Low-latency analytics on colossal data streams with

SummaryStore

Nitin Agrawal, Ashish Vulimiri

Samsung Research

This paper in 30 seconds!

SummaryStore: approximate store for “colossal” time-series data

Key observation: in time-series analyses

- Newer data is typically more important than older
- Can get away with approximating older data more

In real applications (forecasting, outlier analysis, …) and microbenchmarks:

scale	**1 PB** on commodity node (compacted **100x**)
latency	**< 1s** at 95th %ile
error	**< 10%** at 95th %ile

Low-latency analytics on colossal data streams with

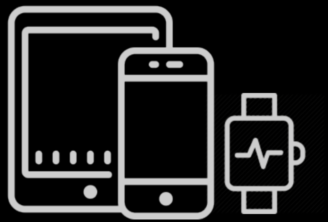
SummaryStore

Nitin Agrawal, **Ashish Vulimiri**

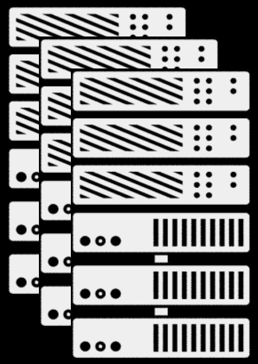
Samsung Research

“Colossal” streaming data

4 TB /car /day
x 100s thousands cars



10 MB /device /day
x millions devices



10 TB /data center /day
x 10s data centers

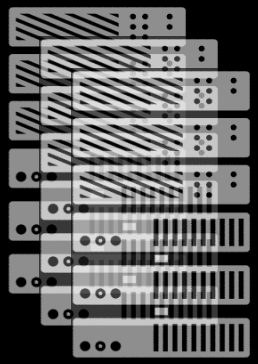
20 GB /home /day
x 100s thousands homes

“Colossal” streaming data

4 TB / car / day
x 100s thousands cars

10 MB / device / day
x millions devices

Hundreds of **TB** to **PB** / day

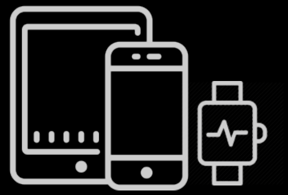


10 TB / data center / day
x 10s data centers

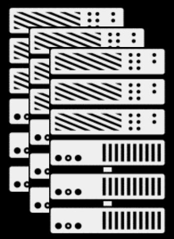
20 GB / home / day
x 100s thousands homes

Stream analytics on “colossal” data

Need to support timely analytics



- Forecast [Facebook Prophet]
- Recommend [SoundCloud, Spotify]
- Detect outliers [Etsy Kale]
- Telemetry [Splunk, Twitter Observability]

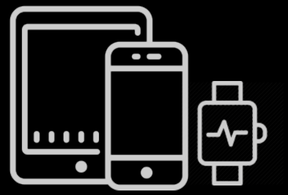


Stream analytics on “colossal” data

Current solutions

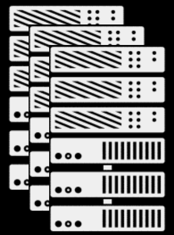
In-memory analytics systems

- Interactive latency, but $$$$
- Need secondary system for persistence



Conventional time-series stores

- High latency, still quite expensive



Goal: build a low-cost, low-latency

store for stream analytics

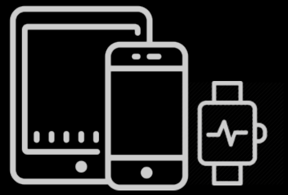
approximate

Stream analytics on “colossal” data

Current solutions

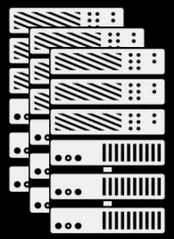
In-memory analytics systems

- Interactive latency, but $$$$
- Need secondary system for persistence



Conventional time-series stores

- High latency, still quite expensive



Approximate data stores?

- Promising reduction in cost & latency in other domains
- Current systems are not viable for streaming data

Key insight

We make the following observation:

many stream analyses favor newer data over older
existing stores are oblivious, hence costly and slow

Examples:

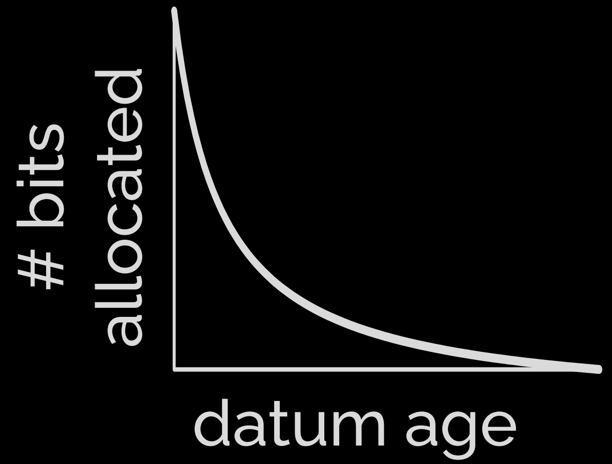
Spotify, SoundCloud	Time-decayed weights in song recommender
Facebook EdgeRank	Time-decayed weights in newsfeed recommender
Twitter Observability	Archive data past an age threshold at lower resolution
Smart-home apps	Decaying weights in e.g. HVAC control, energy monitor

...

SummaryStore: approximate store for stream analytics

Our system, **SummaryStore**

- Approximates data leveraging observation that analyses favor newer data
- Allocates fewer bits to older data than new: each datum *decays* over time

