Stream Frequency over Interval Queries

Rana Shahout

joint work with: Ran Ben Basat, Roy Friedman

VLDB 2019

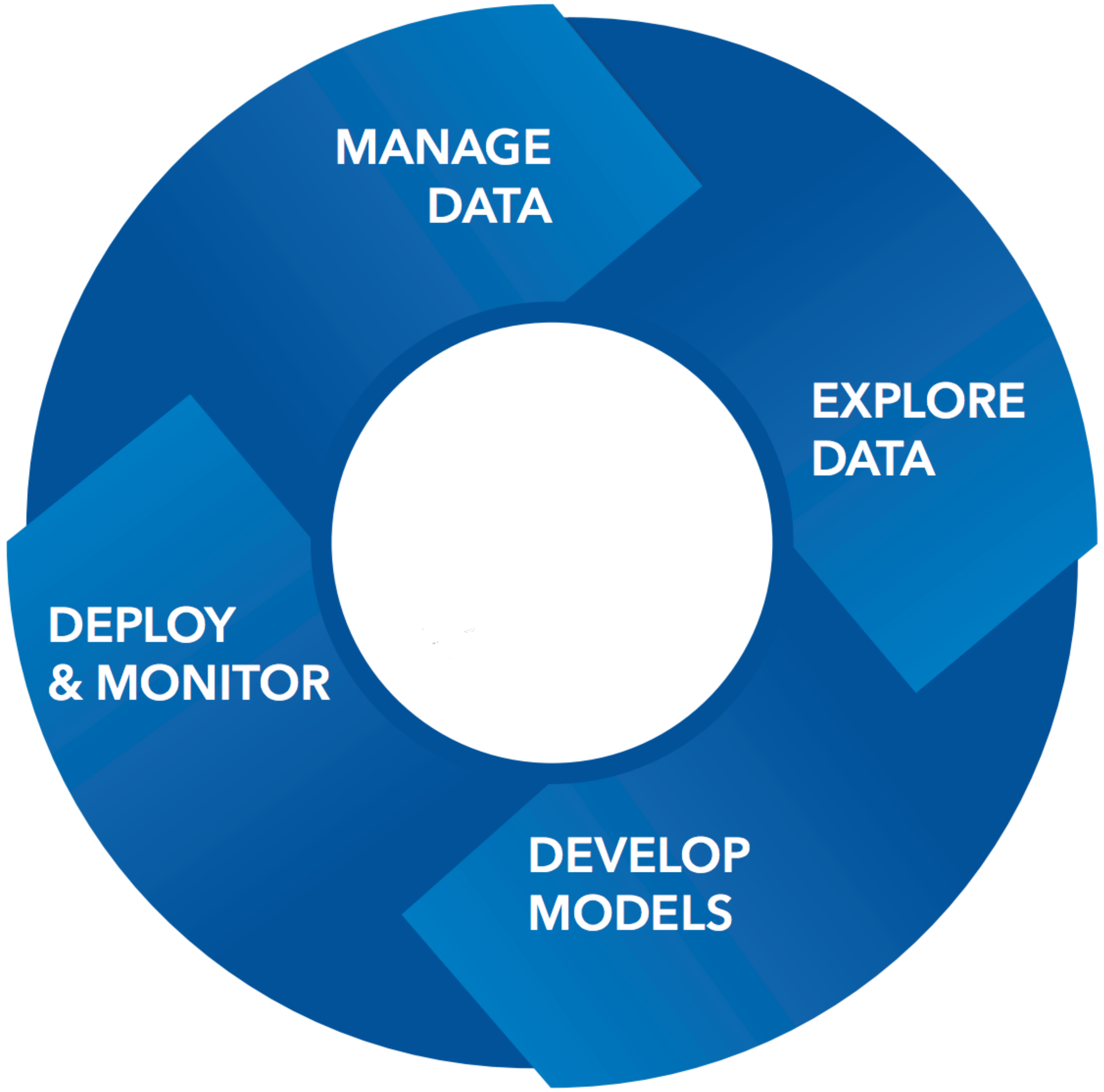
- Example: Traffic to popular websites (Amazon, Google, Facebook)

- Problems, Stream is hard to:
    - Store
    - Process
    - Transfer

Requirement: keep up with the rate of incoming data (line speed)

Why Data Management is Important

Essential for many applications such as: network monitoring, financial data trackers,…



Stream Formal Definition

- Given a universe U, a stream $S = x1, x2, \ldots \in U_*$ is a sequence of universe elements
- We are interested in computing a function f on S
- Examples of interesting f functions include **frequency**, heavy hitters, and count distinct

Example of Data Monitoring

- "How many times an item has appeared in the stream??"
- **Naive Solution**: Allocate an exact counter for each element
- **Problem**: Memory constraints

SRAM vs. DRAM

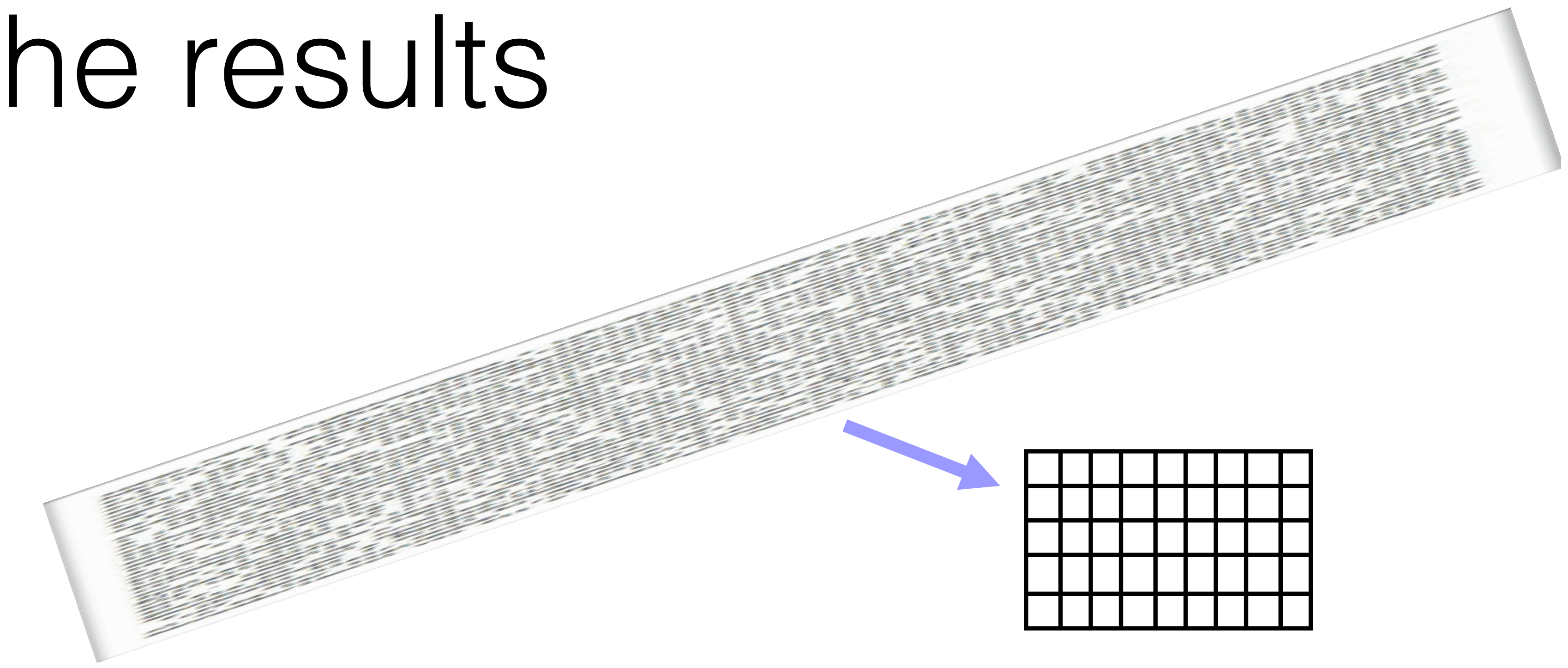
Our solutions focus on minimizing the number of counters needed, thereby allowing the system to monitor a large number of elements using only SRAM.

- Almost all algorithms are approximate, answer with error and guarantee a bound on the error
- Given function f, an approximate algorithm supports two operation:
    - **Add**(x) - append x to stream S
    - **Query** - return estimation of f on S

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



Challenges:

- Determine what portion to keep in the limited space
- Determine how to efficiently compute the summary in data update

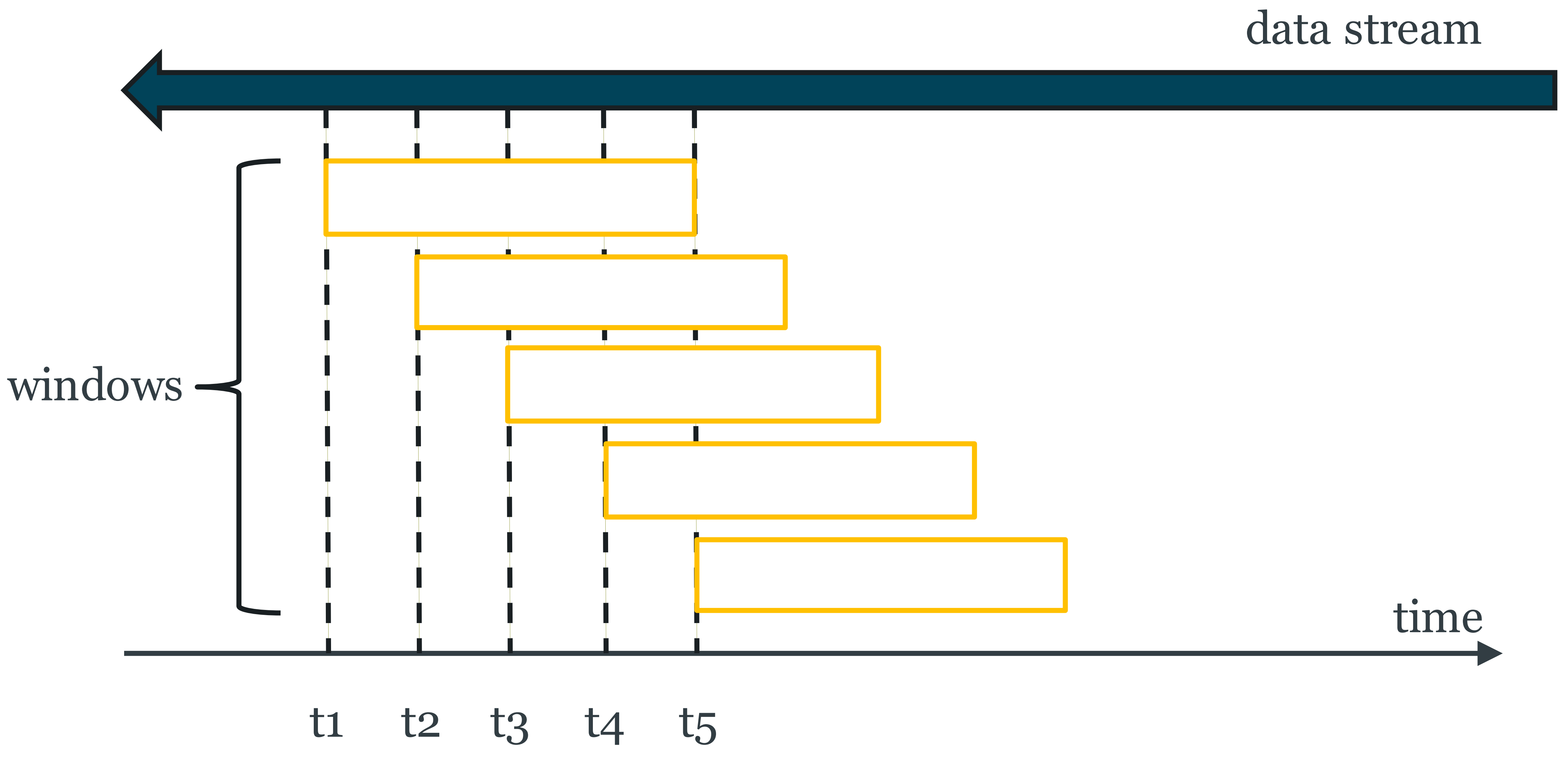
Within the last million items..

