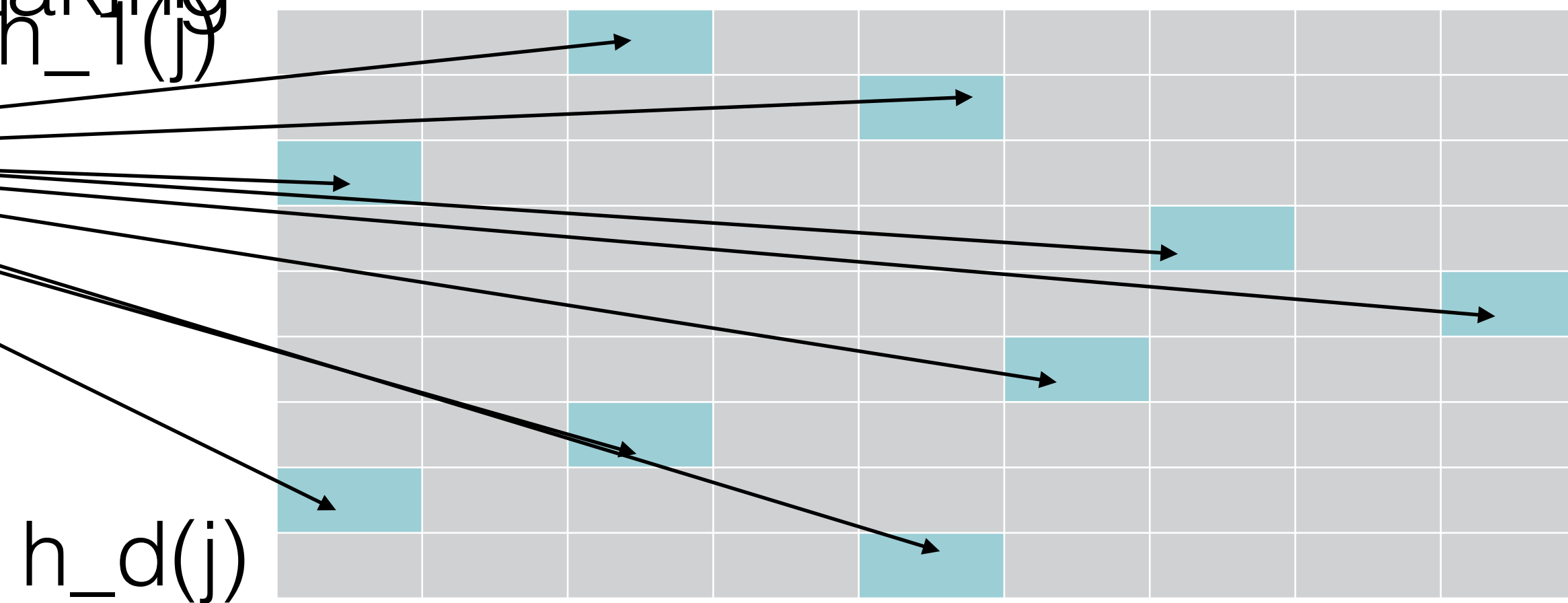
- Sketch that estimate item's frequency over a
- Creates a small summary as an array of  $w \times d$
- Use  $d$  hash functions to map to  $[1..w]$





# Count min sketch

- Estimate item  $j$  by taking  
insert  $j$



$$\min_k \{CM[k, h_k(j)]\}$$

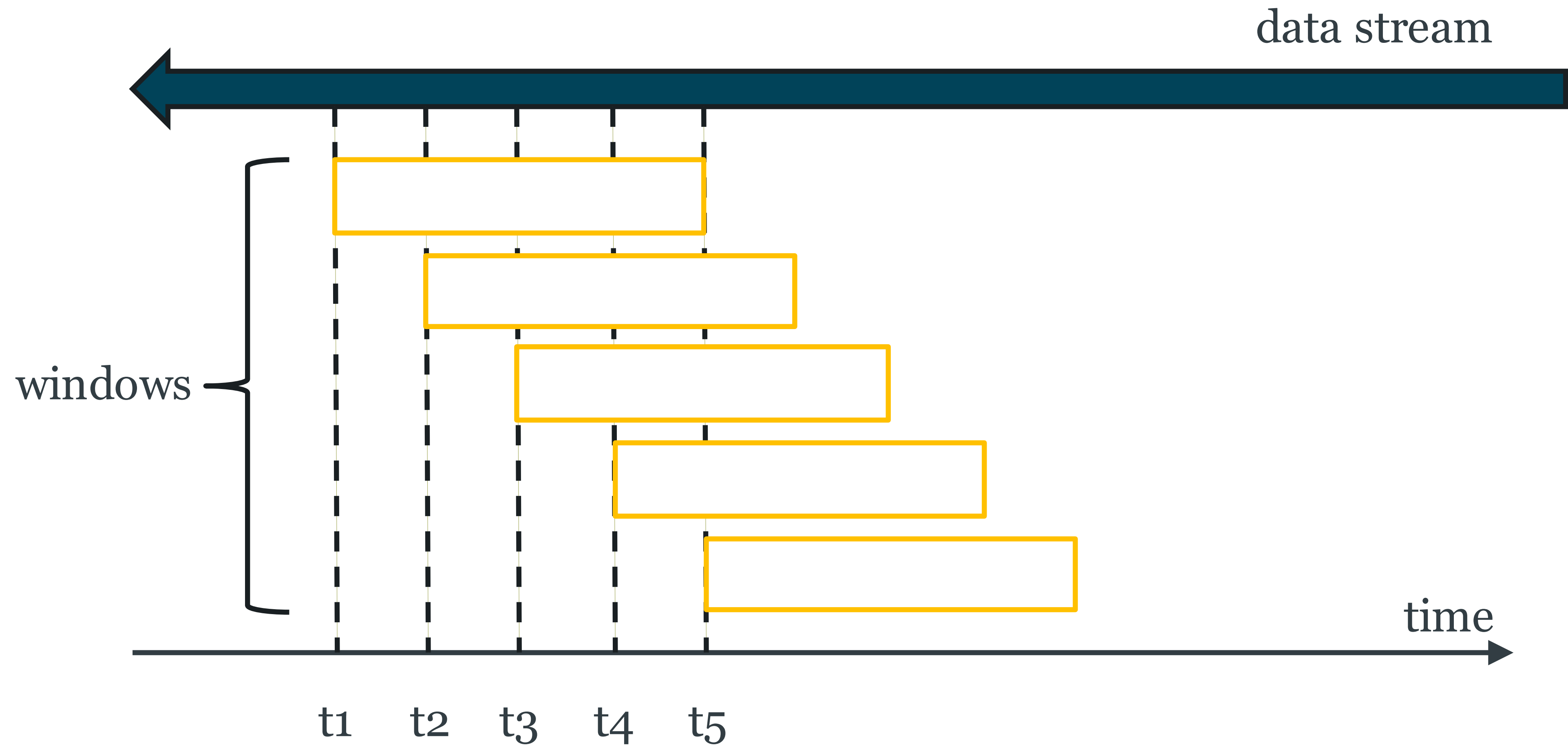
# Count min sketch Guarantees

- CM sketch guarantees approximation error on point queries less than  $\epsilon \|A\|_1$  (input stream as a vector  $A$ ) in space  $O(1/\epsilon \log 1/\delta)$
- Probability of error is less than  $1-\delta$

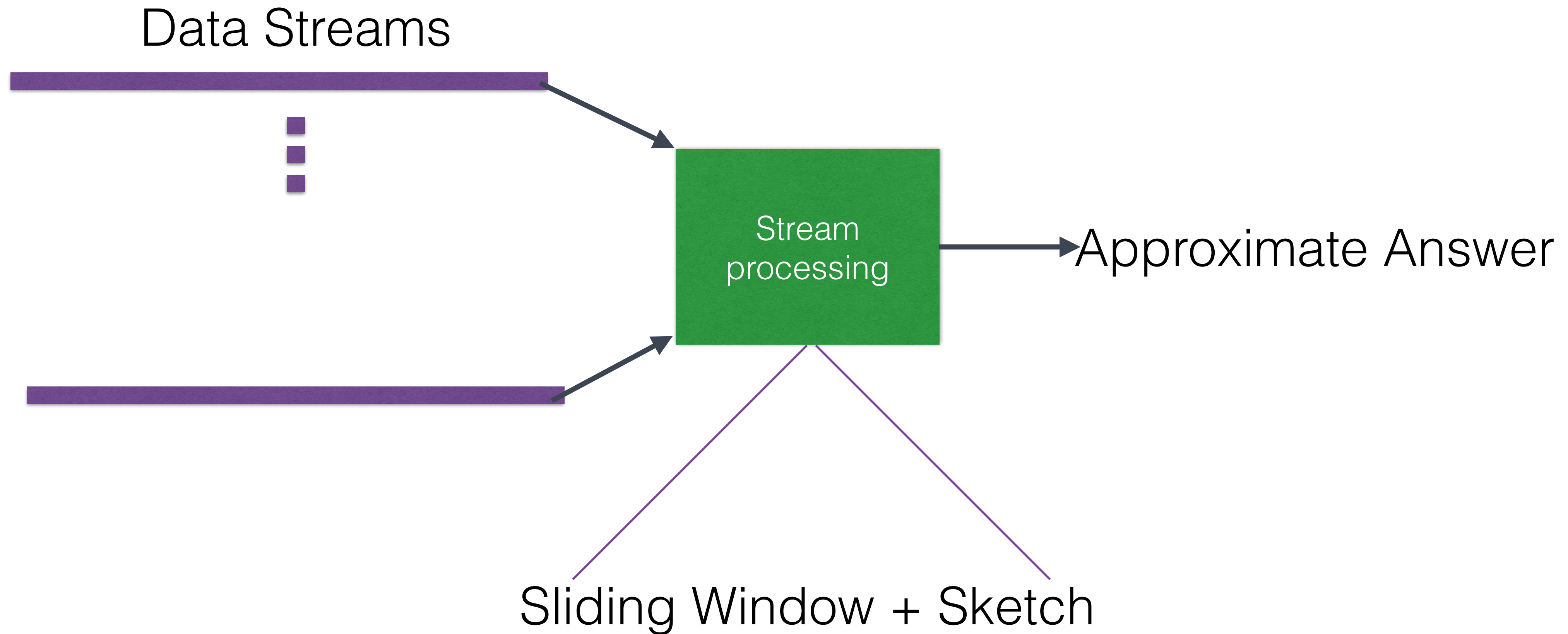
# Sliding windows

- For most applications, OLD data is considered less relevant
- Apply aging mechanism for the sketches
- Sliding Window Model:
  - Only last “w” elements are considered

# Sliding windows



# Computation Model



# The Problem with existing sliding window solutions

The window of interest may not be known a priori

OR

may be multiple interesting windows



# Contribution

We study a model that allows the user to specify an interval of interest that is contained within the last  $w$  items at query time