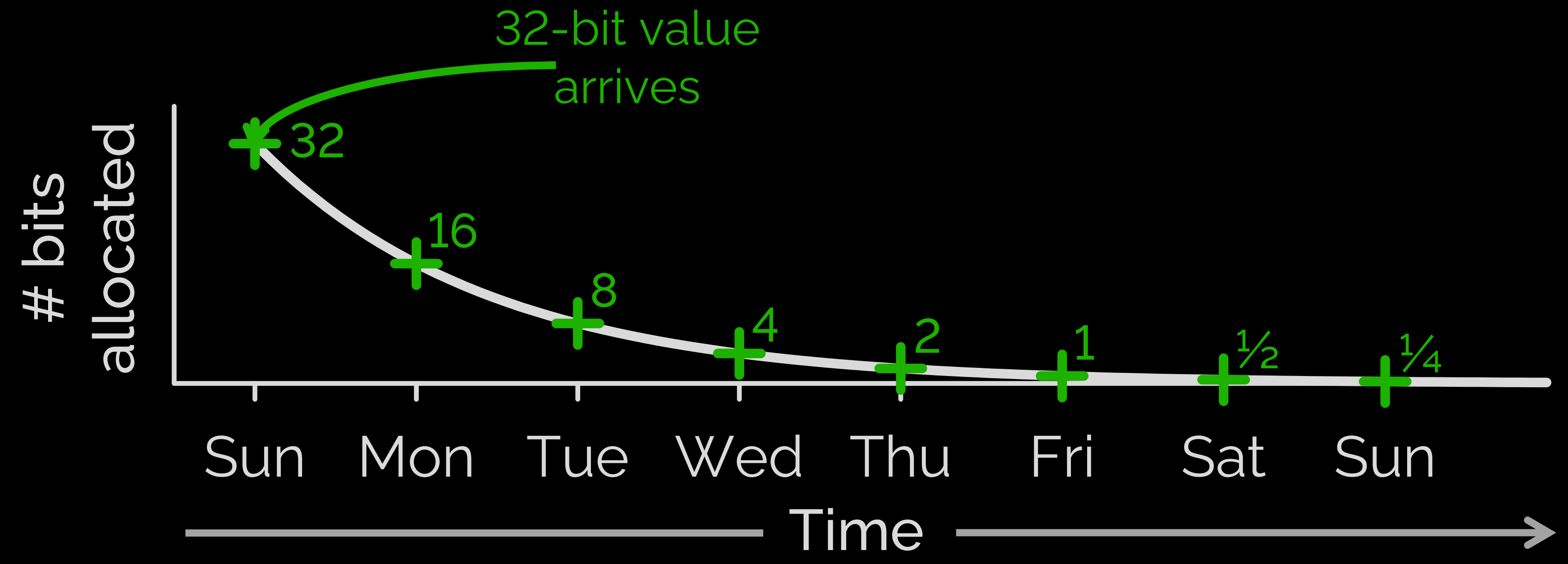


SummaryStore: approximate store for stream analytics

Our system, **SummaryStore**

- Allocates fewer bits to older data than new: each datum *decays* over time

Example decay policy: halve number of bits each day

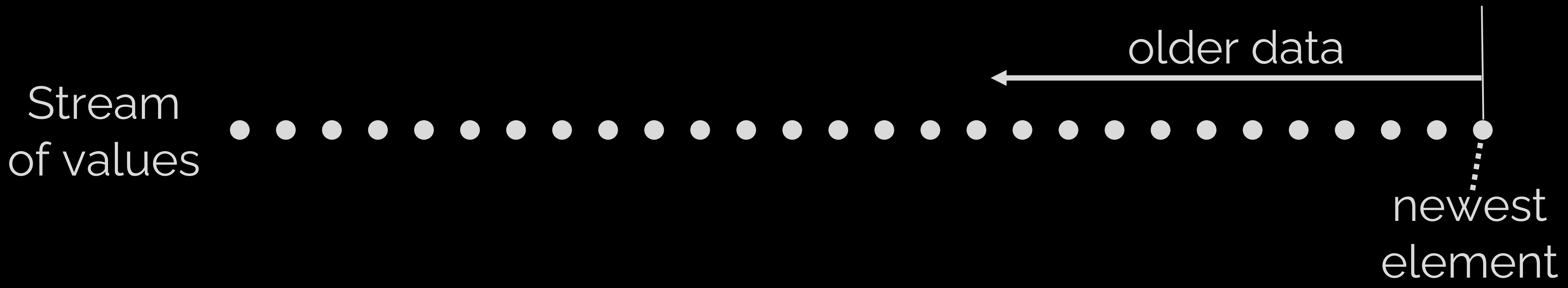


Outline

1. Time-decayed stream approximation
2. Processing writes
3. Handling queries
4. Evaluation

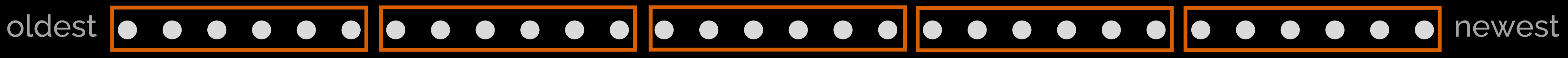
Time-decayed stream approximation

through windowed summarization



Time-decayed stream approximation

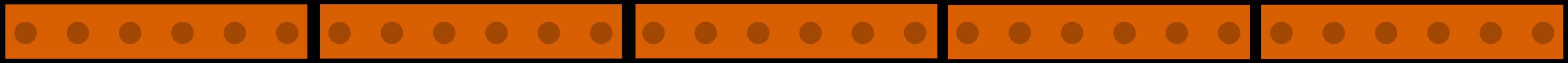
through windowed summarization



1. Group values in windows

Time-decayed stream approximation

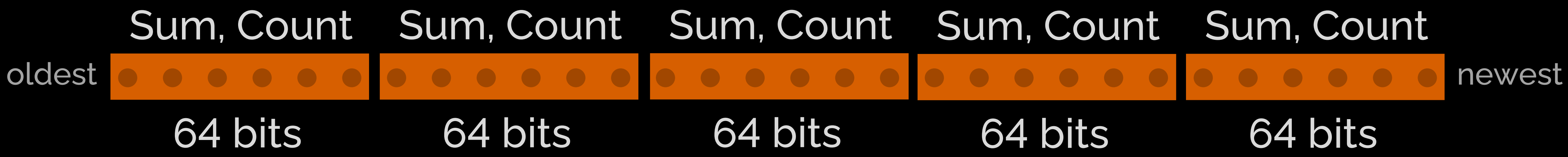
through windowed summarization

oldest  newest

1. Group values in windows. Discard raw data

Time-decayed stream approximation

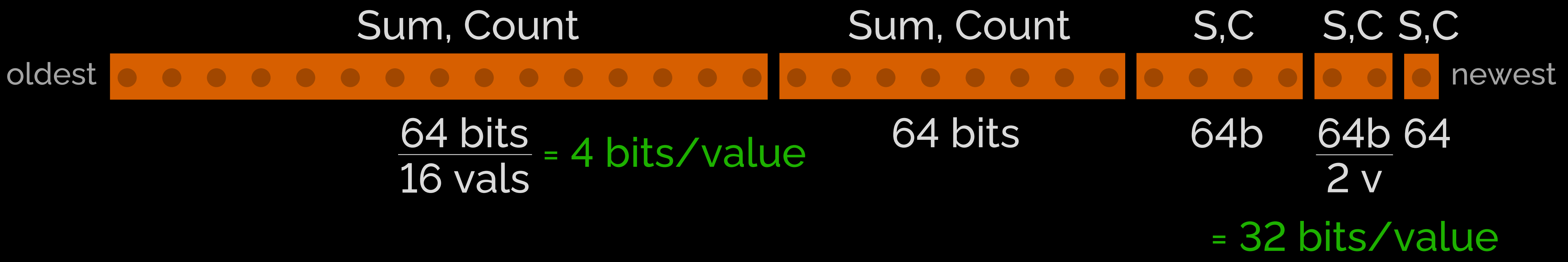
through windowed summarization



1. Group values in windows. Discard raw data, keep only window summaries
   - e.g. Sum, Count, Histogram, Bloom filter, …
   - Each window is given same storage footprint