how many times a user bought a  between 202 and 172 most recent items?

how many times a user bought  between 505 and 251 most recent items?

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



The Problem With Existing 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

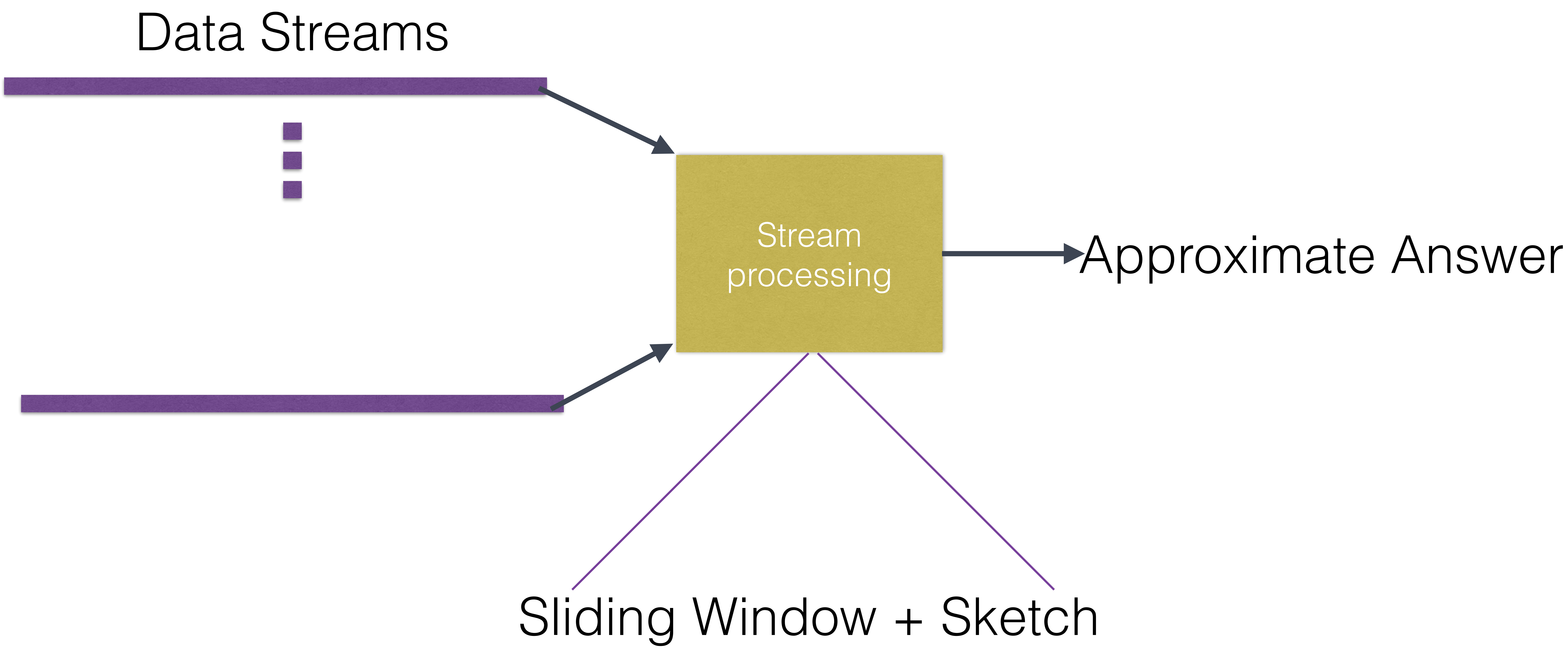
Problem Definition

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

$$(W,\varepsilon)-IntervalFrequency:$$

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

Computation Model



Existing Works - ECM

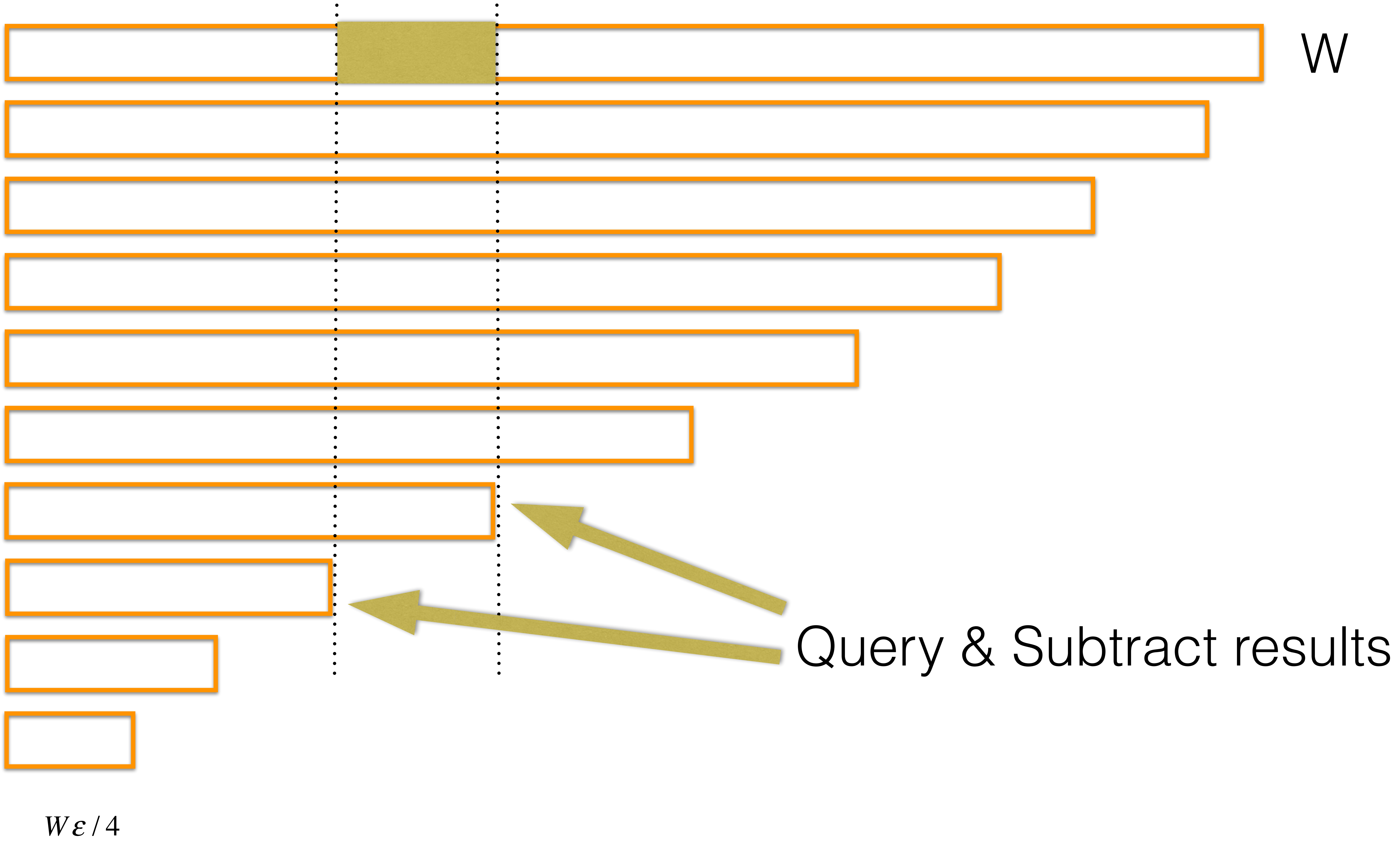
- ECM combines Count-Min Sketch with Exponential Histograms
- Count-Min Sketch is a stream sketch for estimating item frequency
- Exponential Histograms is a sliding window counter that can guarantee a bounded relative error
- ECM sketch replaces each Count-Min counter with an Exponential histogram

- Naive Solution: RAW algorithm
- Advanced Solutions:
  - ACC_K algorithm
  - HIT algorithm

Naive Solution: Raw

- Uses several instances of a black box algorithm that solves frequency estimation over a 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

Query interval



RAW vs. Existing Solution (ECM)

- Both consume same amount of memory
- Raw achieves a better query performance, (update time of ECM is better)

N-Interval Problem

- Arriving elements may be inserted to blocks

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

- IntervalQuery(x,i,j): Compute **exact** number of blocks x appears in between blocks `i,j`

N-Interval Problem → Reduction → $(W,\varepsilon)-IntervalFrequency$

Decides when to insert an item to block

N-Interval Problem Definition

- N-Interval Problem: Block Interval Frequency
- Add(`x`): Given an element `x`, append it to stream
- EndBlock(): New block inserted, old block leaves
- IntervalQuery(`x,i,j`): Return the number (without error) of blocks `x` appears in between blocks `i,j`