We improve space and operation performance of the existing work

# Existing Works - ECM

- Introduce sketching technique, called ECM - Exponential Count Min
- ECM combines count-min sketch with Exponential histograms
- Exponential histograms is a sliding window counter that can guarantee a bounded relative error

# Existing Works - ECM

- ECM sketch replace each Count-Min counter with Exponential histogram
- ECM has an error probability

# Naive solution: Raw

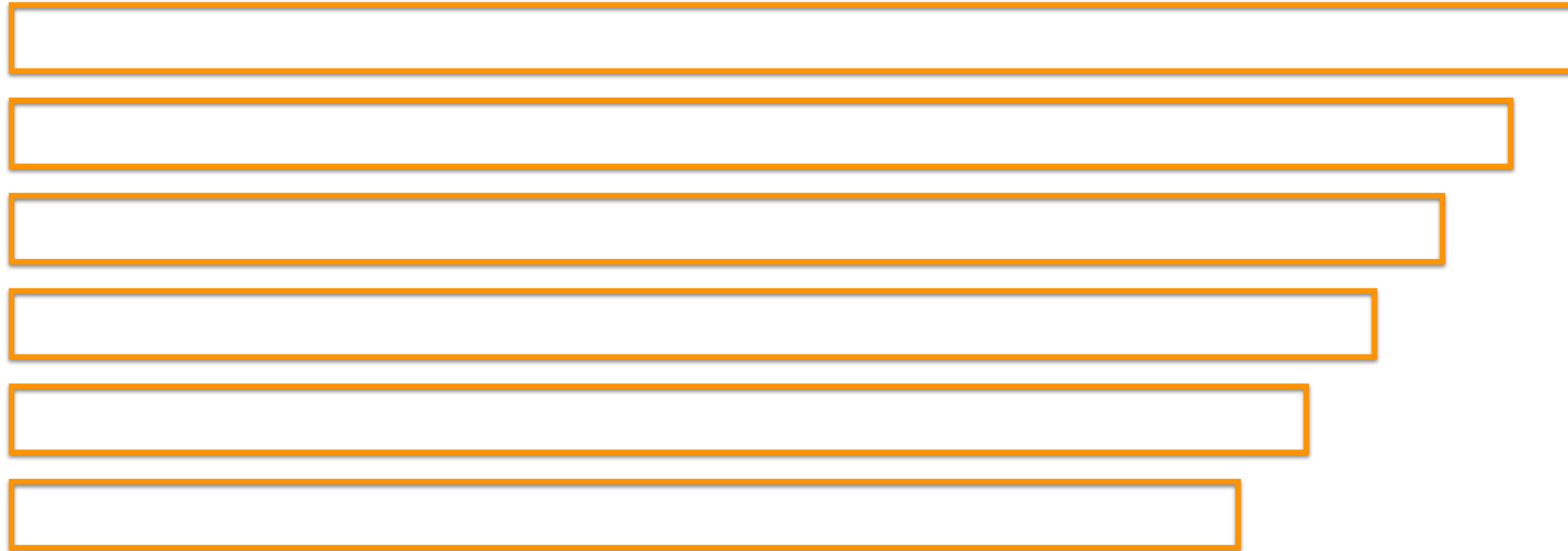
- Uses several instances of a black box algorithms that solves frequency estimated over **fixed sized** window
- Add( $\mathbf{x}$ ): Add item  $\mathbf{x}$  to all instances
- Interval Query:
  1. Query the relevant instances (closest to interval range)
  2. Subtract the result

W



$W\varepsilon/4$   $W\varepsilon/4$   $W\varepsilon/4$

$W\varepsilon/4$



# RAW vs. ECM

- RAW achieves constant query time while ECM answers queries in  $O(\varepsilon^{-1} \log W \log \delta^{-1})$
- Both consumes same amount of memory
- RAW is deterministic while ECM has an error probability



# Problem Definition

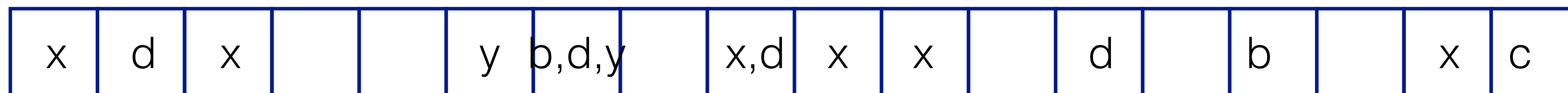
- **Add( $\mathbf{x}$ )**: Given an element  $\mathbf{x}$ , append it to stream
- **IntervalFrequency( $\mathbf{x}, i, j$ )**: Return an **estimation** of  $\mathbf{x}$  frequency between the  $i$  and  $j$  most recent elements of  $\mathbf{s}$ , denoted by  $\hat{f}_x^{i,j}$

*(W, ε) – IntervalFrequency :*

$$f_x^{i,j} \leq \hat{f}_x^{i,j} \leq f_x^{i,j} + W \varepsilon$$

# n-interval problem

- Arriving elements are inserted to blocks



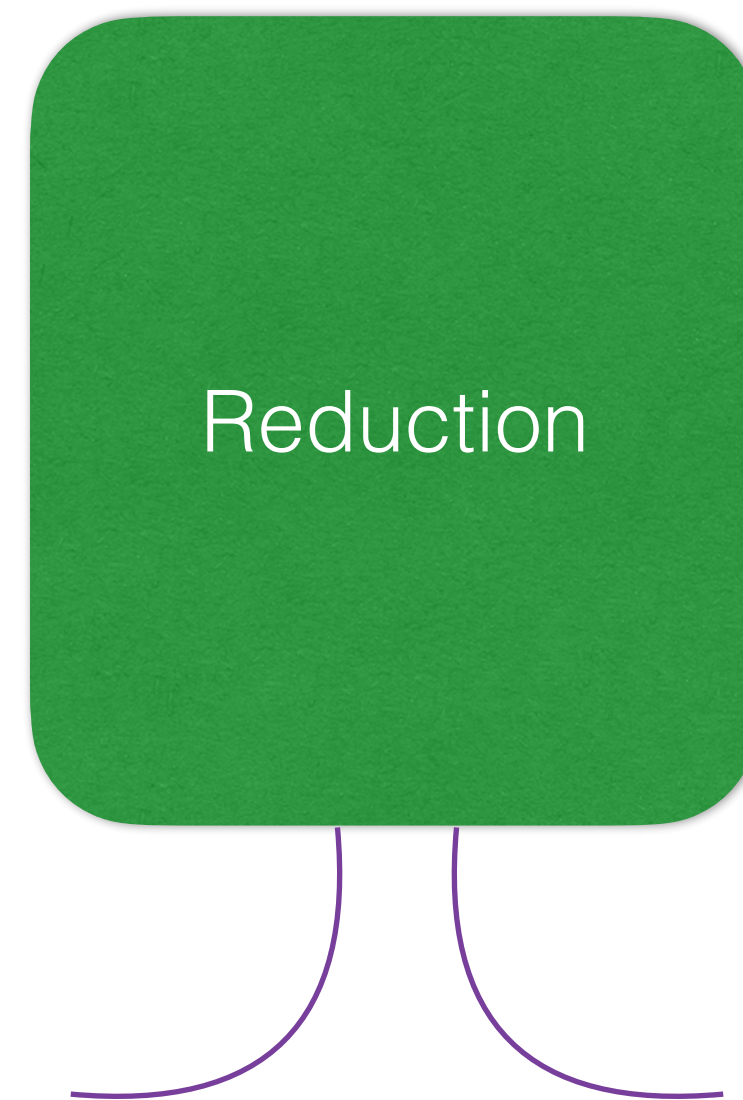
- Compute exact **block** interval frequencies within the blocks

N=18

# N-Interval problem Definition

- n-Interval Problem: Block Interval Frequency
- Add( $\mathbf{x}$ ): Given an element  $\mathbf{x}$ , append it to stream
- EndBlock(): New block inserted, old block leaves
- IntervalQuery( $\mathbf{x}, i, j$ ): Return the number (**without error**) of blocks  $\mathbf{x}$  appears between blocks  $i, j$

**n-Interval Problem**



*$(W, \varepsilon)$  - Interval Frequency*

Decide when to add elements in the blocks

# Reduction

1. Break the stream into  $w$  sized frames
2. Divide each frame into  $n$ -equal-sized blocks, each of size  $w\varepsilon$
3. Employ Space Saving to track element frequency within each frame
  1. Whenever a counter reaches an integer multiple of the block size, associated it to most recent block
  2. When the frame ends, flush Space Saving instance