Time-decayed stream approximation

through windowed summarization

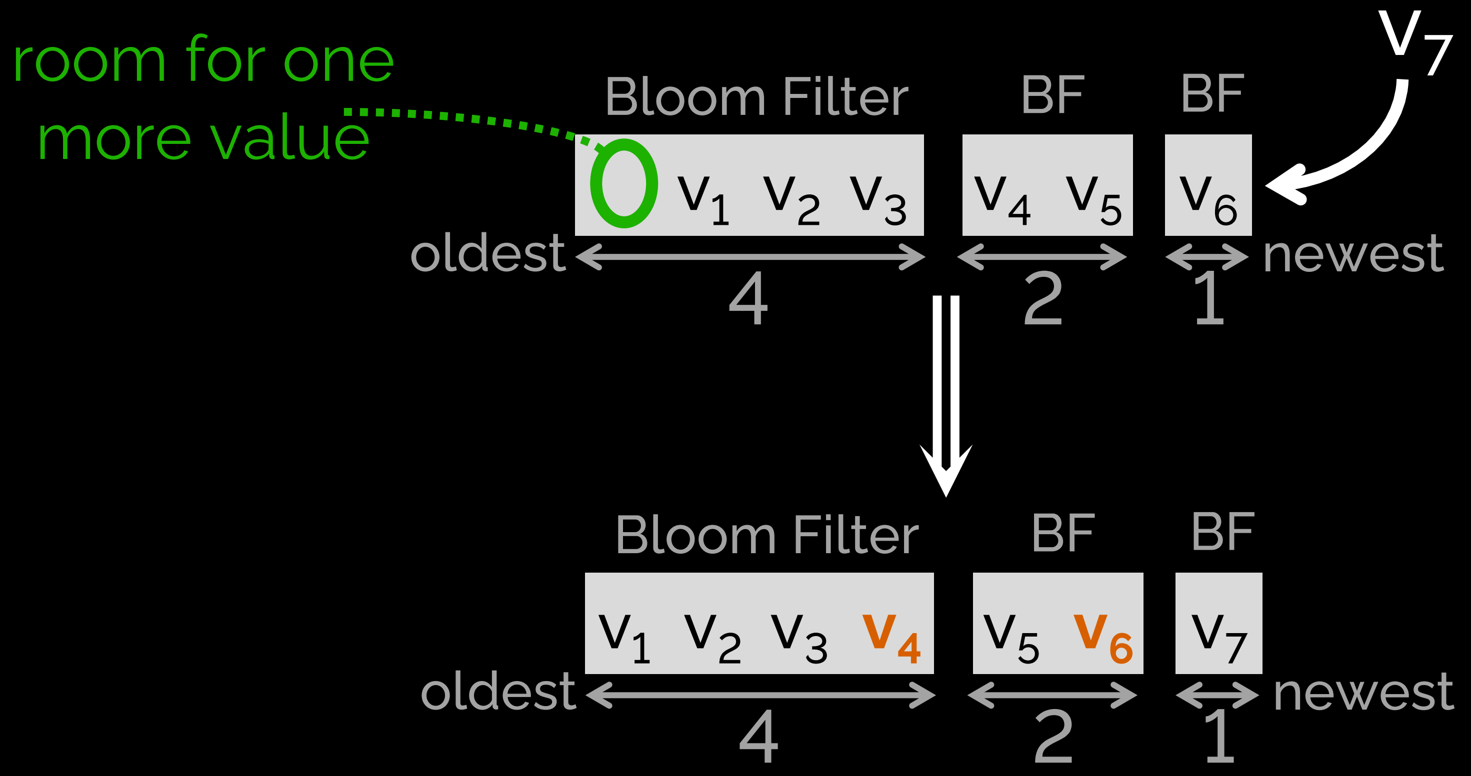


1. Group values in windows. Discard raw data, keep only window summaries
   - e.g. Sum, Count, Histogram, Bloom filter, ...
   - Each window is given same storage footprint
2. To achieve decay, use longer timespan windows over older data

Outline

1. Time-decayed stream approximation
2. Processing writes
3. Handling queries
4. Evaluation

Processing writes: Challenge



Configuration:

Window lengths 1, 2, 4, 8, ….

Each window has Bloom filter

Don't have raw values, only window summaries (Bloom filters)

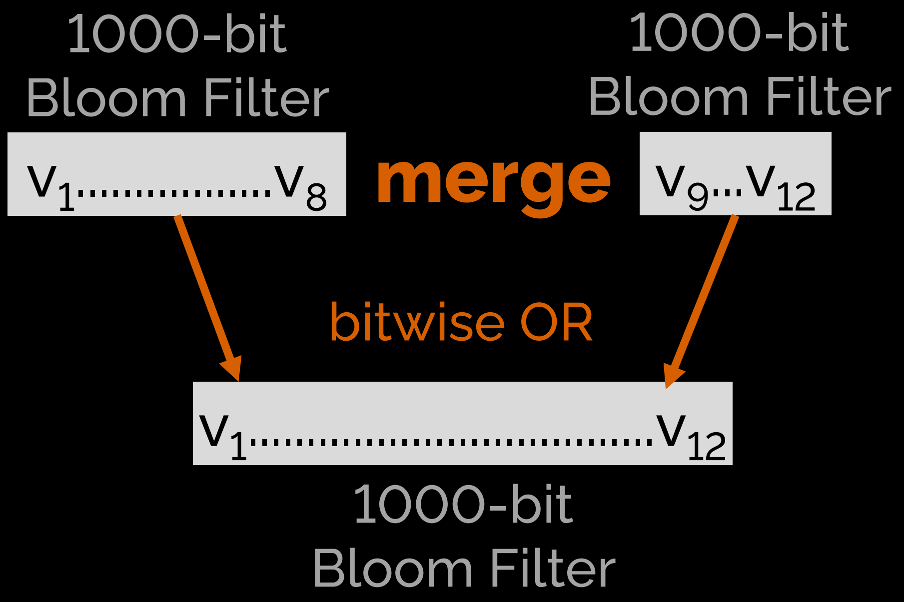
How do we "move" v_4, v_6 between windows?