Reduction

1. Break the stream into $\mathtt{W}$ 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

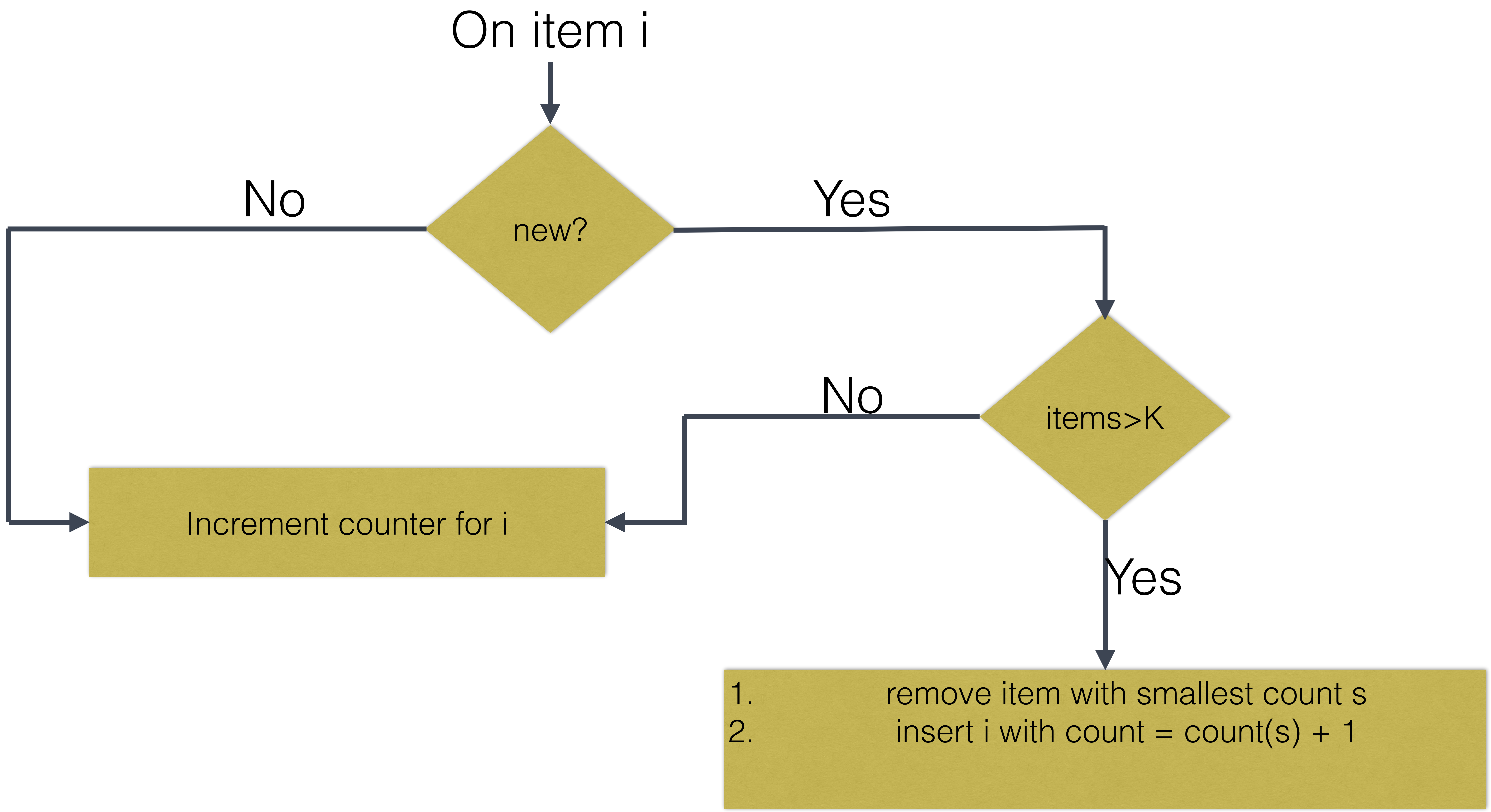
    1. Whenever a counter reaches an integer multiple of the block size, associated it to most recent block

    2. When the frame ends, flush Space Saving instance

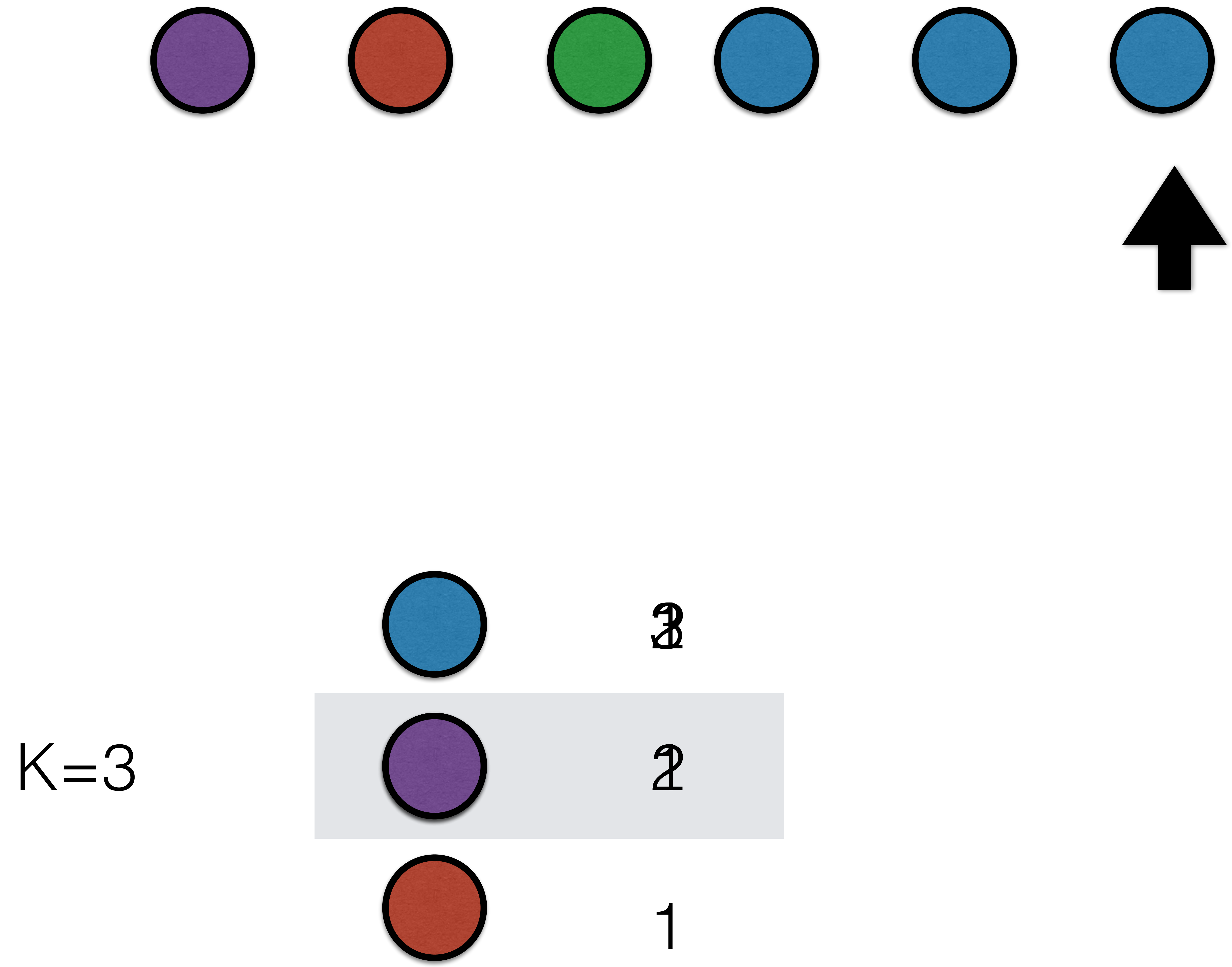
Space Saving Model

- Counter algorithm
- Keep **`k`** items and counts initially zero
- Count first **`k`** distinct item exactly
- Only over-estimation errors
- Frequency estimation is more accurate for significant elements

Space Saving Model



Space Saving Algorithm



Implementing ADD(x)

x

Space Saving

offset:0

x

If result mod block size = 0

if offset mod block size =0
EndBlock

Implementing IntervalFrequency(`x,i,j`)