# Implementing ADD(x)

x



x



Space Saving

If result mod block size = 0



if offset mod block size = 0  
EndBlock



# Implementing IntervalFrequency( $x, i, j$ )

1. Compute relevant blocks number
2. Call Query of n-interval problem
3. Return block size \* result

**n-Interval Problem**



*$(W, \varepsilon)$  - Interval Frequency*

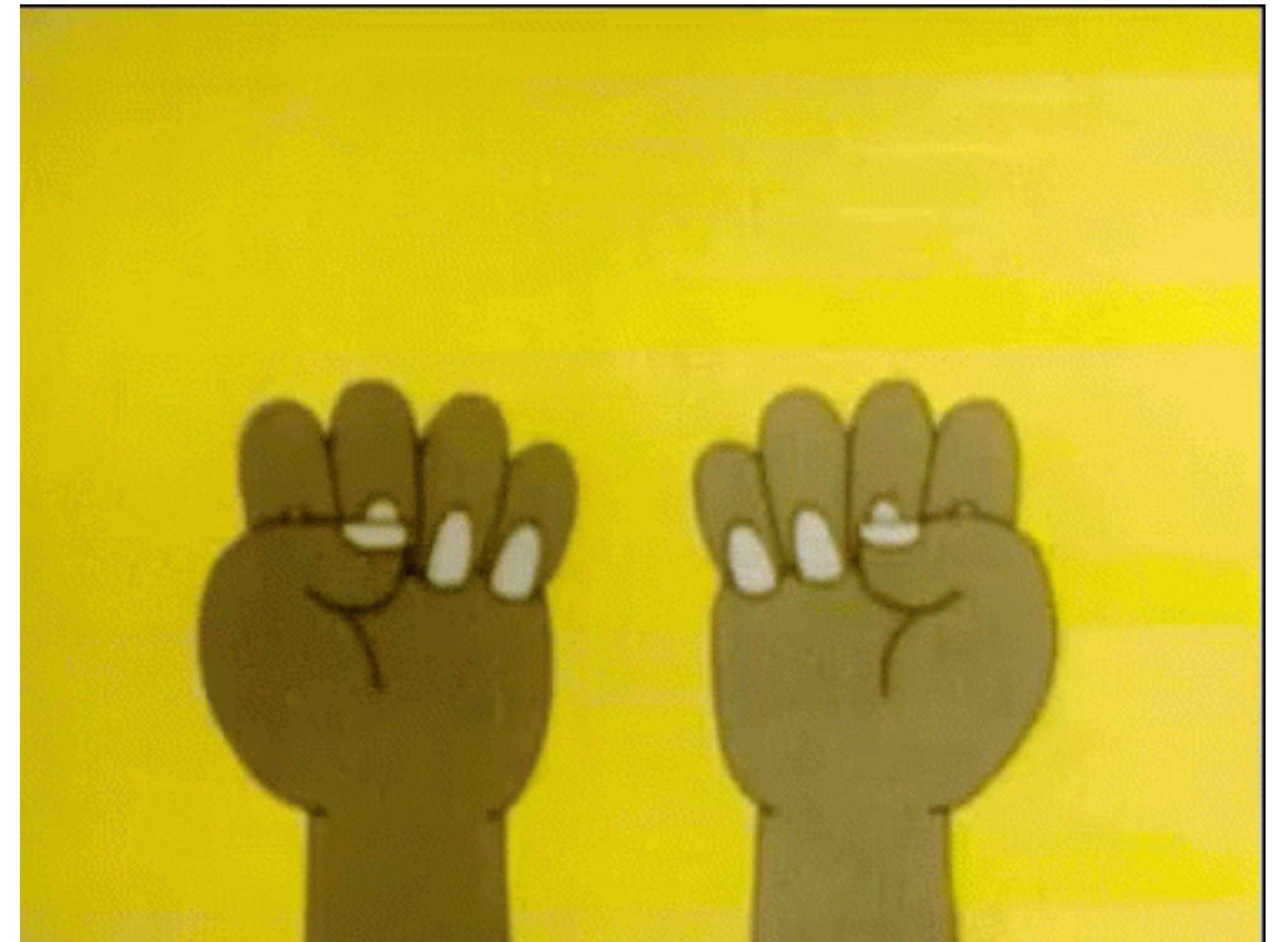
# Advanced algorithms

1. ACC\_K Algorithms
2. HIT Algorithm

Solve n-Interval problem

# Acc Algorithm

Approximate Cumulative Count



# Acc algorithm

- Family of algorithms that solves n-interval problem
- solves the problem using at most  $k$  hash tables for update and  $ACQ_{k+1}$  for queries
- The larger  $k$  is, The algorithm takes less space but is also slower
- Break the stream into frames of size  $n$  (maximal window size)
  - Any  $n$  sized window intersects with at most two frames



# $ACC_1$ algorithm

- Each block has a table that tracks how many times each item has arrived from the beginning of the frame
- Query at most 3 tables:
  - Within the frame - compute interval by subtracting 2 tables
  - If it crosses two frames, one additional query

wasteful?!



# $ACC_2$ algorithm

- Saves space at expense of additional table access
- Breaks each frame to  $\sqrt{n}$  sized segments
- At end of each segments, we keep level-1 table that counts item frequencies from the beginning of the frame
- level-0 tables computes frequency within a segment for each block

# ACC Algorithm

x	d	x			y	b,d,y	x,d	x	x		d		b		x	c
---	---	---	--	--	---	-------	-----	---	---	--	---	--	---	--	---	---

$ACC_1$

x 2	x 2	x 2	x 2	x 2	x 2	x 3	x 4	x 5	x 5	x 5	x 5	x 5	x 5	x 5	x 1	x 1
d 1	d 1	d 1	d 1	d 2	d 2	d 3	d 3	d 3	d 3	d 4	d 4	d 4	d 4	d 4		c 1
			y 1	y 2	y 2	y 2	y 2	y 2	y 2	y 2	y 2	y 2	y 2	y 2		
				b 1	b 1	b 1	b 1	b 1	b 1	b 1	b 1	b 1	b 2	b 2		

$ACC_2$

x 2			v 1	h 1		x 1	x 2	x 3			d 1		d 1		x 1	x 1
d 1				d 1		d 1	d 1	d 1					b 1			c 1
				v 2												

$Table_0$

$Table_1$

x 2
d 1

x 2
d 2
v 2
b 1



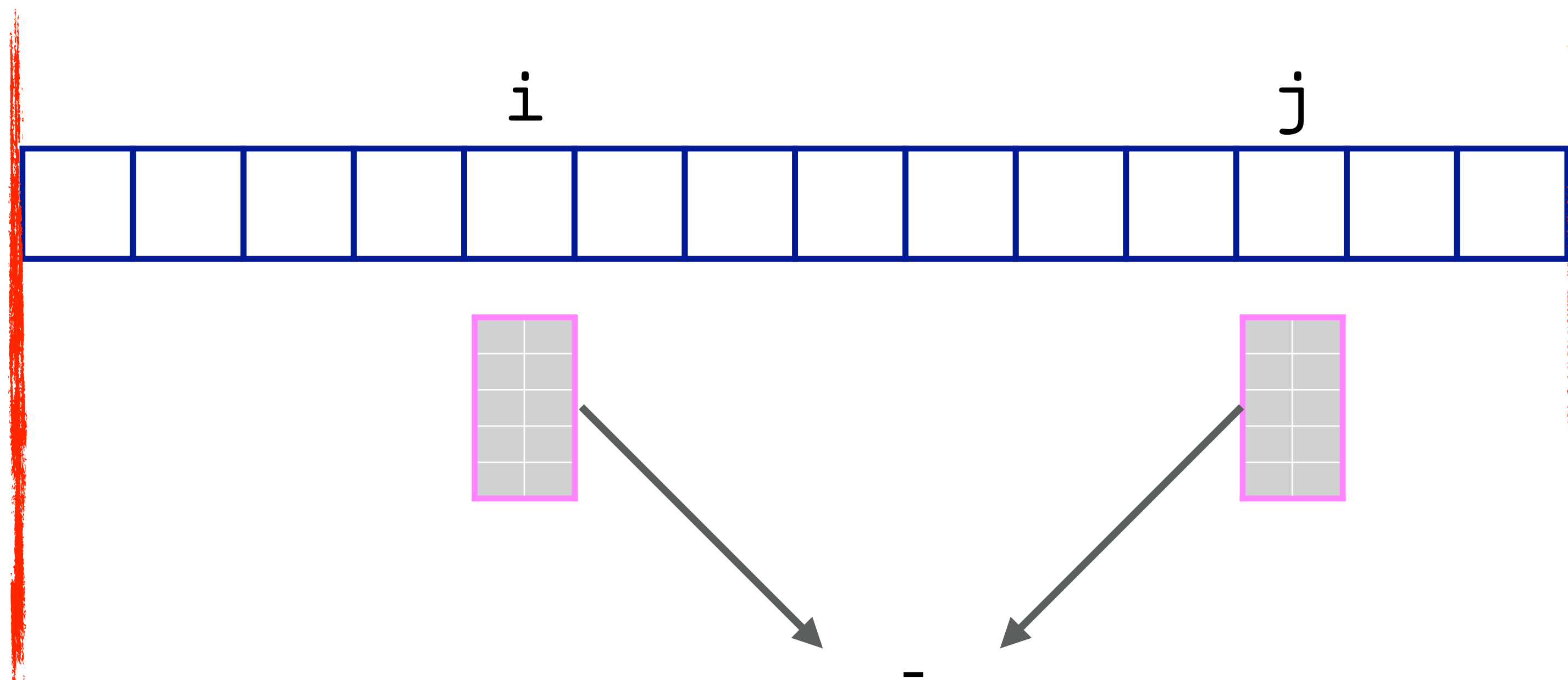
$\sqrt{n}Table_0$

x 5
d 3
v 2
b 1

x 5
d 4
v 2
b 2

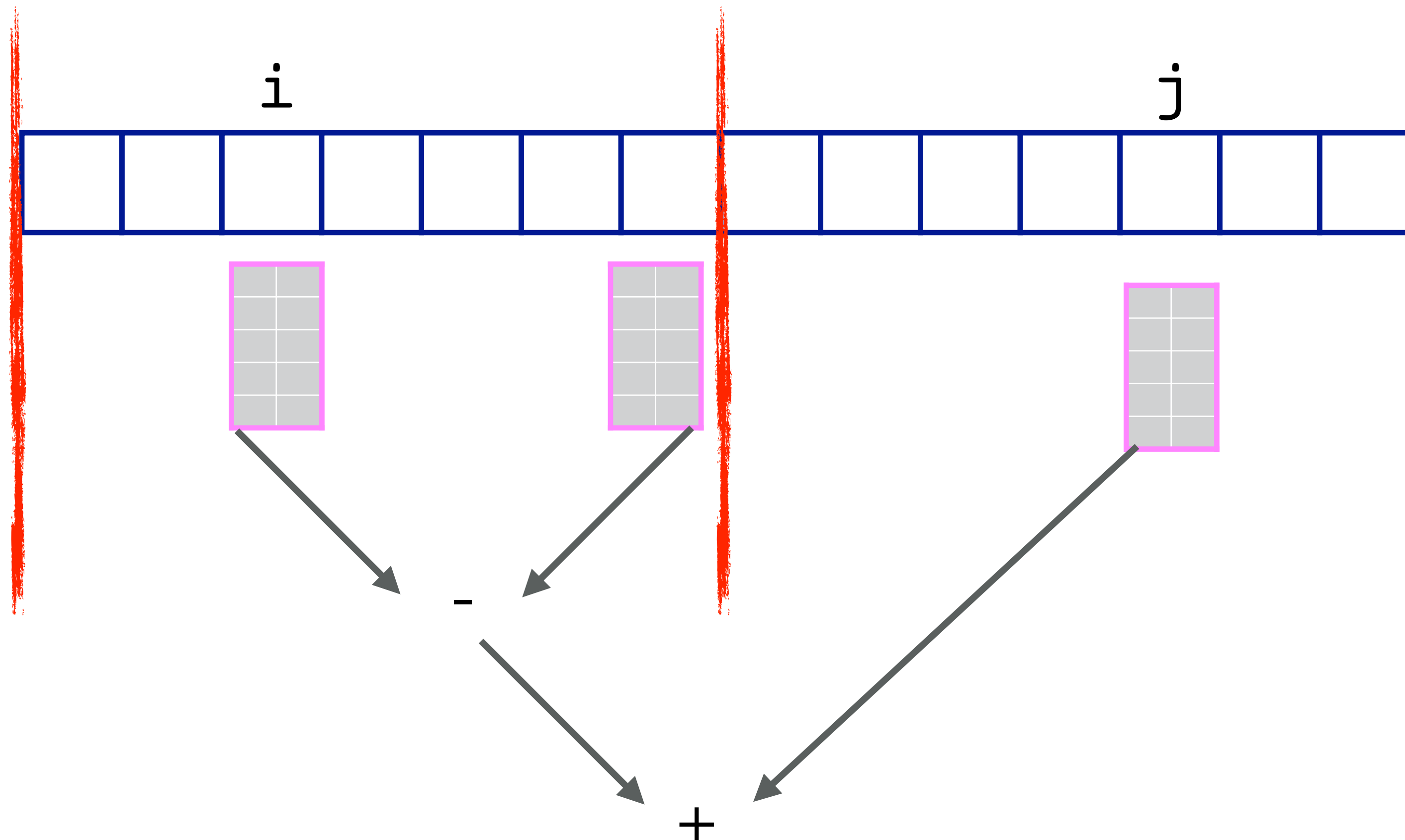
# Answering interval Frequency query

- For  $[i, j]$ , let  $\text{block}_i$ ,  $\text{block}_j$  be the relevant blocks
- If  $\text{block}_i$  and  $\text{block}_j$  are in the same frame:



# Answering interval Frequency query

- If block\_ $i$  and block\_ $j$  are NOT in the same frame:

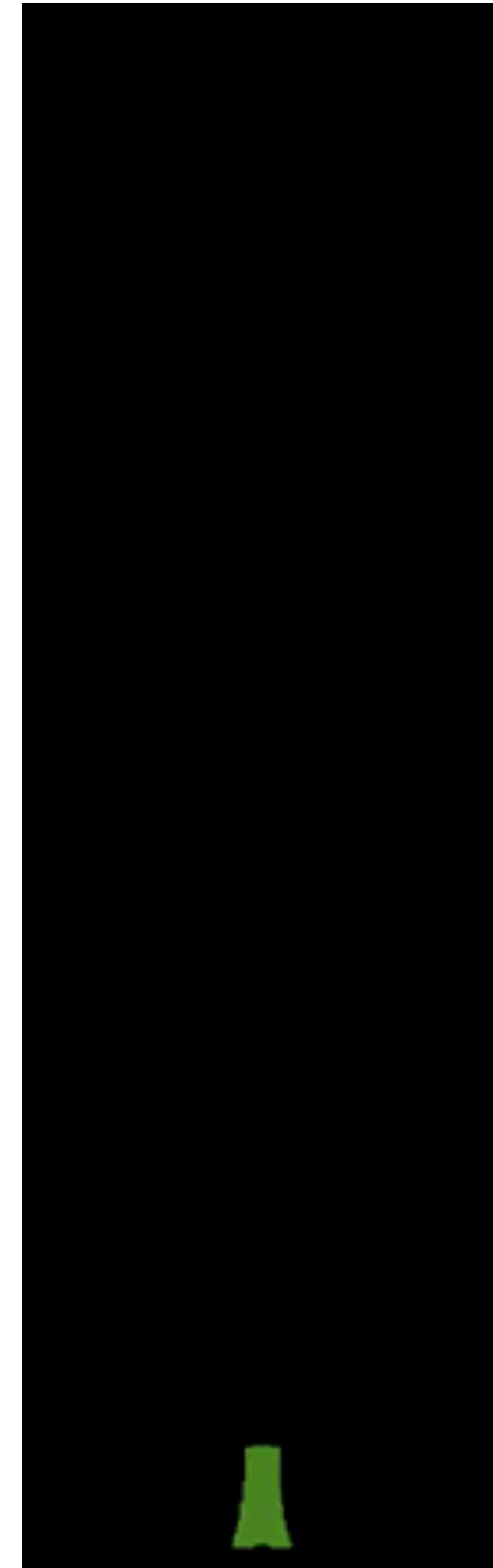


# Answering interval Frequency query

- Corner case:  $ghostTable[l]$  table may includes blocks that already left the table  $level_l$
- **Solution:** Maintain ghostTables for leaving segments
  - $ghostTable[l]$  - contains  $ghostTable$  if last leaving block
- Subtract the corresponding ghostTable as well  $ghostTable[l]$   $level_l$

# Hit Algorithm

Hierarchical Interval Tree



# HIT Algorithm

- Uses hierarchical tree structure
- Nodes stores partial frequency of its sub-tree
- tracks how many times each item arrived within block

$level_0$  of tracks how many items arrived between

$level_l$   $block_i$

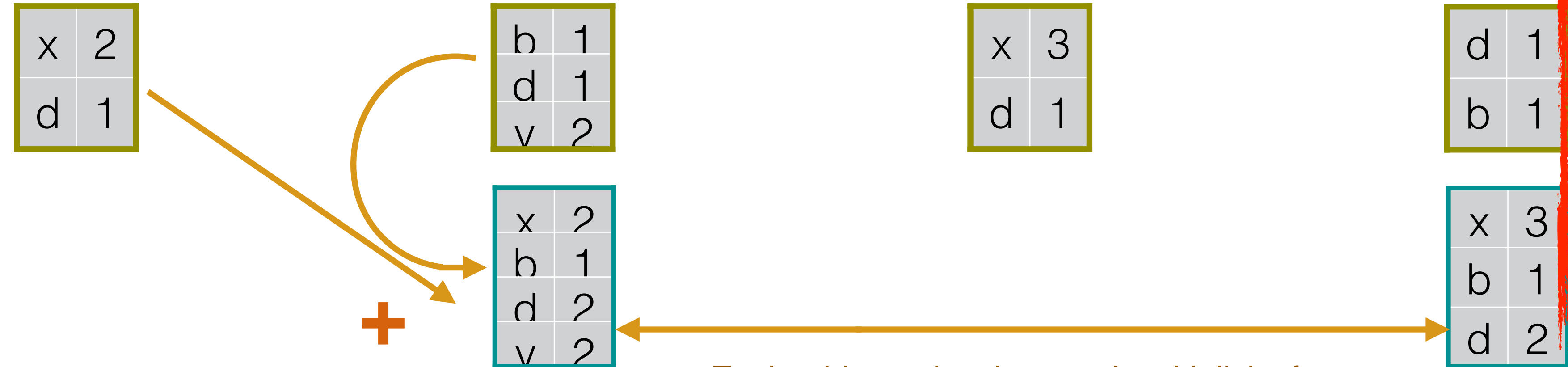
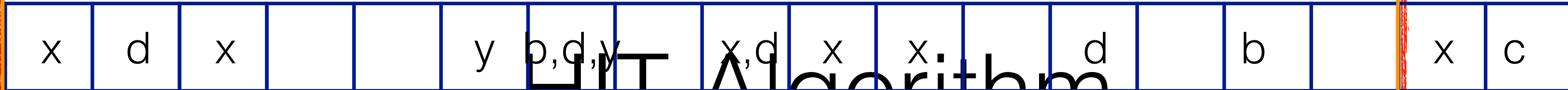
$[block_{i-2^l+1}, block_i], 0 < l \leq trailing\_zeros(i)$



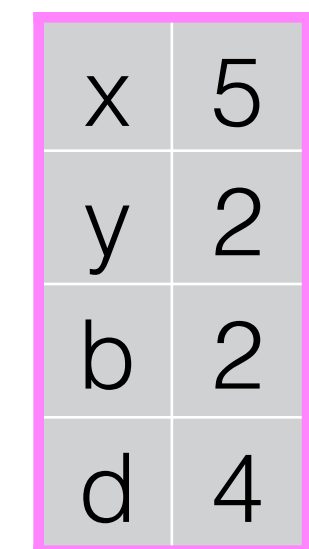
# HIT Algorithm

- Each level contains tables for half the blocks of previous level
- Higher levels of the tree allow efficient time computation

# Hit Algorithm



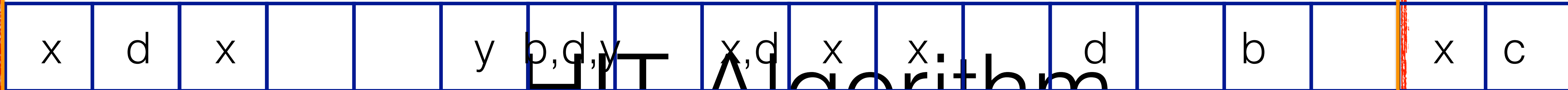
Each table tracks elements' multiplicity from the previous same level block



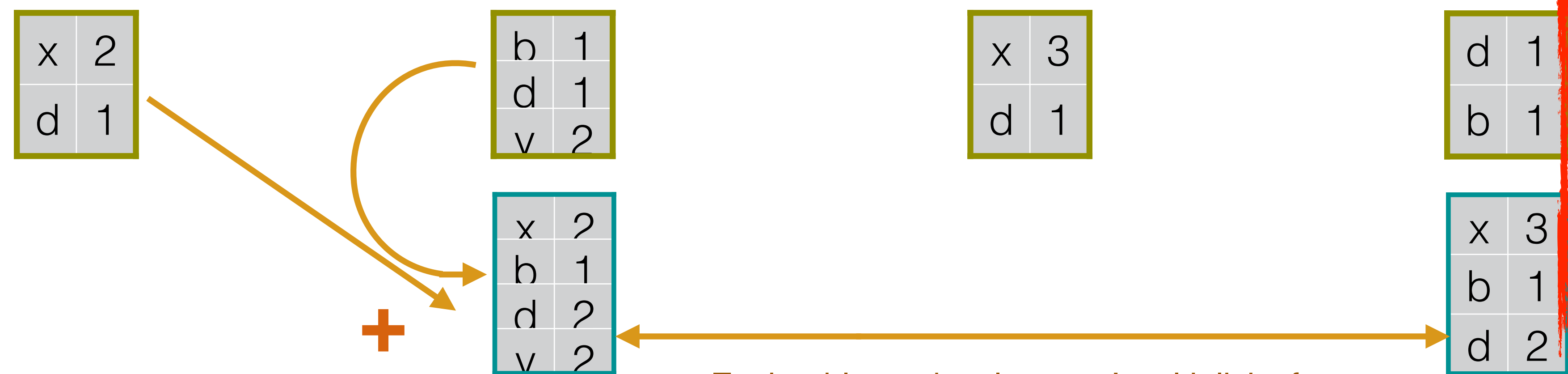
# Answering interval Frequency query

- For  $[i, j]$ , let  $\text{block}_i, \text{block}_j$  be the relevant blocks
  - Scan backward from  $\text{block}_j$  to  $\text{block}_i$ , Greedily using the highest possible level at each point.
  - If  $\text{block}_j > \text{block}_i$  all tables are valid
  - Otherwise, use  $\text{level}_0$  between  $\text{block}_0$  to  $\text{block}_j$  and compute  $\text{block}_i$  to  $\text{block}_n$  as before

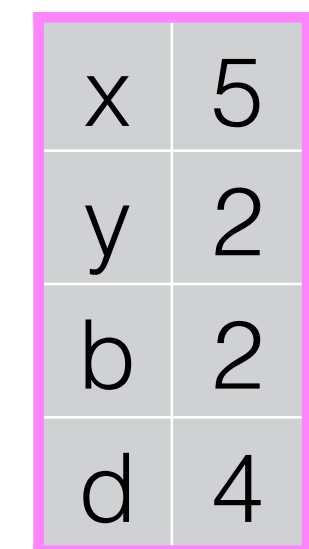
# Hit Algorithm



3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 2



Each table tracks elements' multiplicity from the previous same level block



# Answering interval Frequency query

- Query computation takes at most  $2 \log n$  steps
- Corner case: Content of a table may refer to a departing block
- Solution: Choose always the highest valid level of valid tables