Ingest algorithm

Not possible to actually move values

Instead, use a different technique, building on work by Cohen & Wang†

- Ingest new values into new windows
- Periodically compact data by **merging** consecutive windows
  - Merge all summary data structures



merge operation for

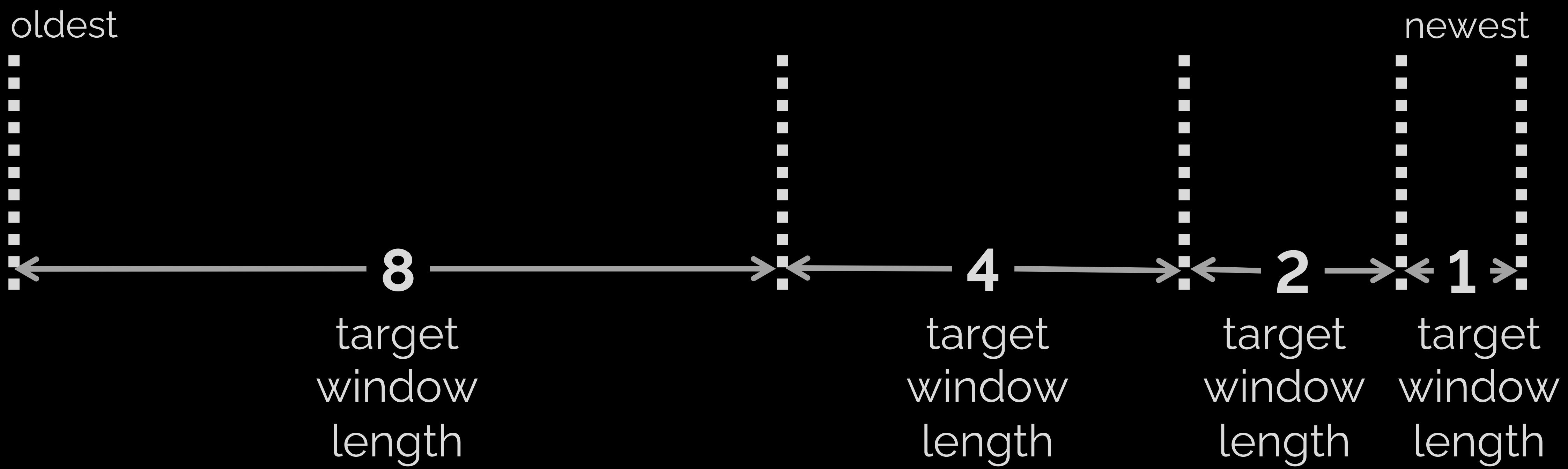
Bloom Filter	:	bitwise OR
Count	:	add
Histogram	:	combine & rebin
		etc

† E. Cohen, J. Wang, "Maintaining time-decaying stream aggregates", J. Alg. 2006

Ingest algorithm

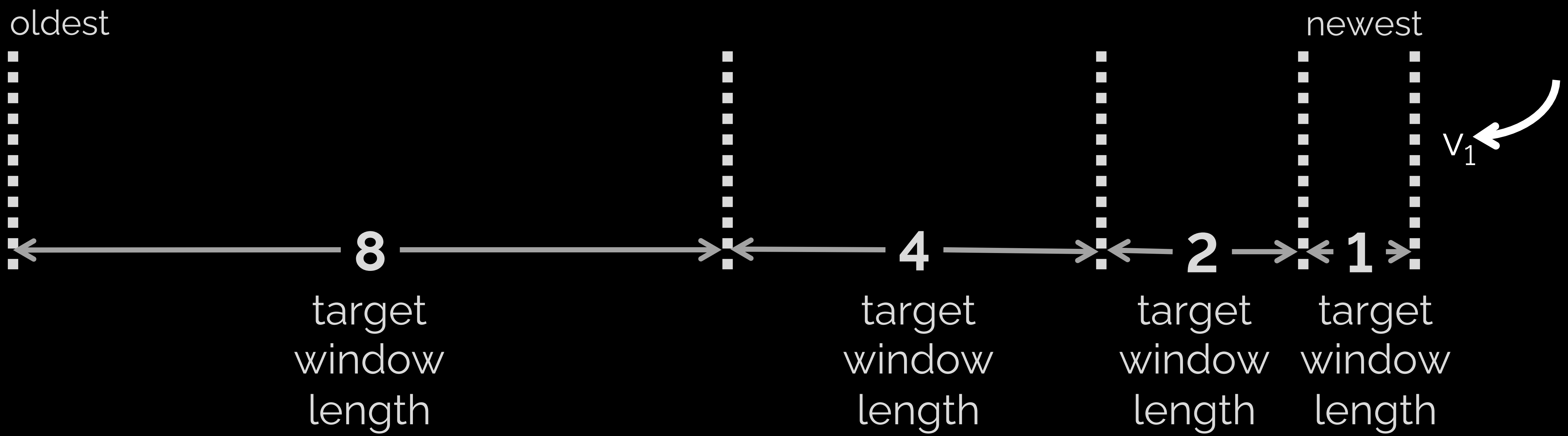
Configure a target sequence of window lengths (choice affects decay)

- Example below uses 1, 2, 4, 8, 16, ...



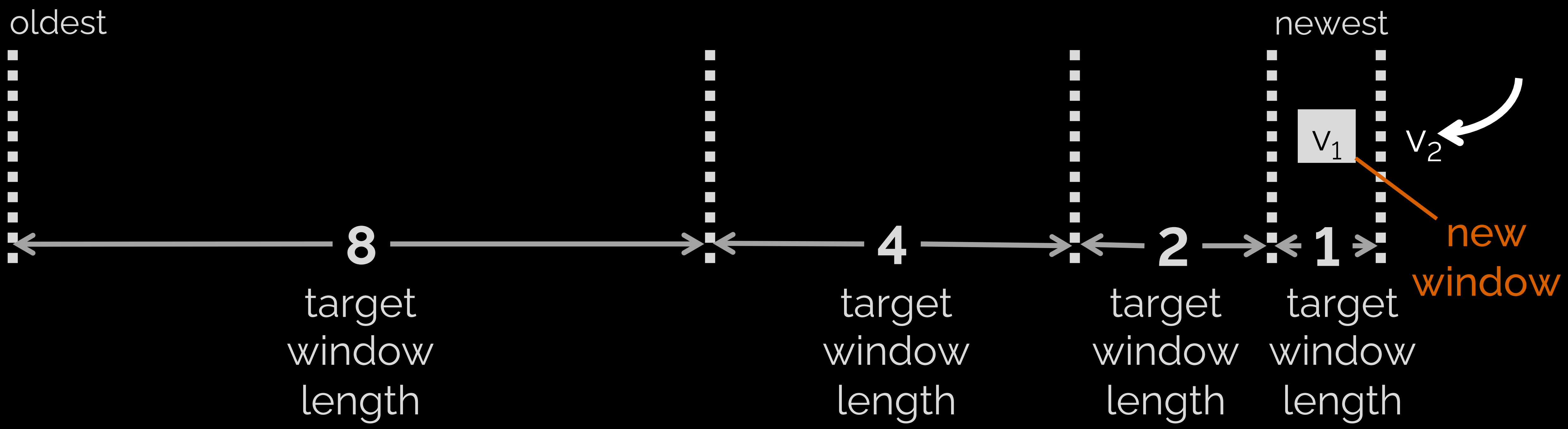
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