1. Compute relevant blocks numbers

2. Call Query of N-Interval problem

3. Return block size `*` result

Advanced Algorithms

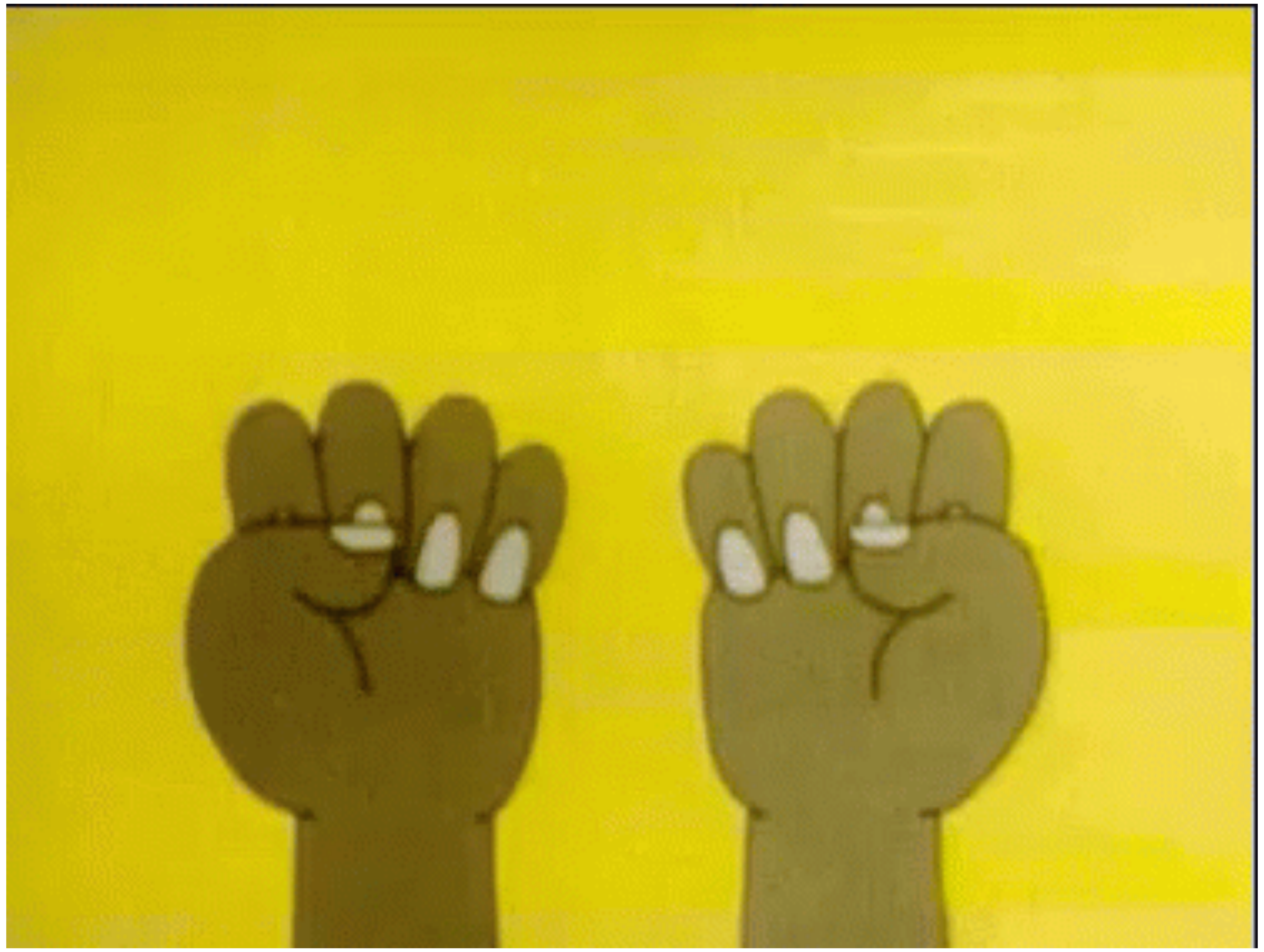
1. ACC_K Algorithms

2. HIT Algorithm

Solve N-Interval problem

Acc Algorithm

Approximate Cumulative Count



ACC Algorithm

- Family of algorithms that solves N-Interval problem
- ACC_k solves the problem using at most **k** tables for update and **2k+1** for queries
- The larger **k** is, The algorithm takes less space but is also slower

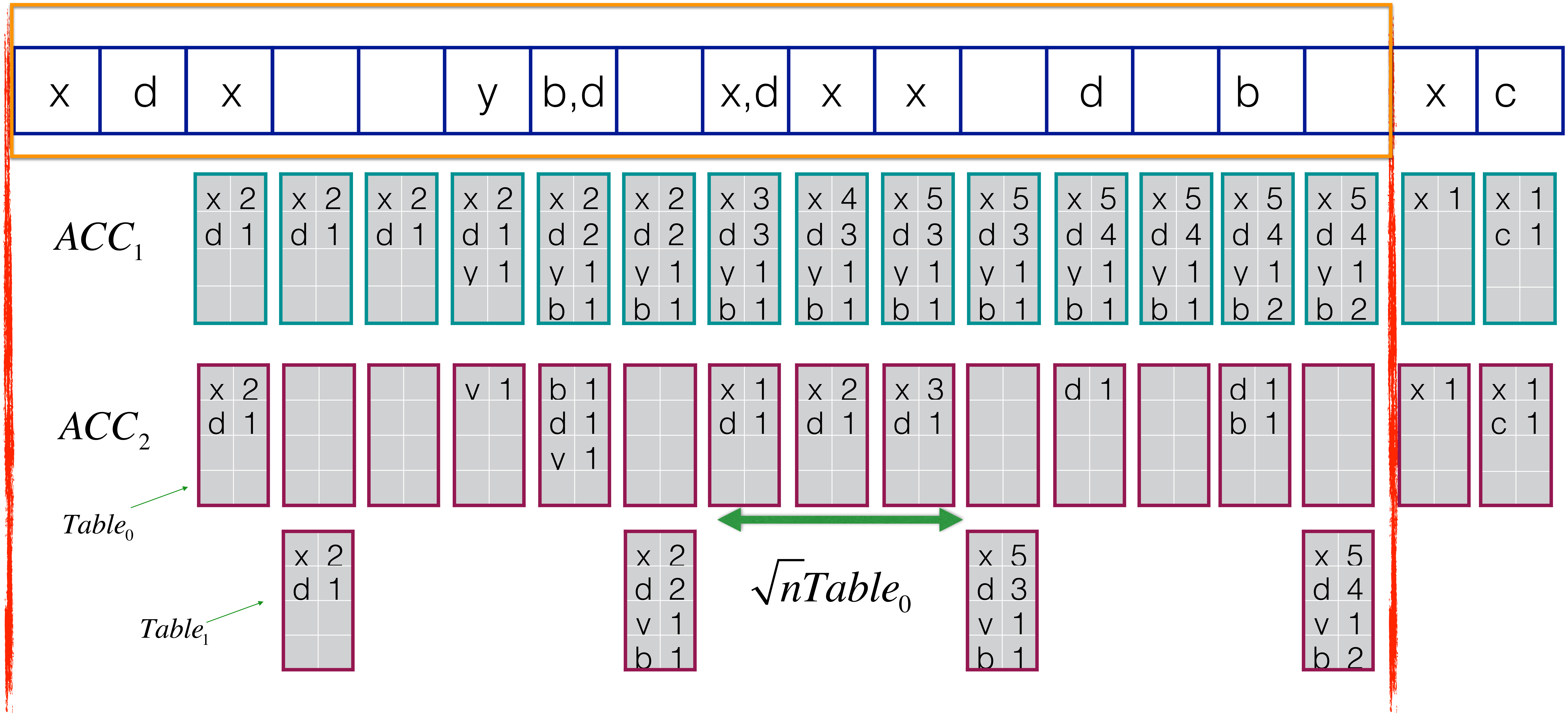
ACC_1 Algorithm

- As part of the reduction we break the stream into W sized frames and divide each frame into blocks.
- 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

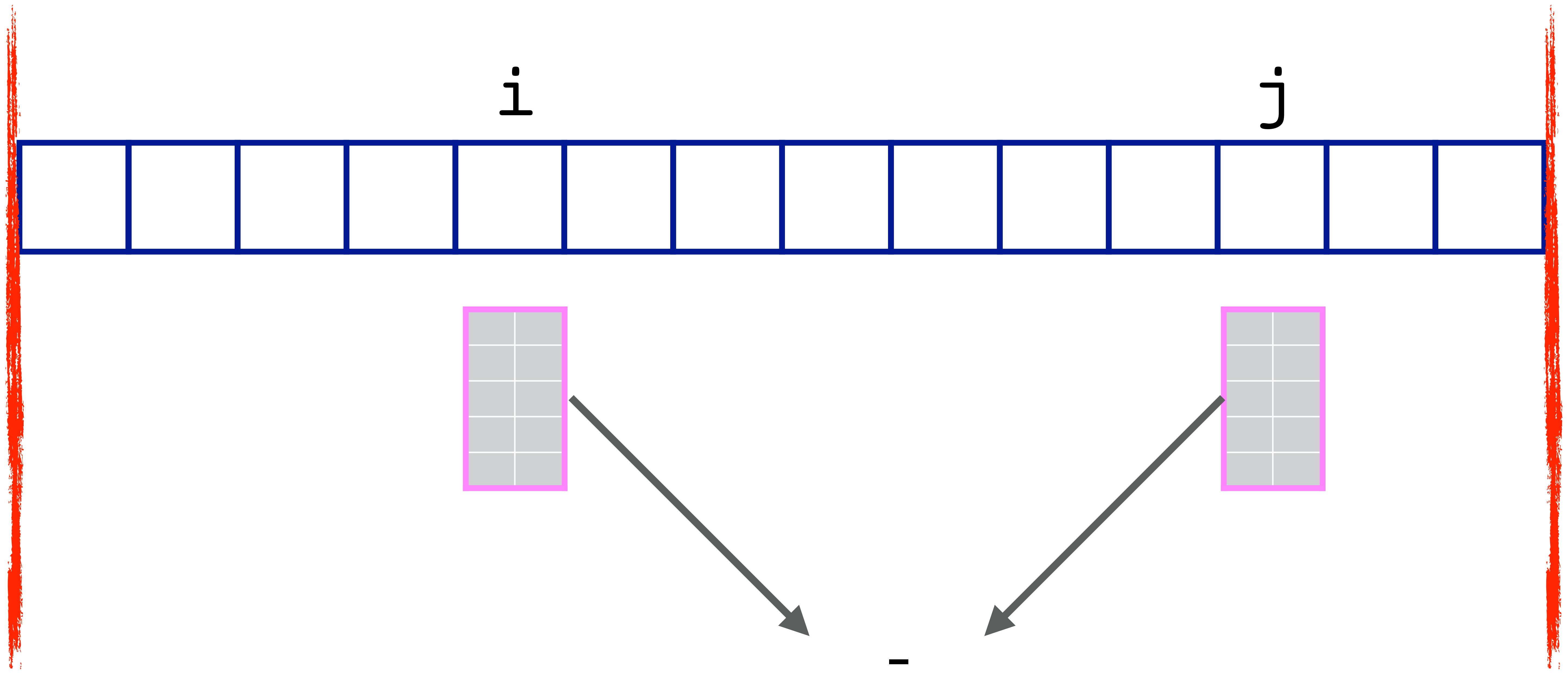
- Saves space at expense of additional table access
- Breaks each frame to $\sqrt{n}$ sized segments
- At end of each segments, we keep level-1 table that counts item frequencies from the beginning of the frame
- level-0 tables computes frequency within a segment for each block