# Evaluations



# Setup

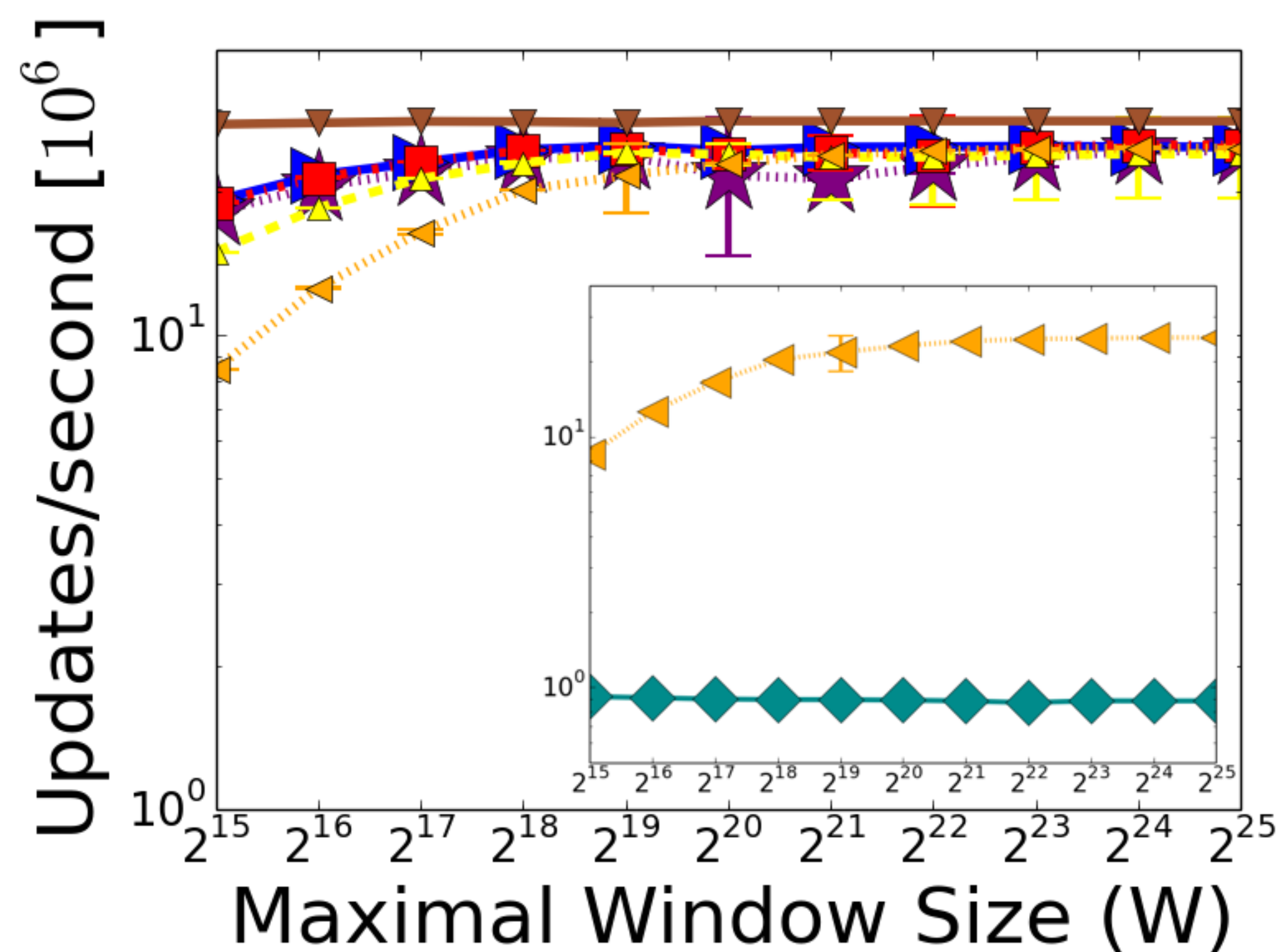
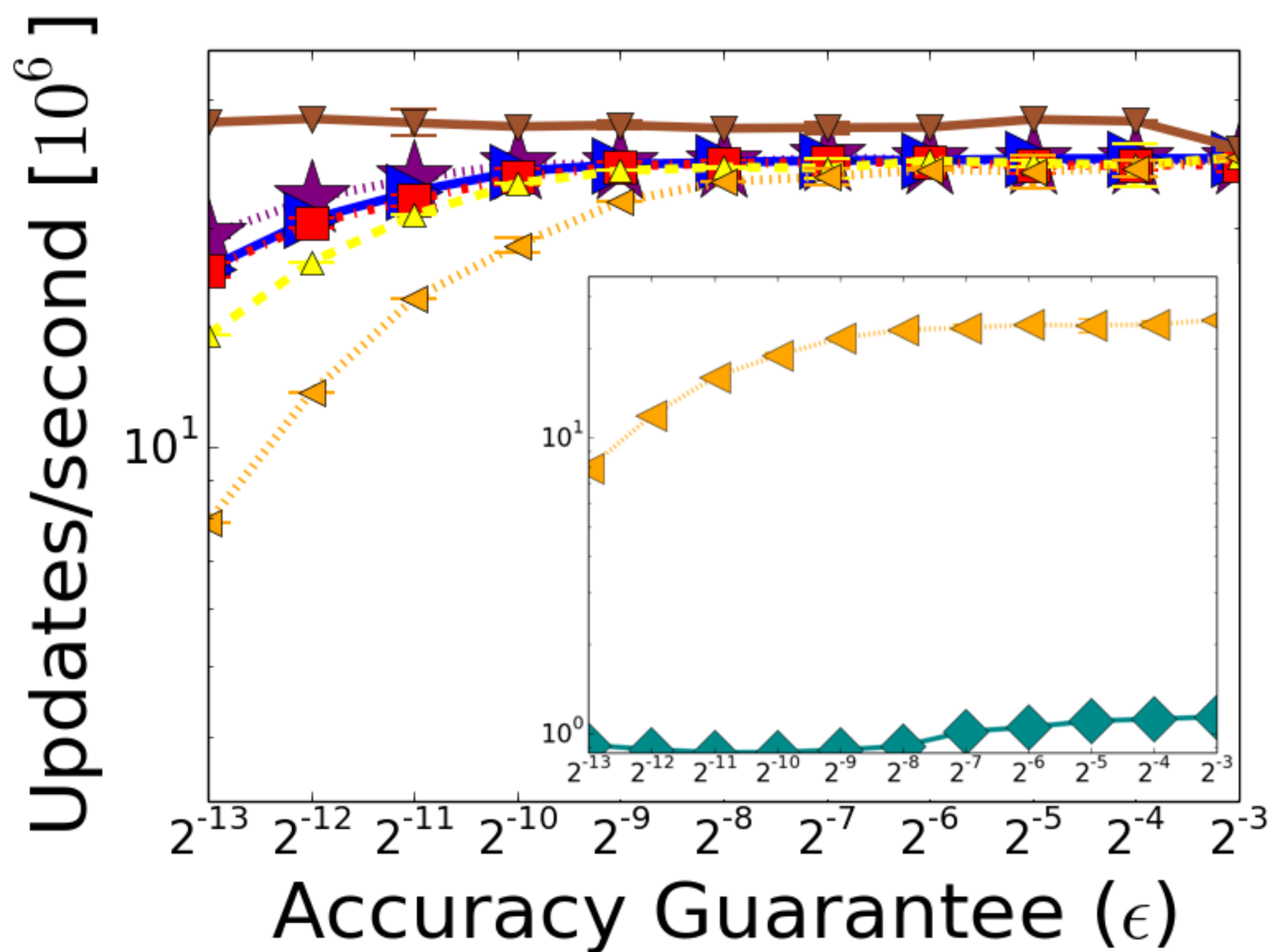
- C++ implementation
- ECM is configured with
- Backbone dataset

$$\delta = 0.01\%$$

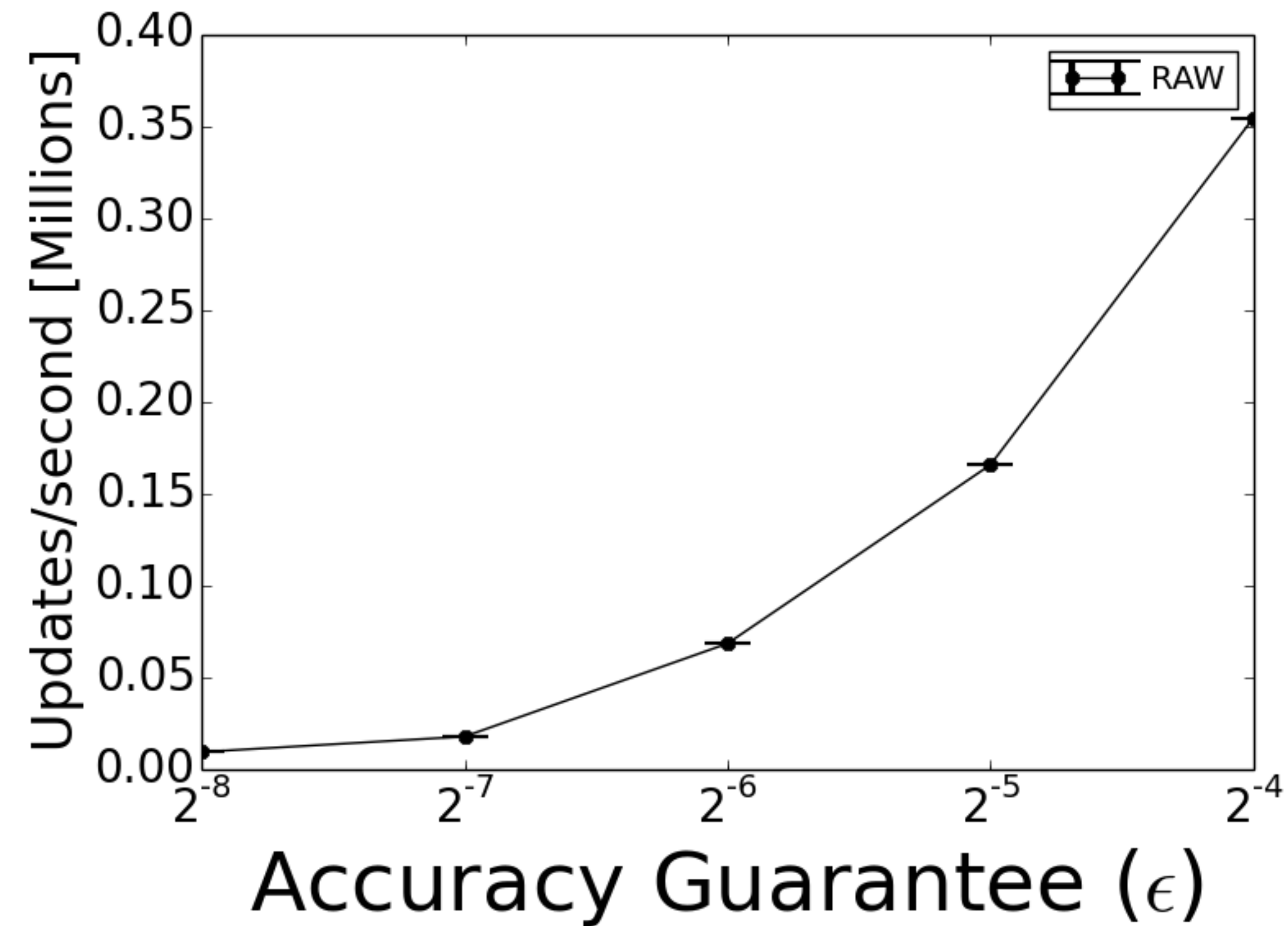
`W= 2^20, epsilon = 2^-8, interval size = 1%*Window`