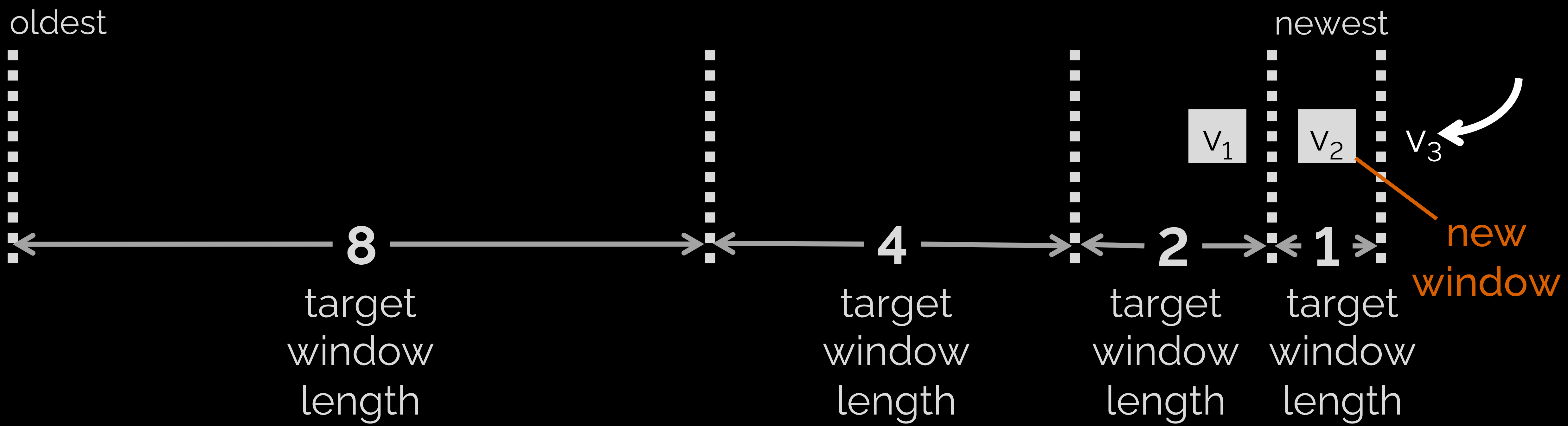
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



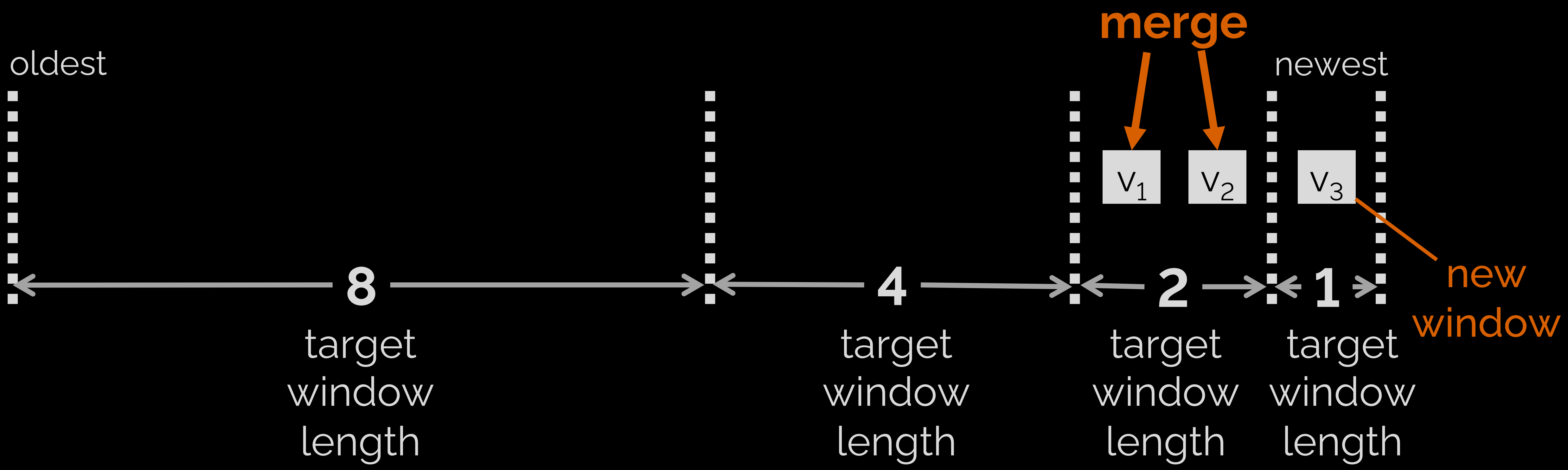
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



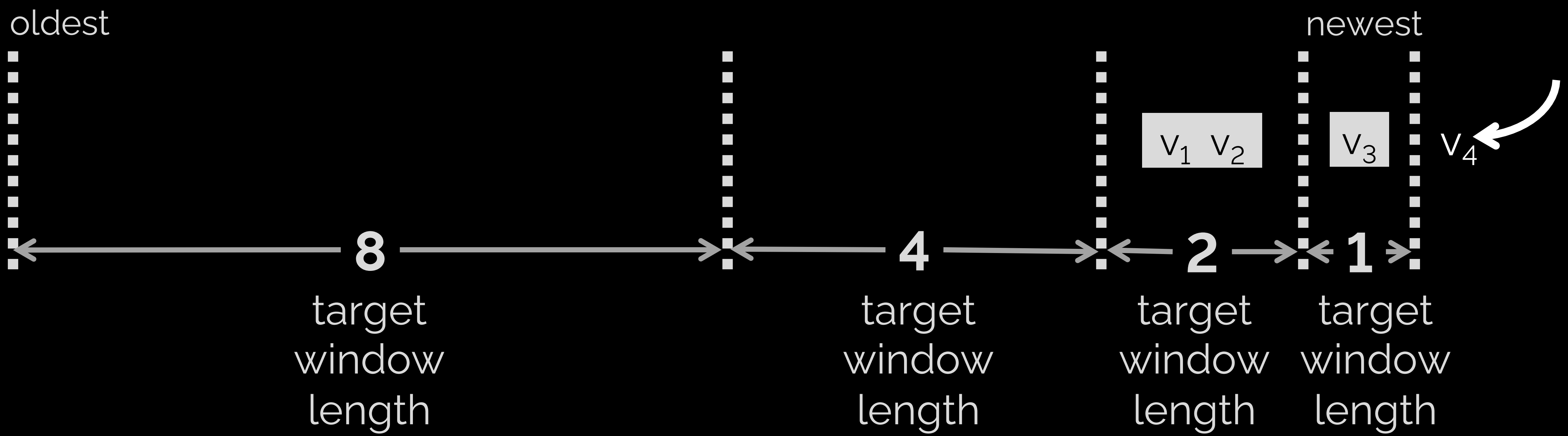
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