x d x y b,d x,d x x d b x c
ACC_1
ACC_2
$Table_0$
$Table_1$
$\sqrt{n} Table_0$

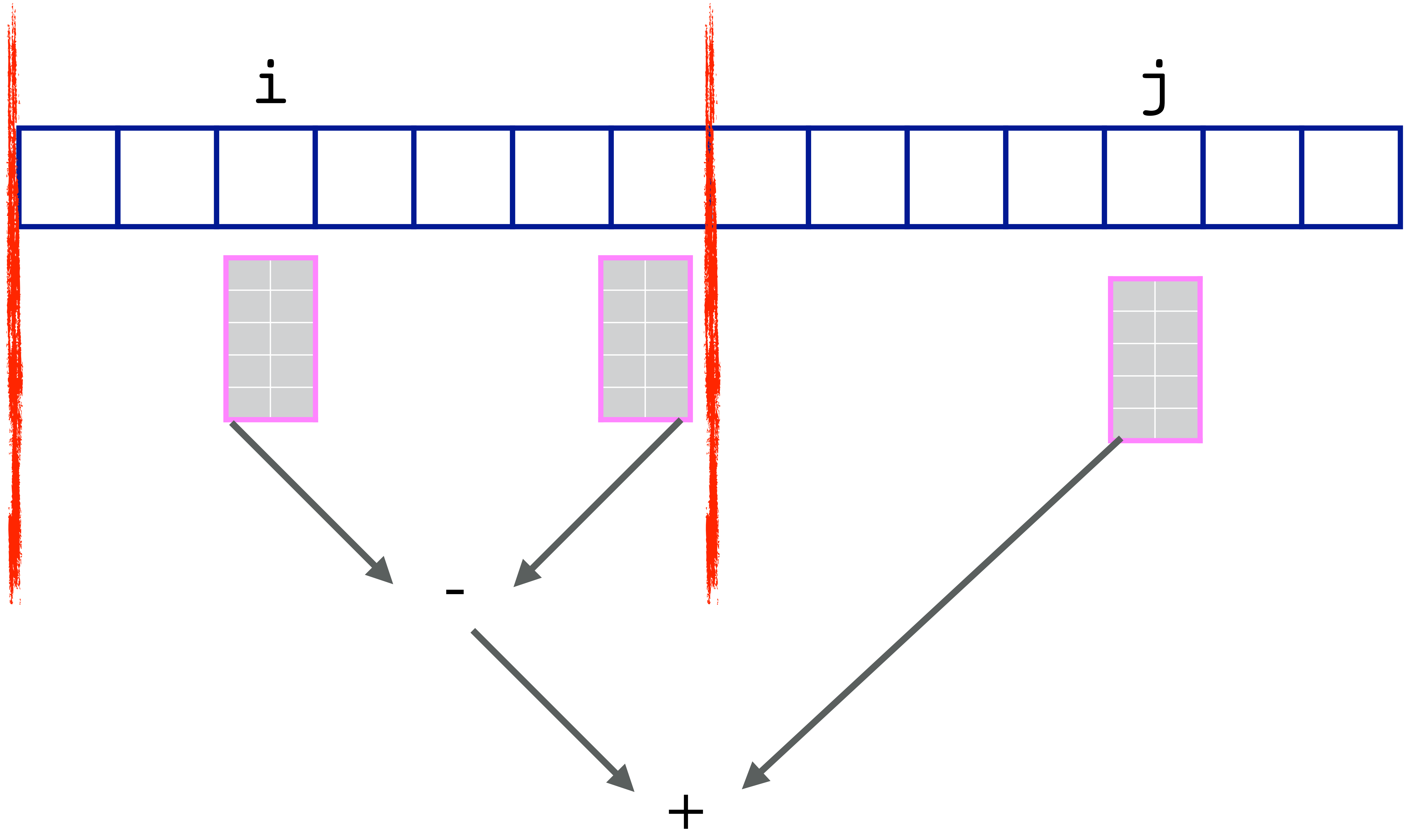
Answering Interval Frequency Query(ACC1)

- For [i,j], let block_i, block_j be the relevant blocks
  - If block_i and block_j are in the same frame:



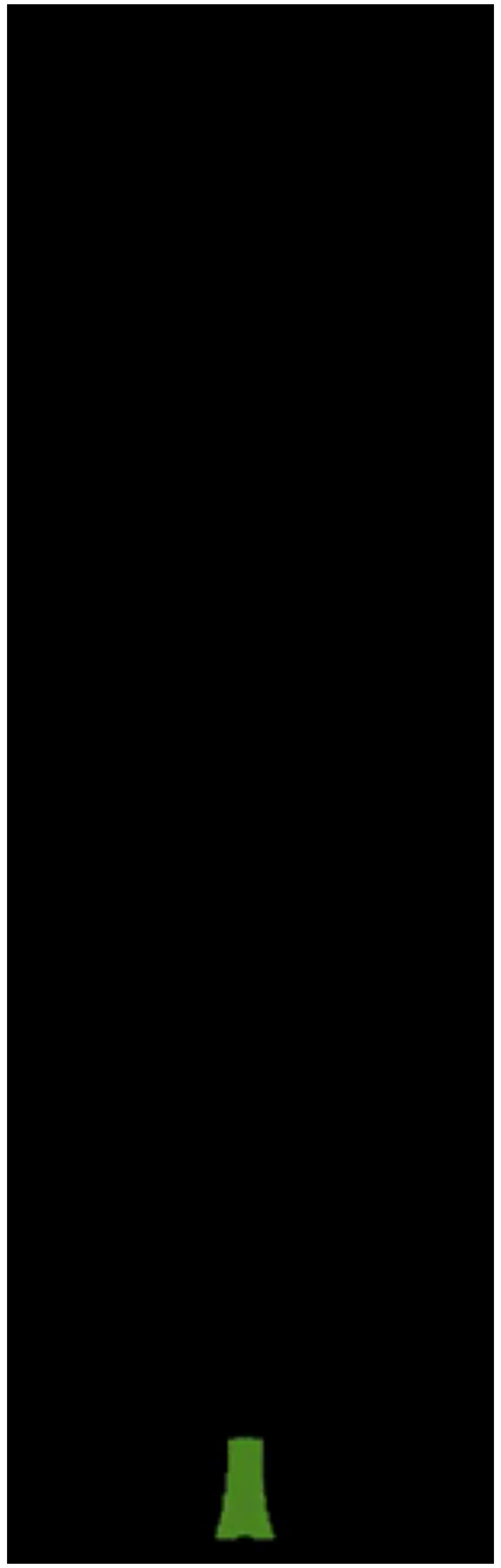
Answering Interval Frequency Query(ACC1)

- If block_`i` and block_`j` are NOT in the same frame:



Hit Algorithm

Hierarchical Interval Tree



HIT Algorithm

- Uses hierarchical tree structure
- Each Node stores partial frequency of its sub-tree
- $level_0$ tracks how many times each item arrived within block
- $level_l$ of $block_i$ tracks how many items arrived between $[block_{i-2^l+1}, block_i], 0 < l \leq trailing\_zeros(i)$

HIT Algorithm

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

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

Answering Interval Frequency Query

- For [`i,j`], let block_`i`, block_`j` be the relevant blocks
    - Scan backward from block_j to block_i, greedily using the highest possible level at each point.
    - If block_`j`> block_`i` all tables are valid
    - Otherwise, use level_0 between block_0 to block_j and compute block_`i` to block_`n` as before