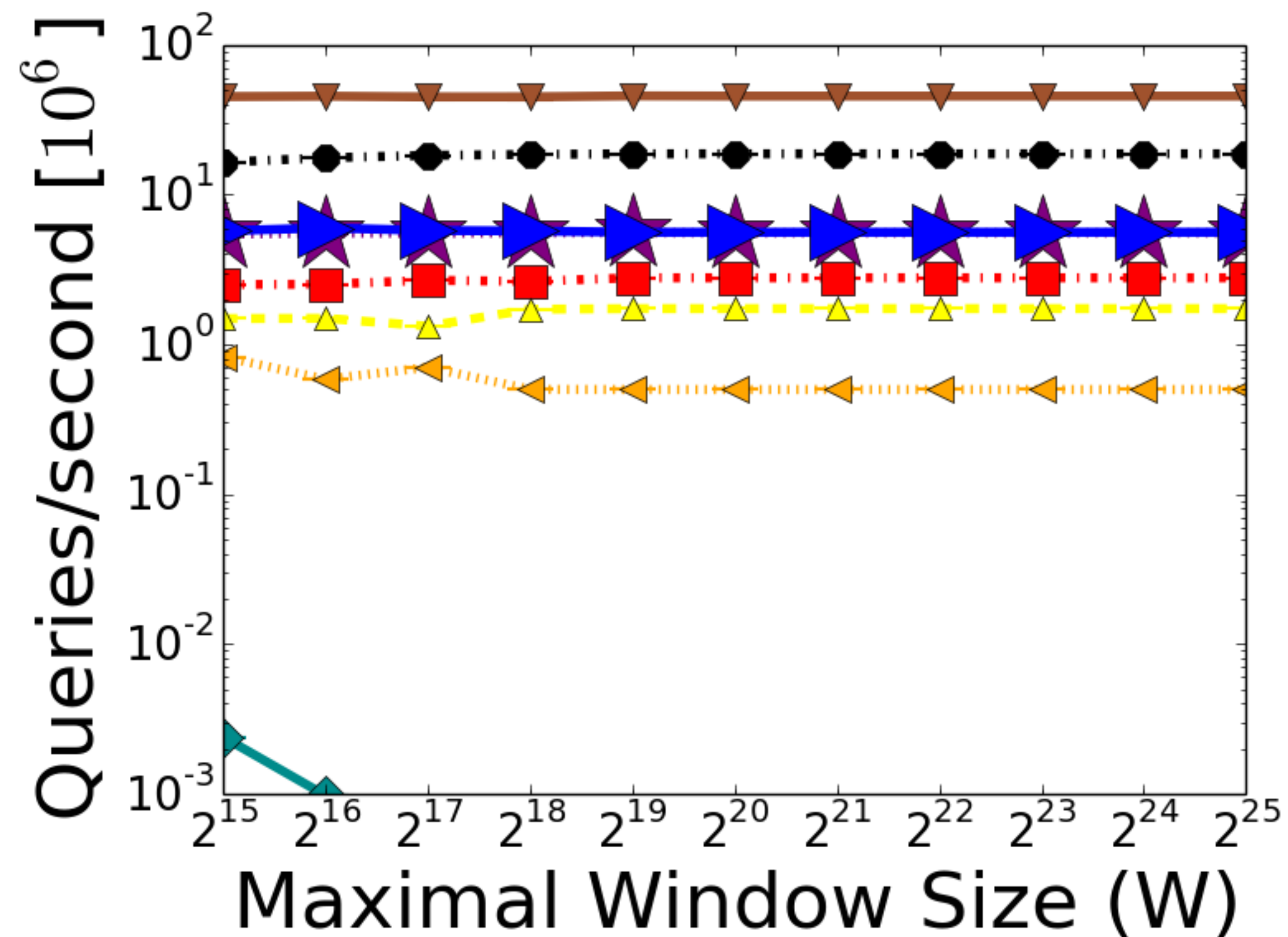
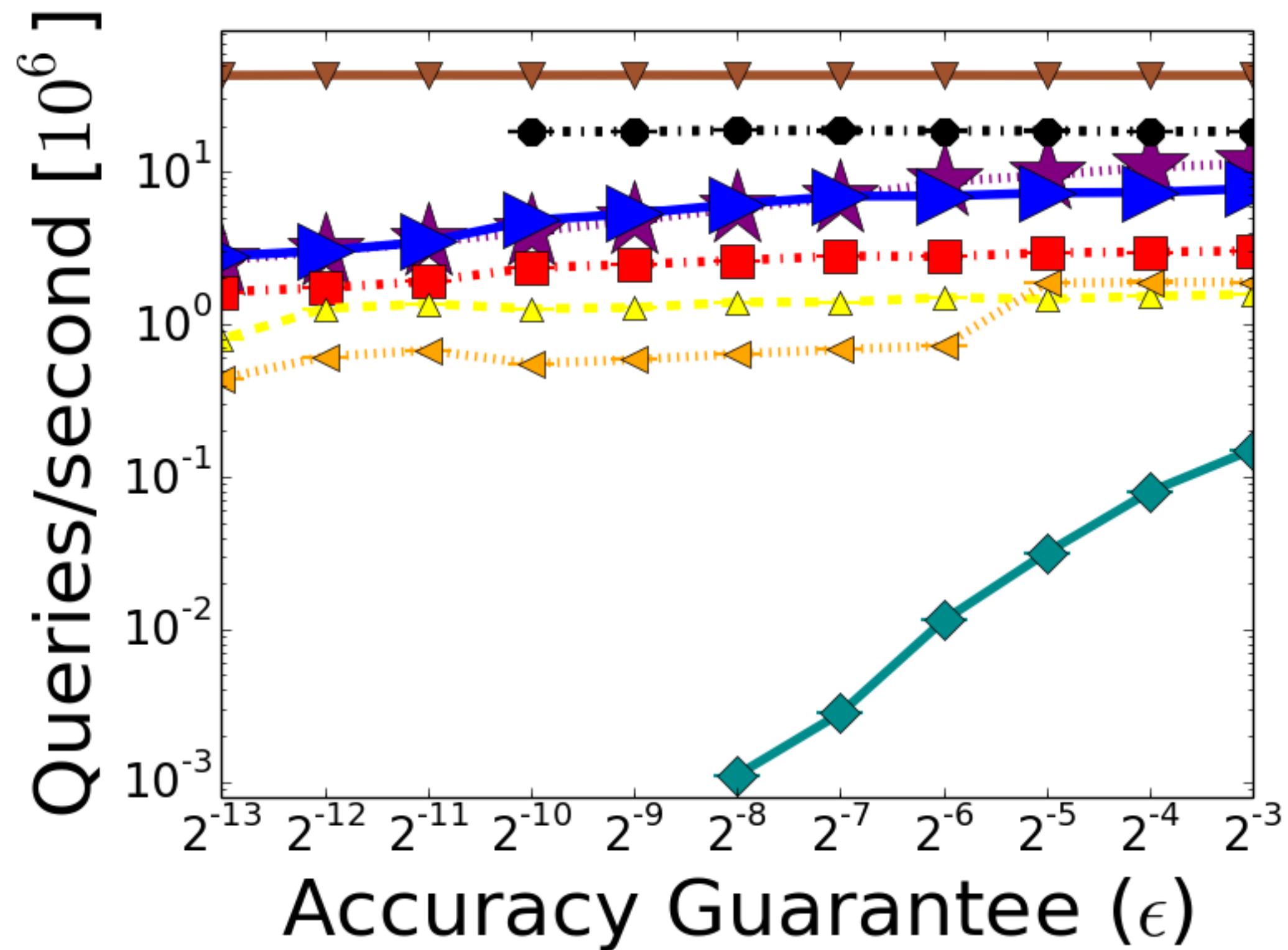
# Update Speed comparison