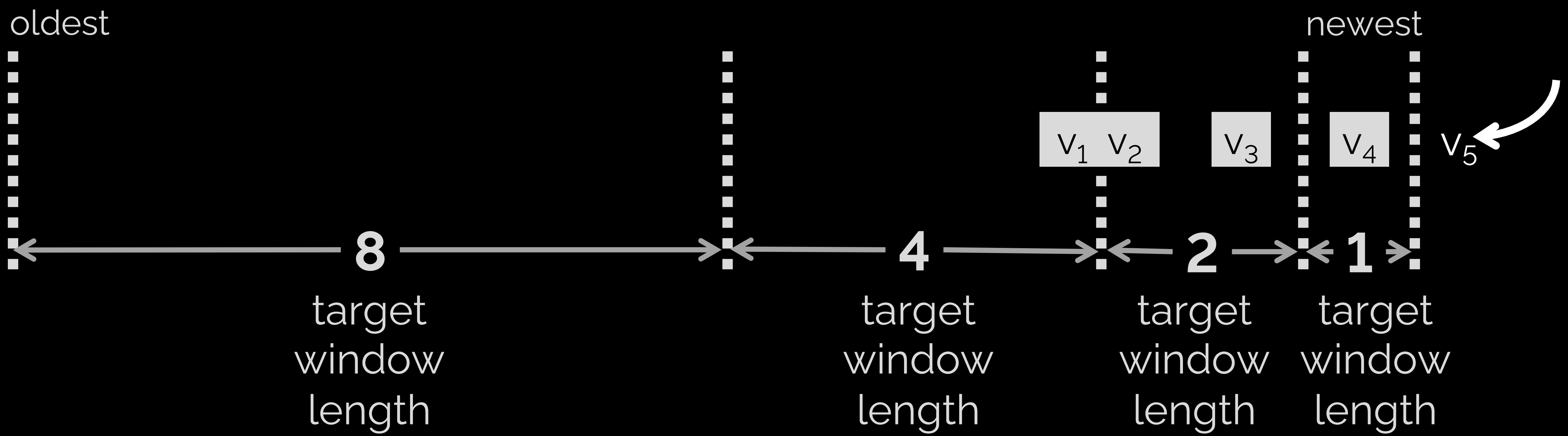
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



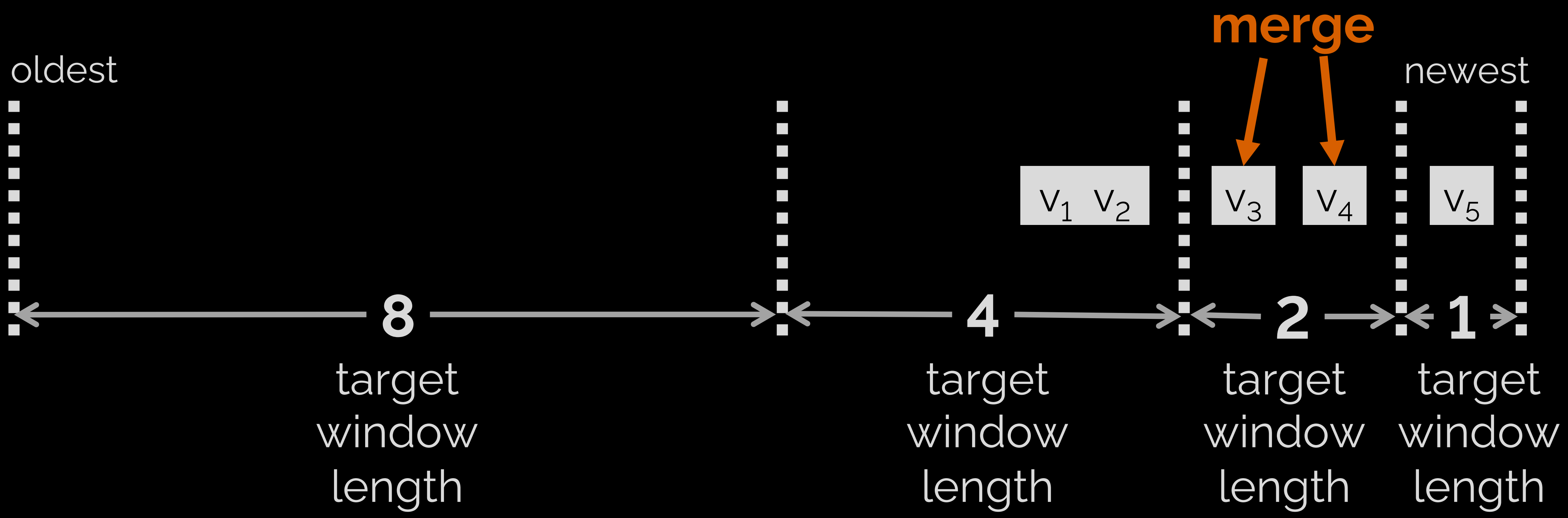
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