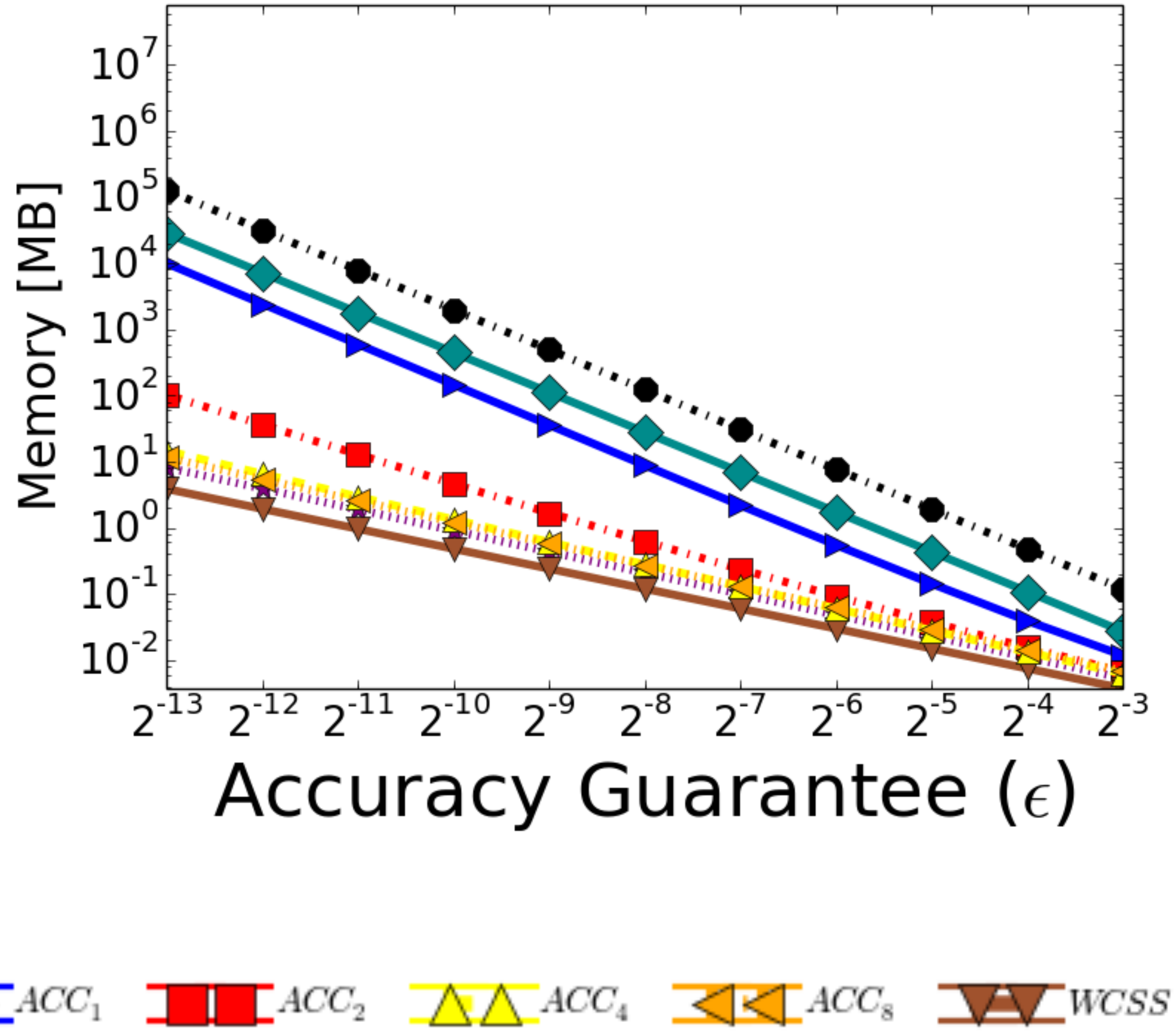
-

Evaluations

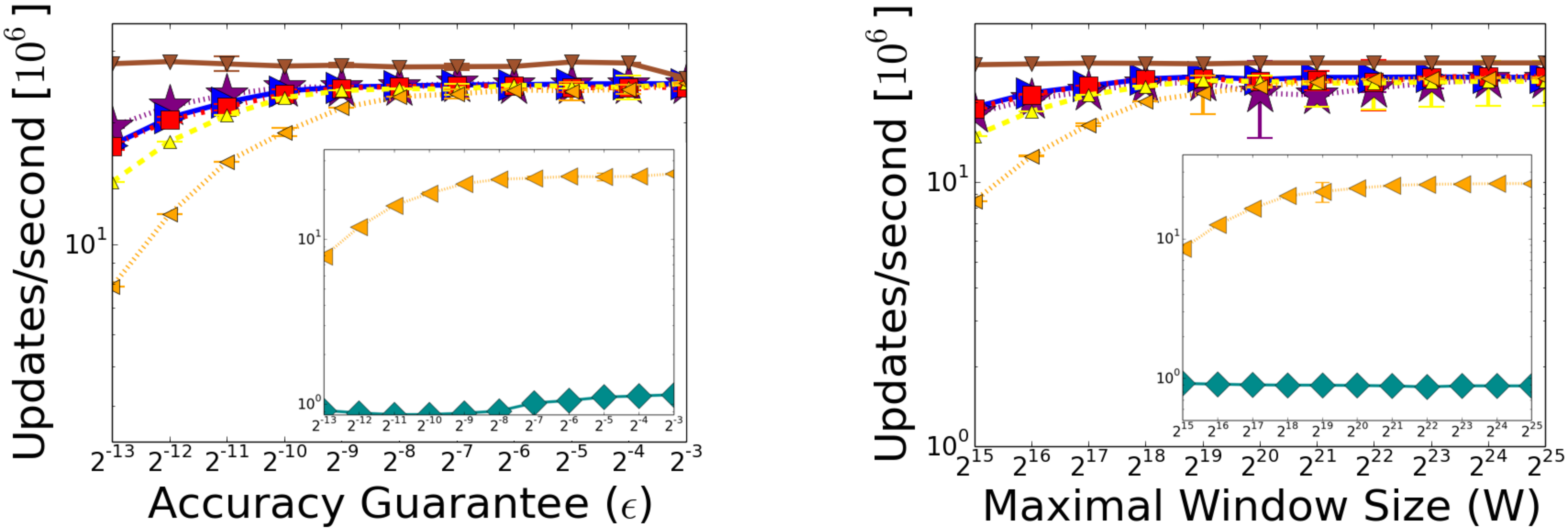
Setup

- C++ implementation
- Backbone dataset

Memory Consumption

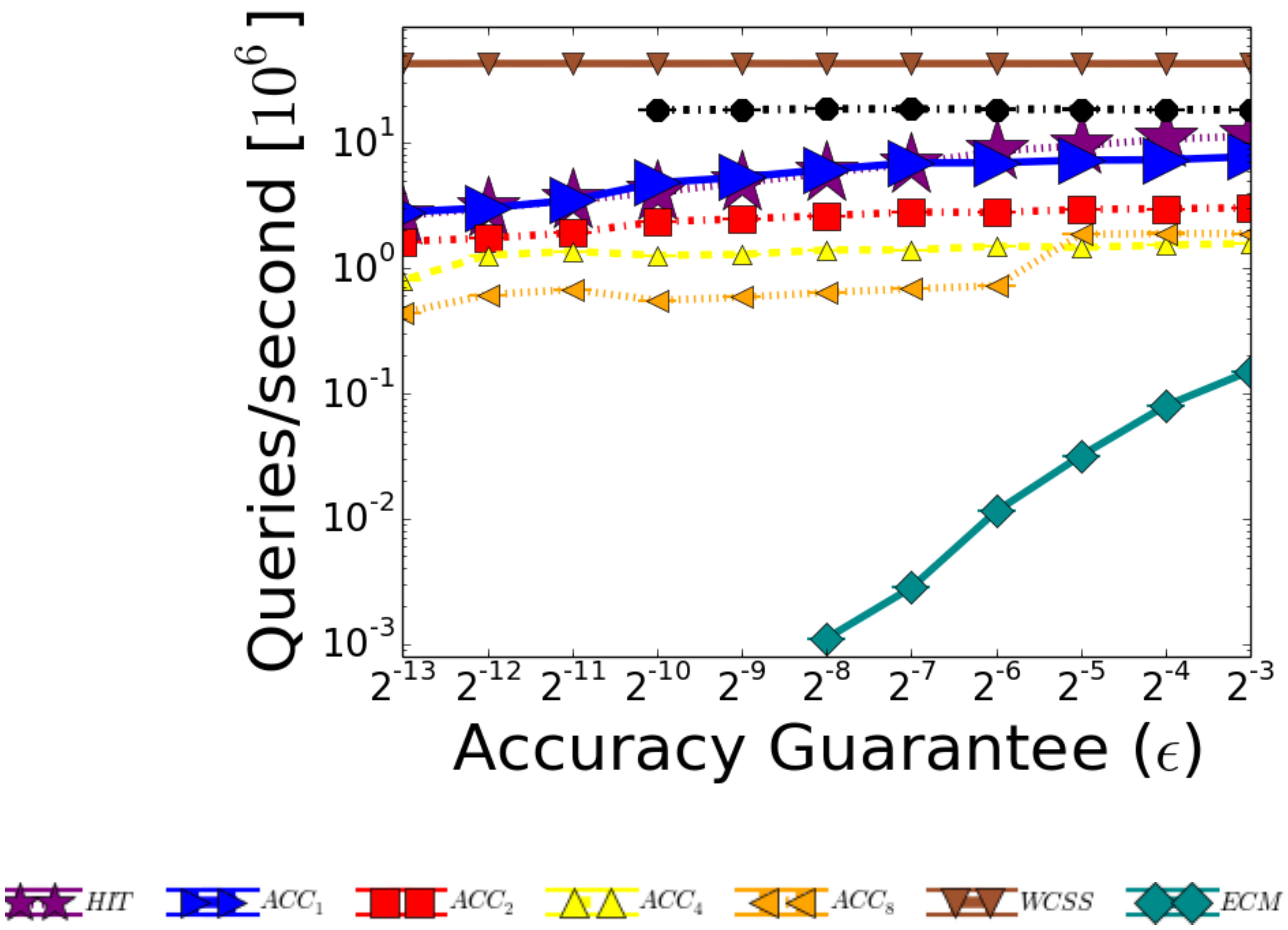


Update Speed Comparison



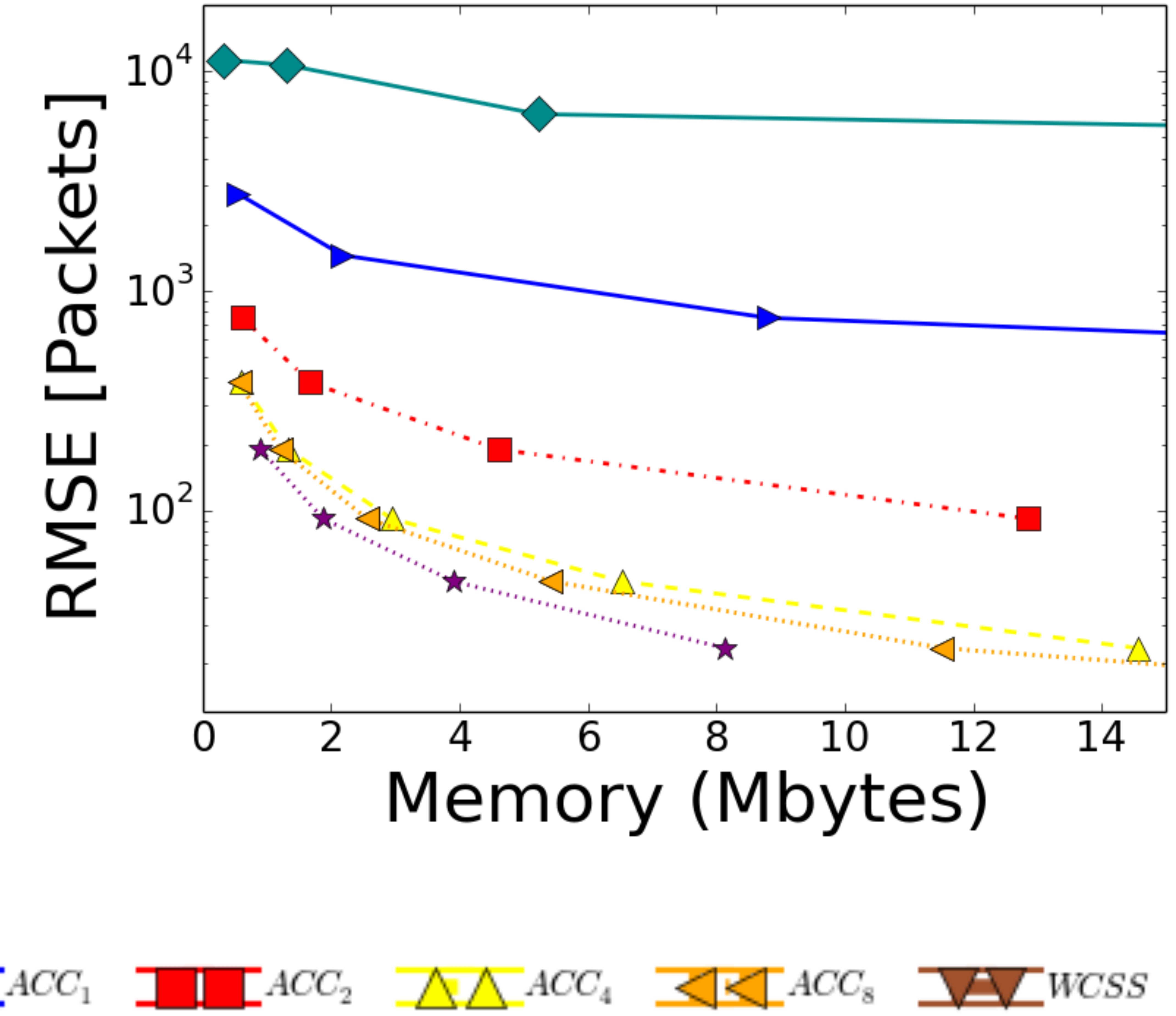
W= 2^20

Query Speed Comparison

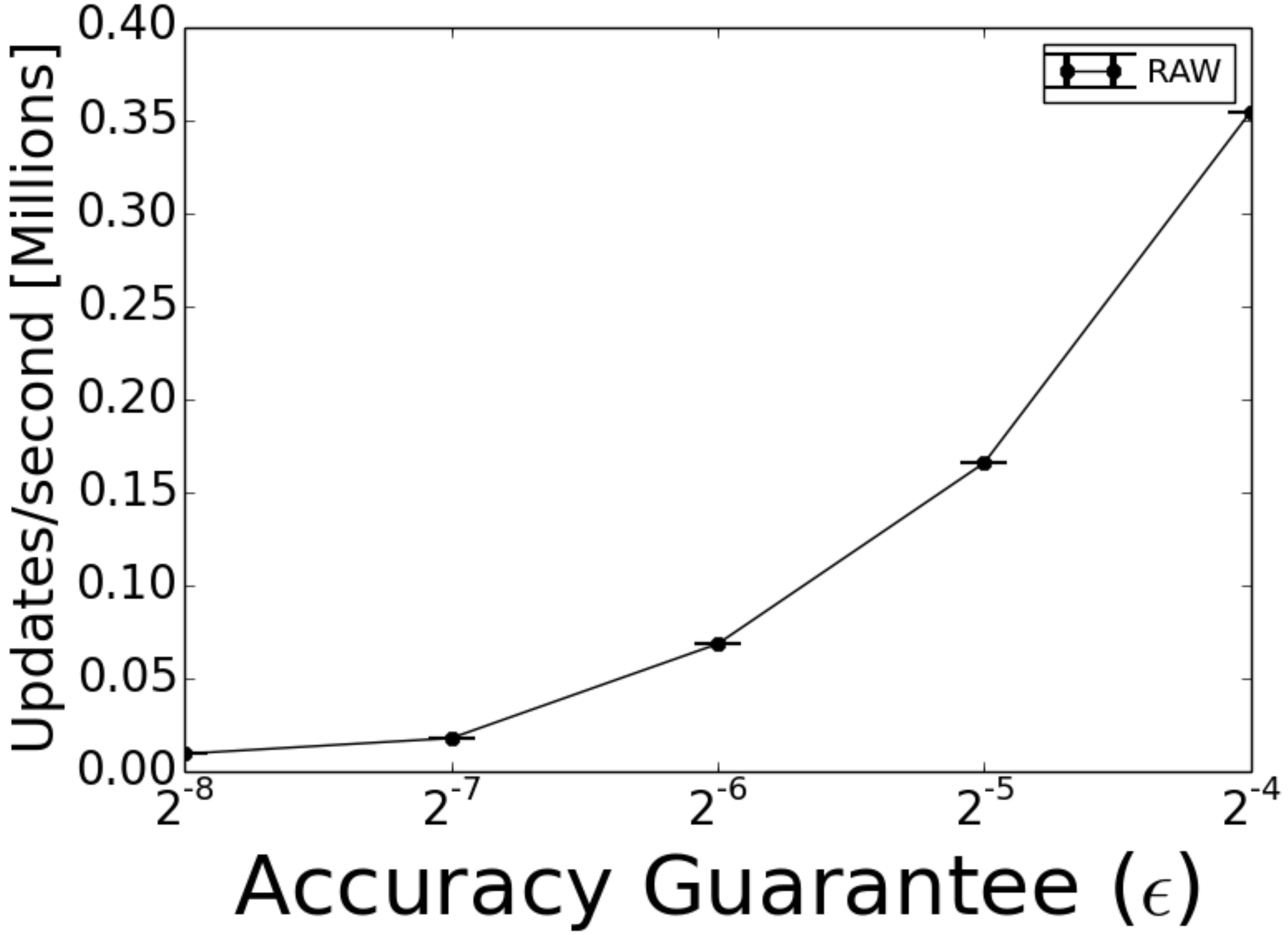


intervals = 1%xW

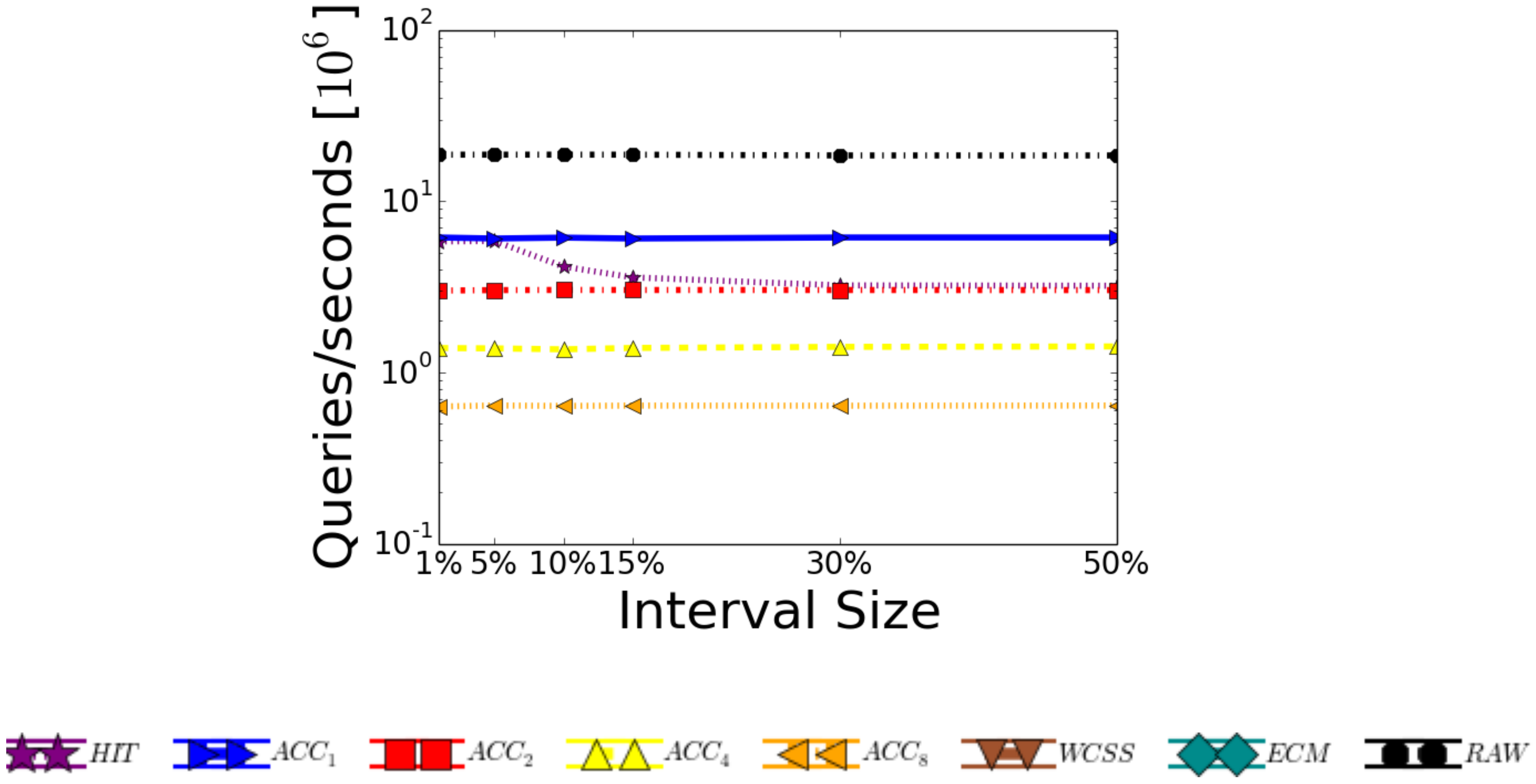
Observed Error



Update Speed Comparison

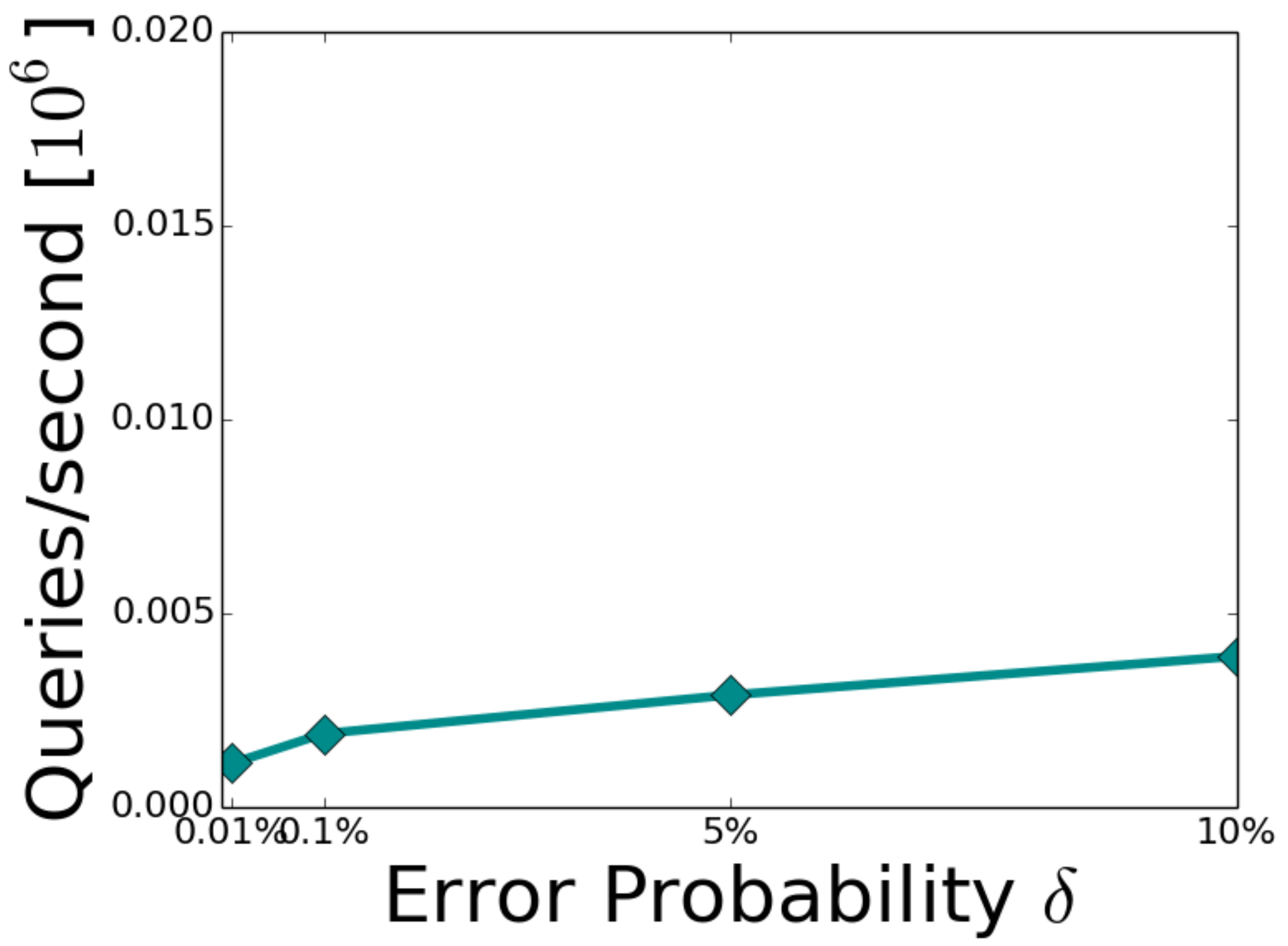


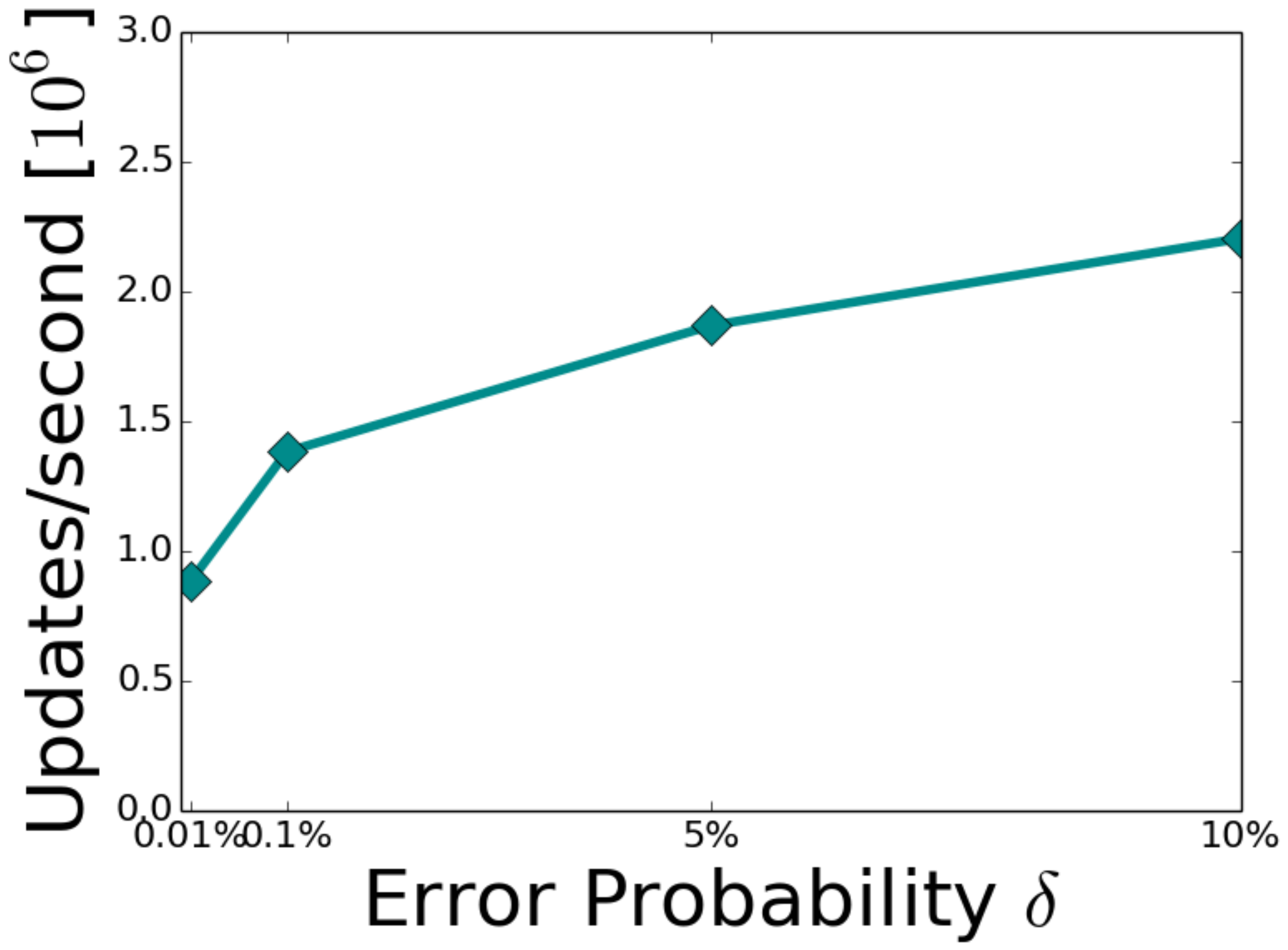