# Update Speed comparison

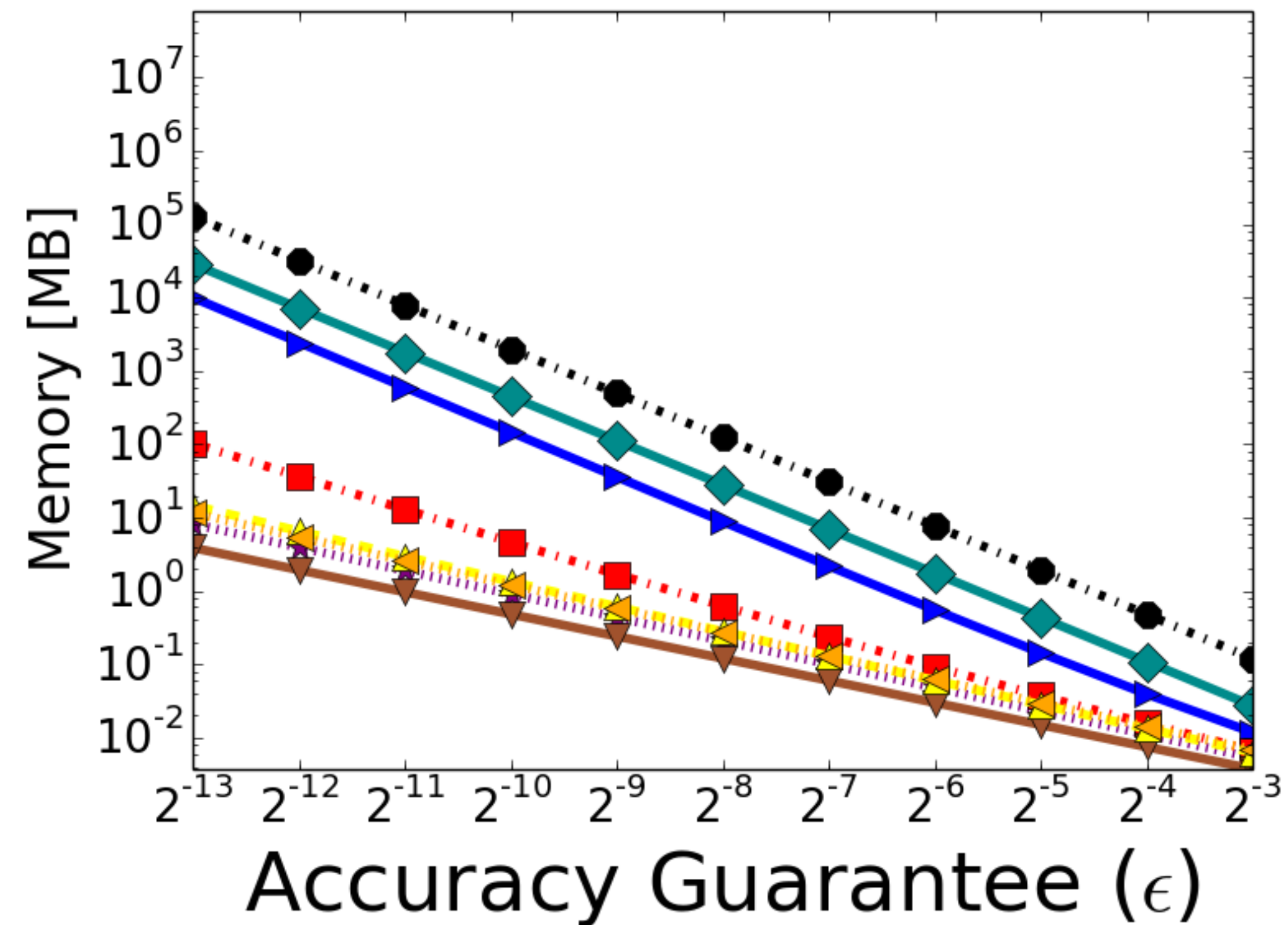


# Query Speed comparison



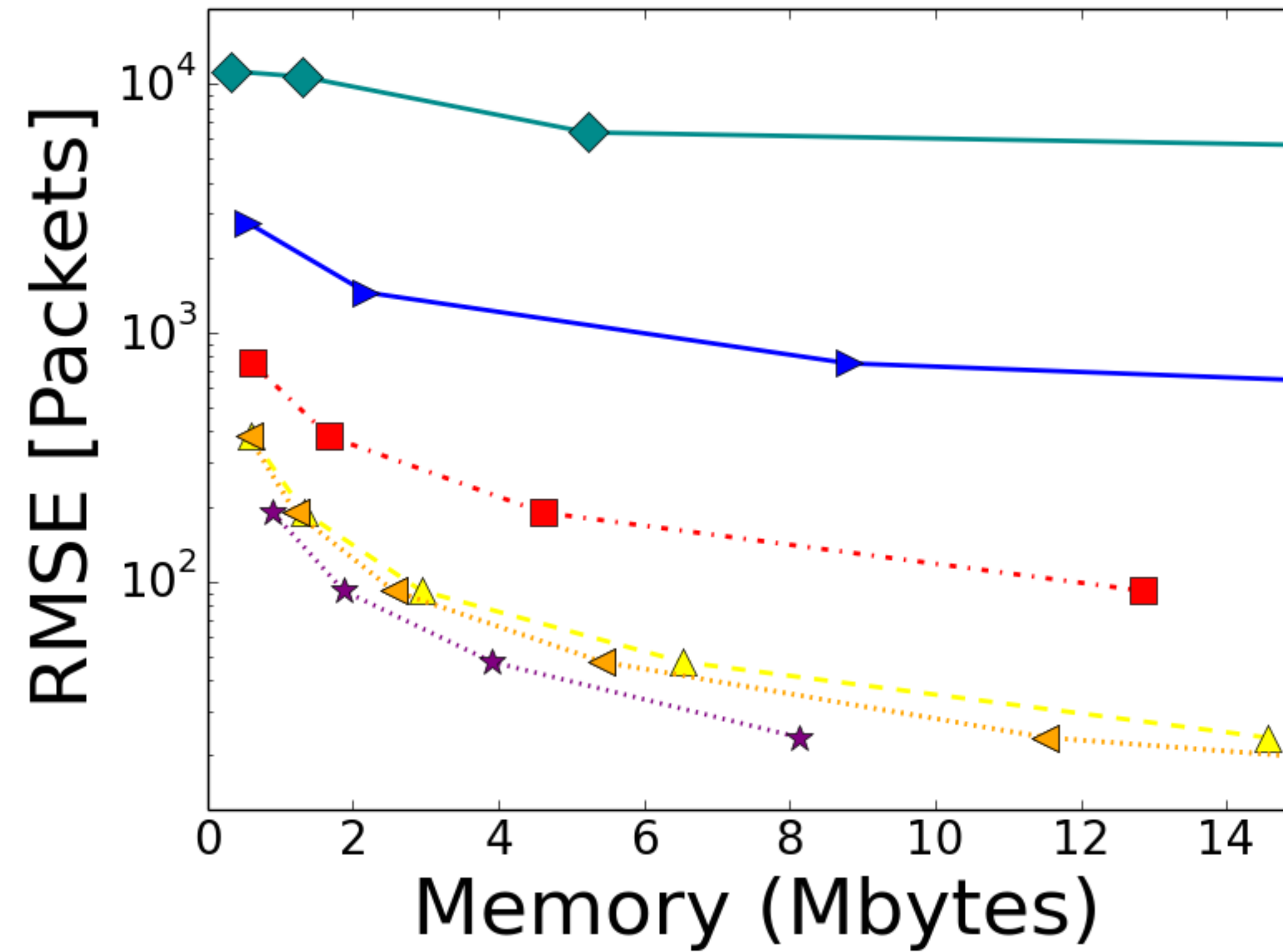
Legend: HIT (purple stars), ACC<sub>1</sub> (blue triangles), ACC<sub>2</sub> (red squares), ACC<sub>4</sub> (yellow triangles), ACC<sub>8</sub> (orange triangles), WCSS (brown inverted triangles), ECM (teal diamonds), RAW (black circles).