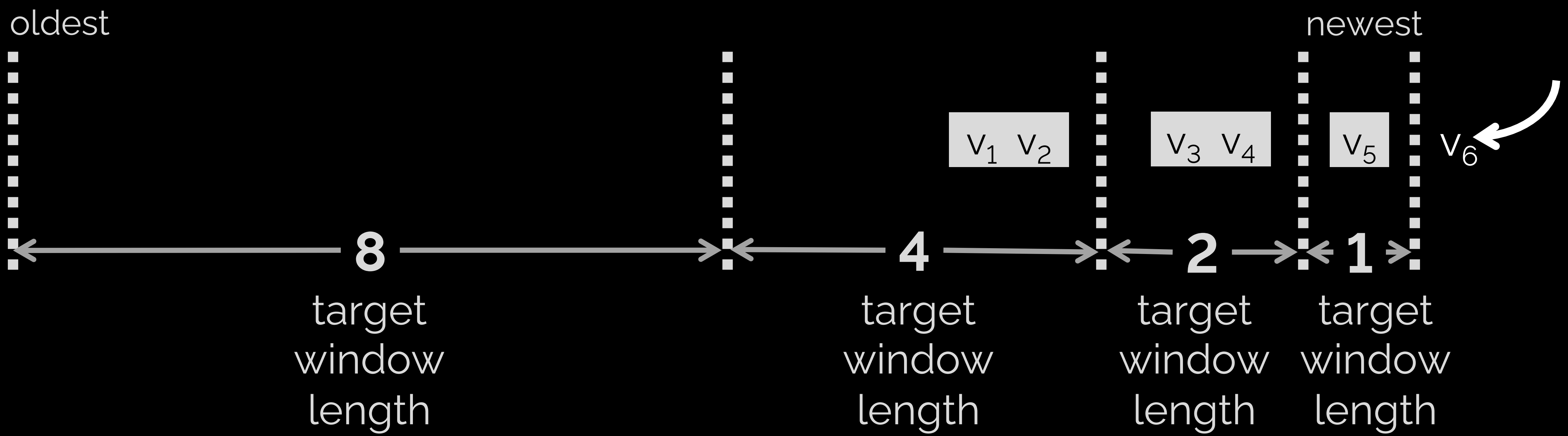
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



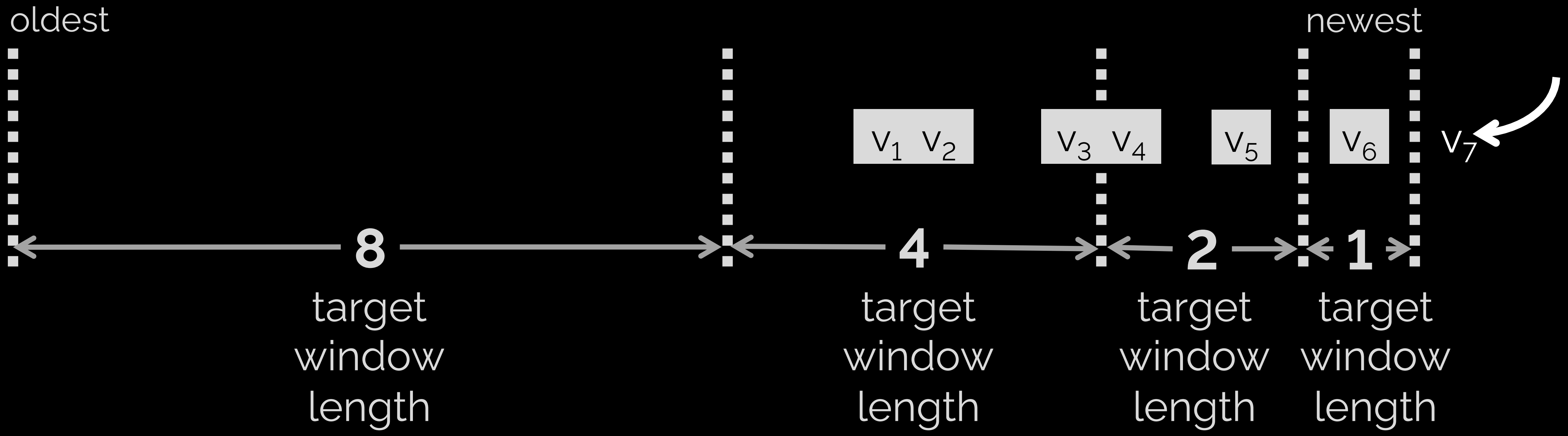
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



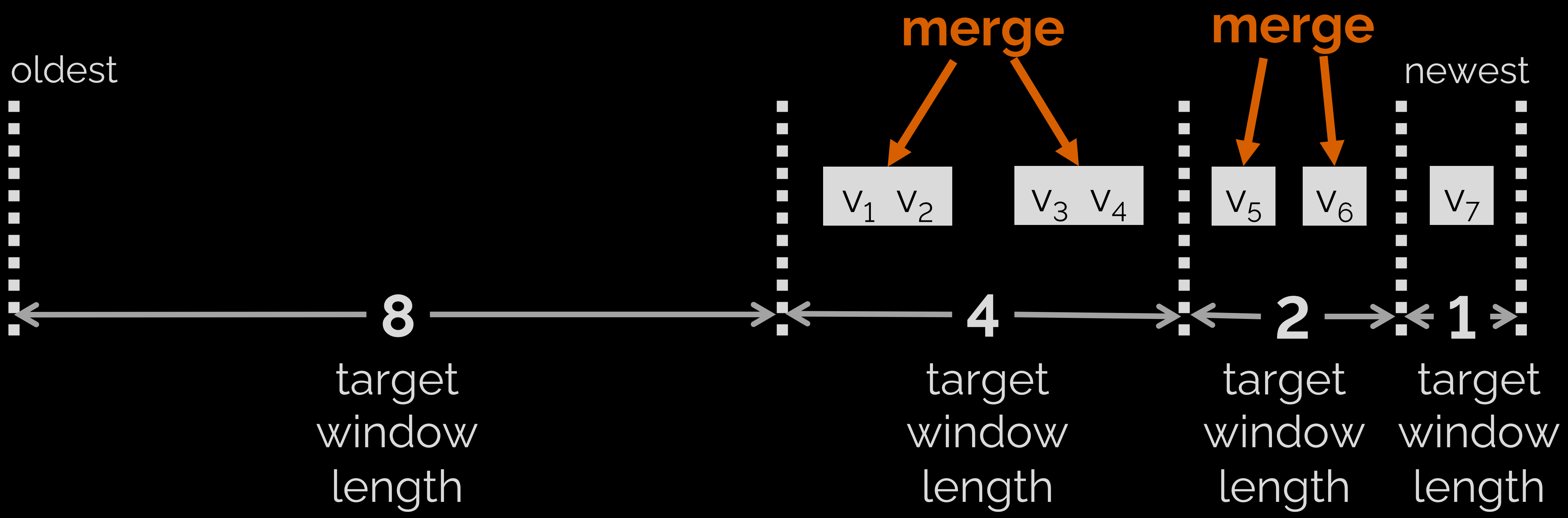
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