Variable Interval Sizes

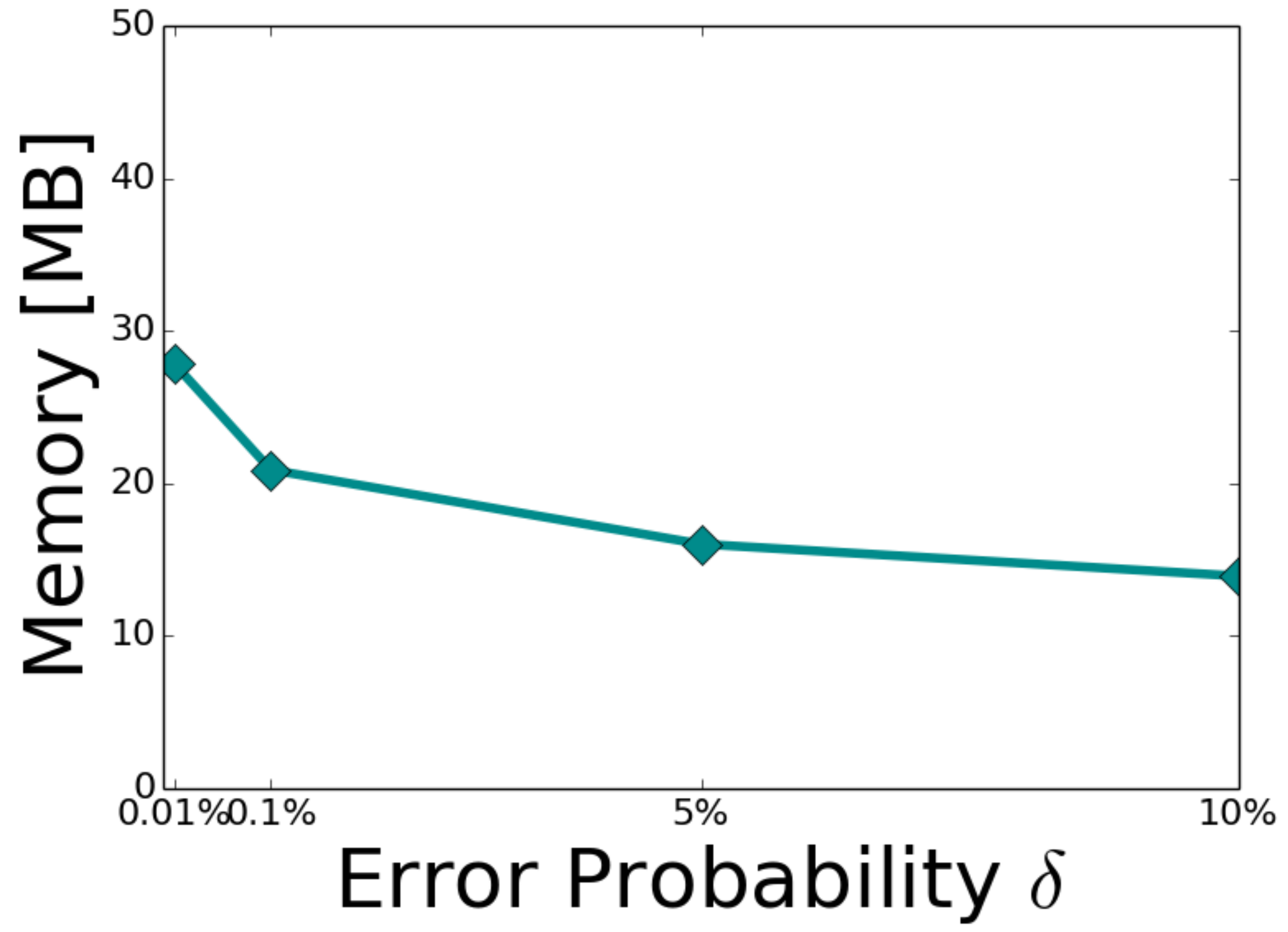


Thank YOU!

ECM Space and Performance Comparison







Algorithm	Space	Update Time	Query Time	Comments
WCSS [8]	$O(\epsilon^{-1}\log(W\|\mathcal{U}\|))$	$O(1)$	$O(1)$	Only supports fixed-size window queries.
ECM [33]	$O(\epsilon^{-2}\log W\log\delta^{-1})$	$O(\log\delta^{-1})$	$O(\epsilon^{-1}\log W\log\delta^{-1})$	Only provides probabilistic guarantees.
RAW	$O(\epsilon^{-2}\log(W\|\mathcal{U}\|))$	$O(\epsilon^{-1})$	$O(1)$	Uses prior art (WCSS) as a black box.
ACC_k	$O\left(\epsilon^{-1}\log(W\|\mathcal{U}\|) + k\epsilon^{-(1+1/k)}\log\epsilon^{-1}\right)$	$O(k+\epsilon^{-2}/W)$	$O(k)$	Constant time operations for $k=O(1)\wedge\epsilon=\Omega(W^{-1/2})$.
HIT	$O(\epsilon^{-1}(\log(W\|\mathcal{U}\|)+\log^2\epsilon^{-1}))$	$O\left(1+\left(\epsilon^{-1}\cdot\log\epsilon^{-1}\right)/W\right)$	$O(\log\epsilon^{-1})$	Optimal space when $\log^2\epsilon^{-1}=O(\log(W\|\mathcal{U}\|))$, $O(1)$ time updates when $\epsilon=\Omega\left(\frac{\log W}{W}\right)$.