# Memory Consumption

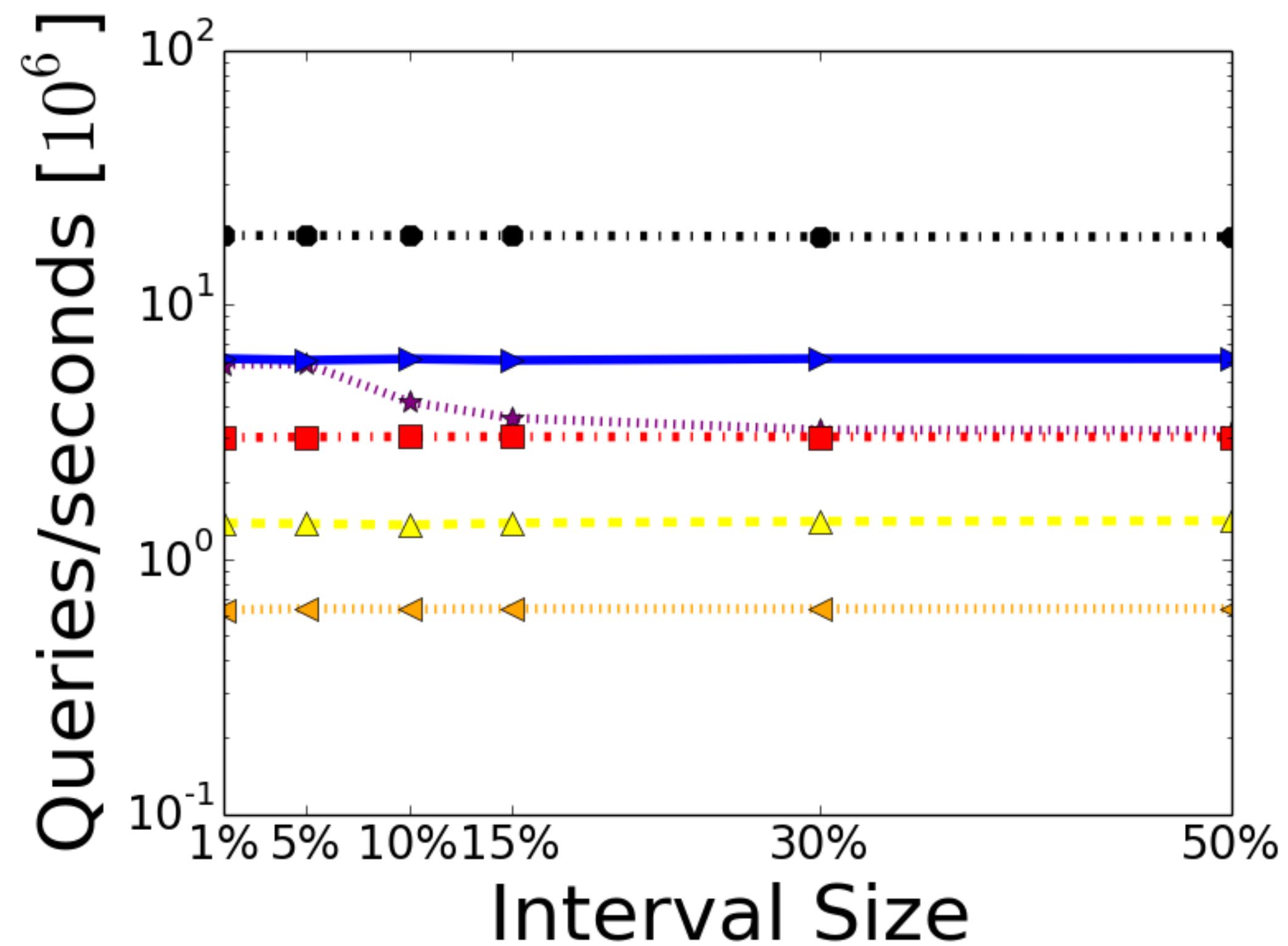




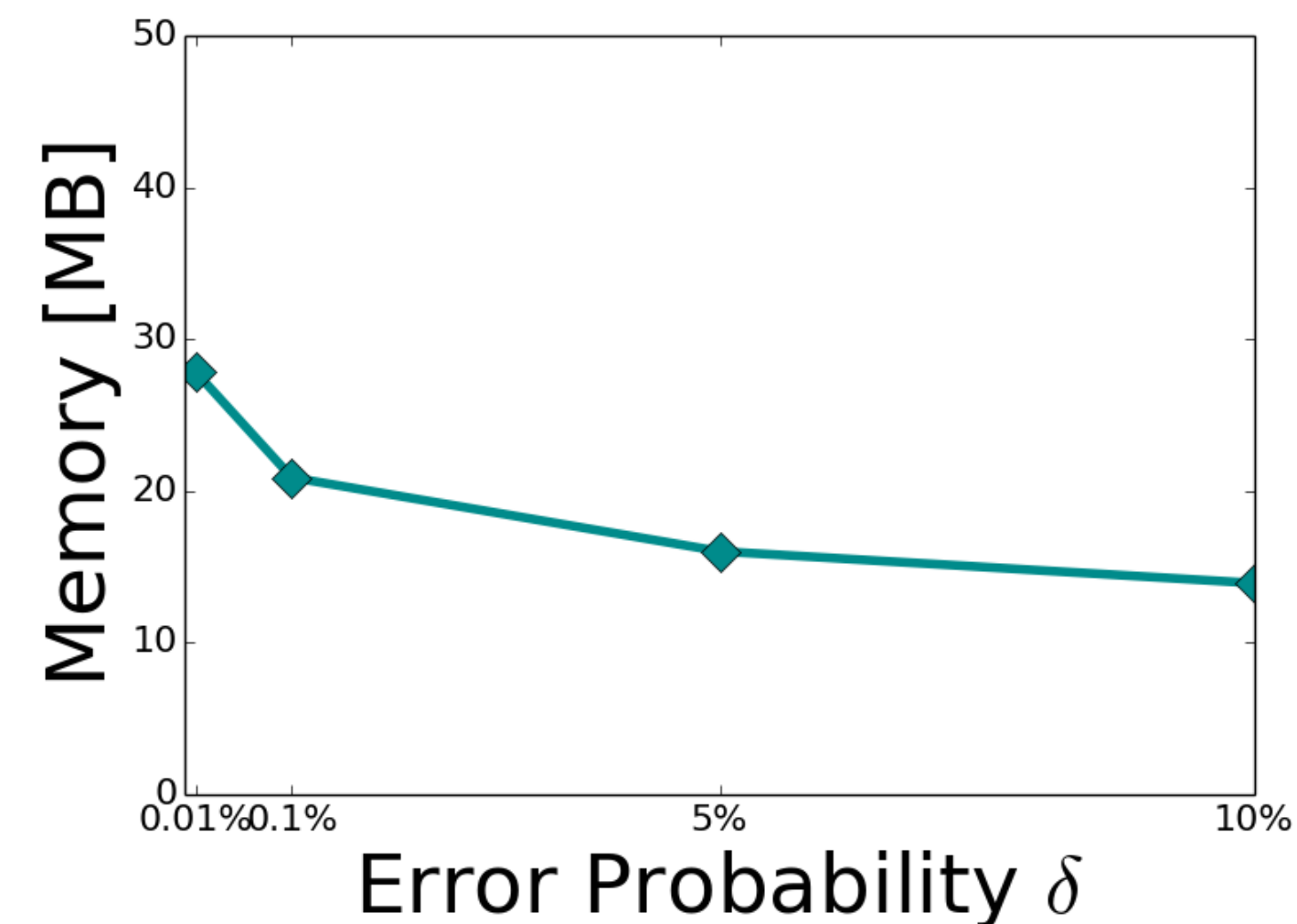
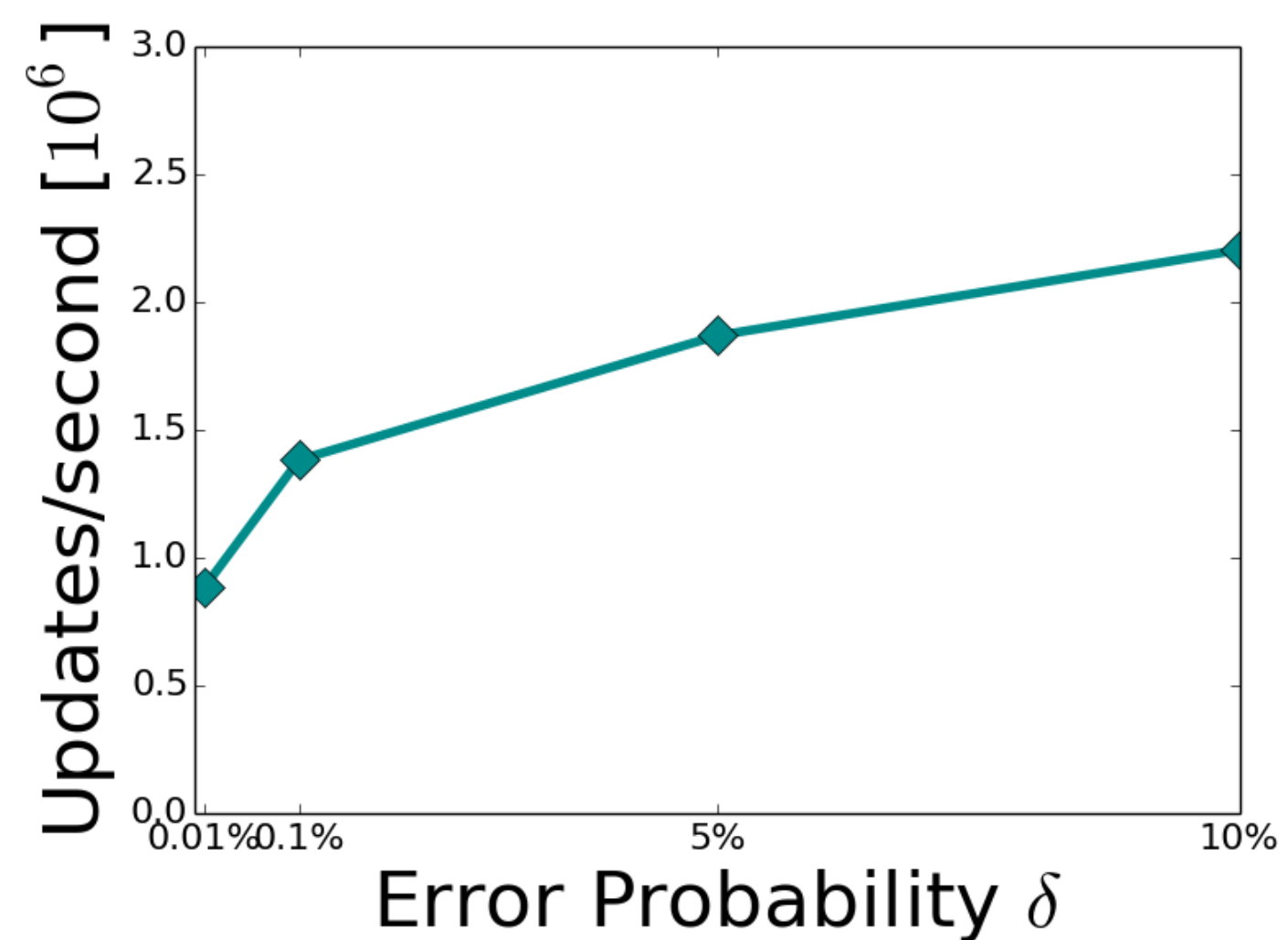
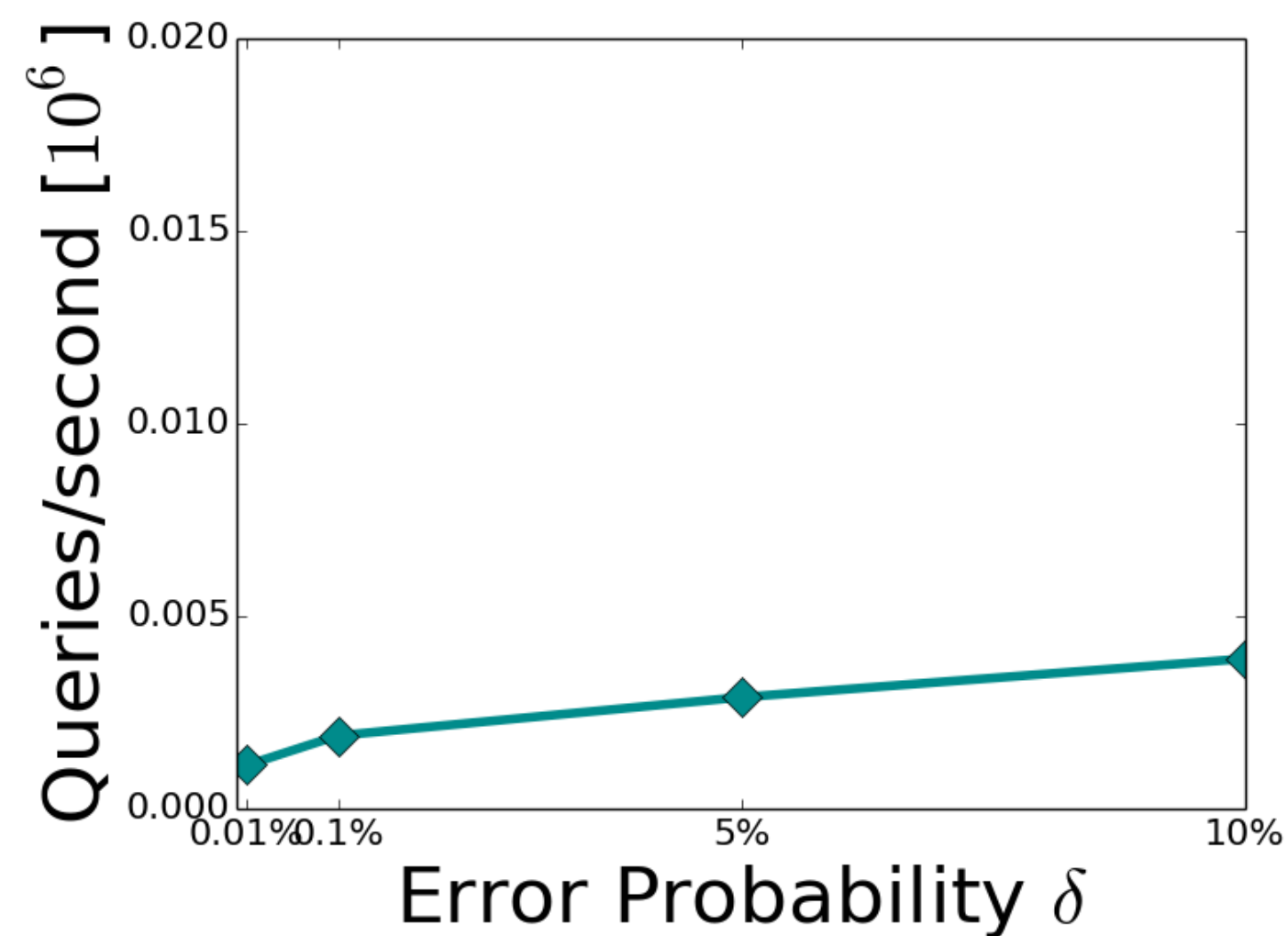
# Observed error



# vart interval sizes



# ECM space and performance comparison





Thank YOU!