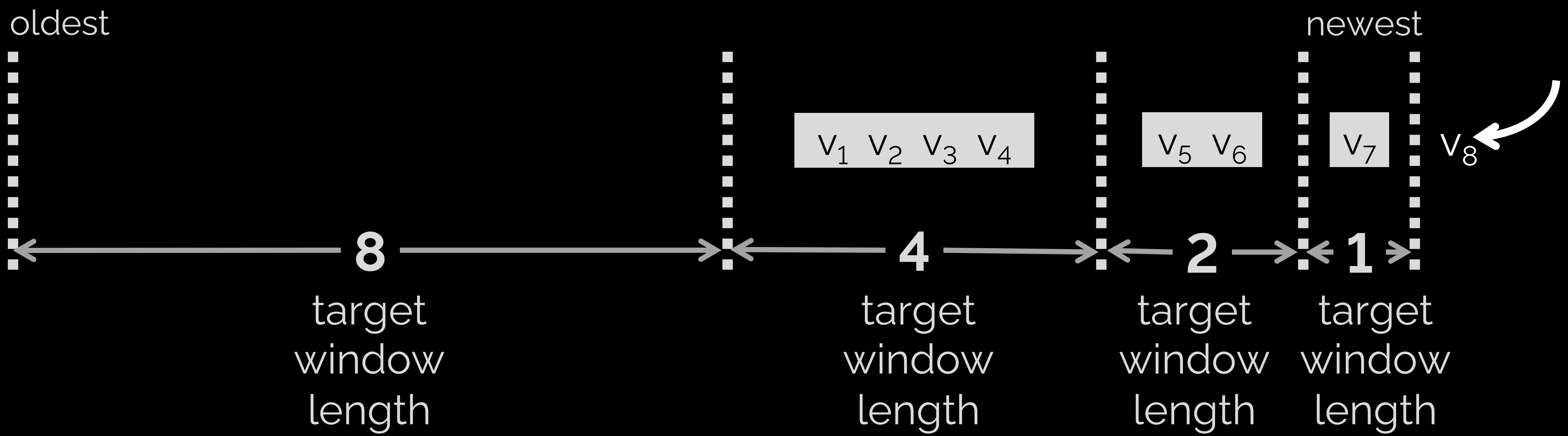
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



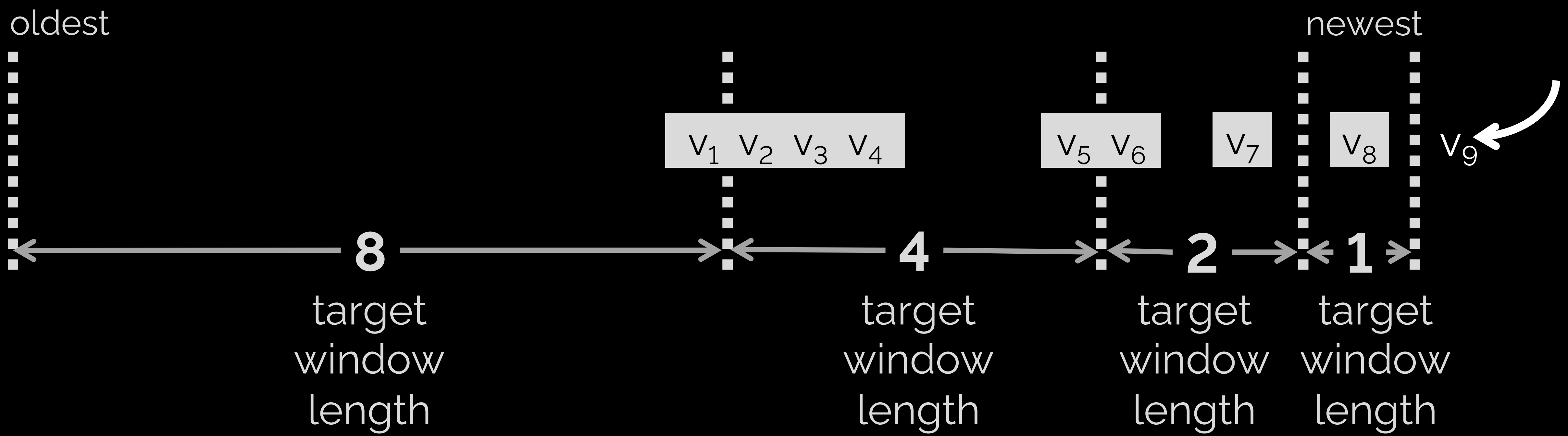
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



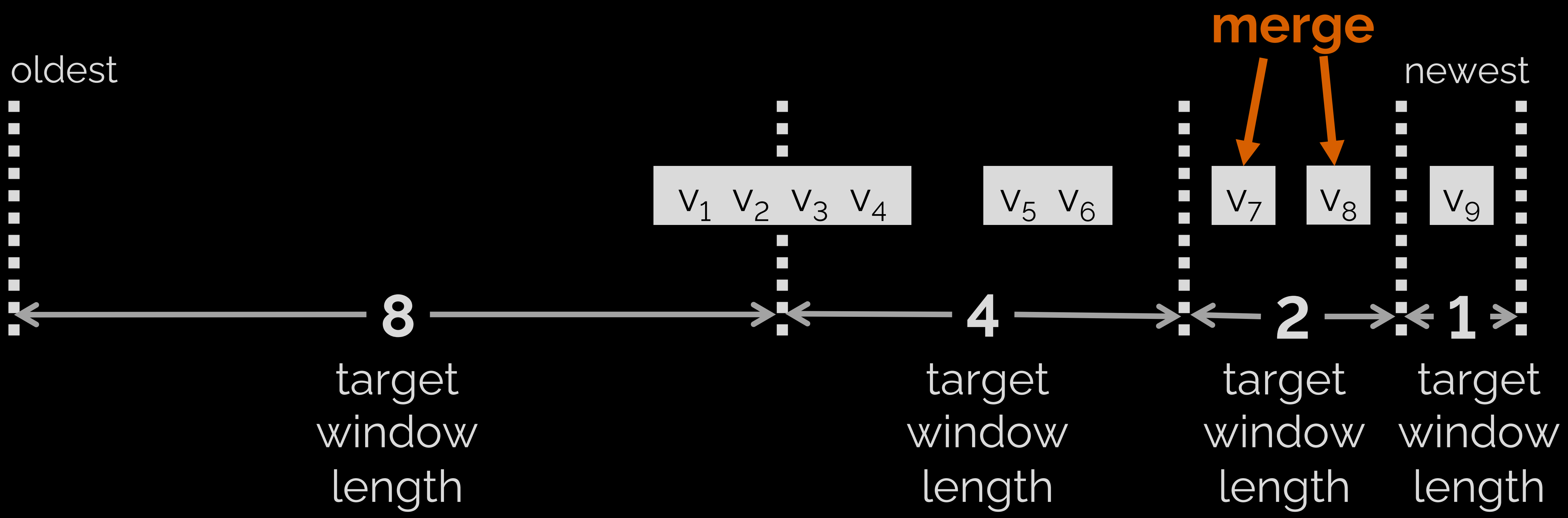
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



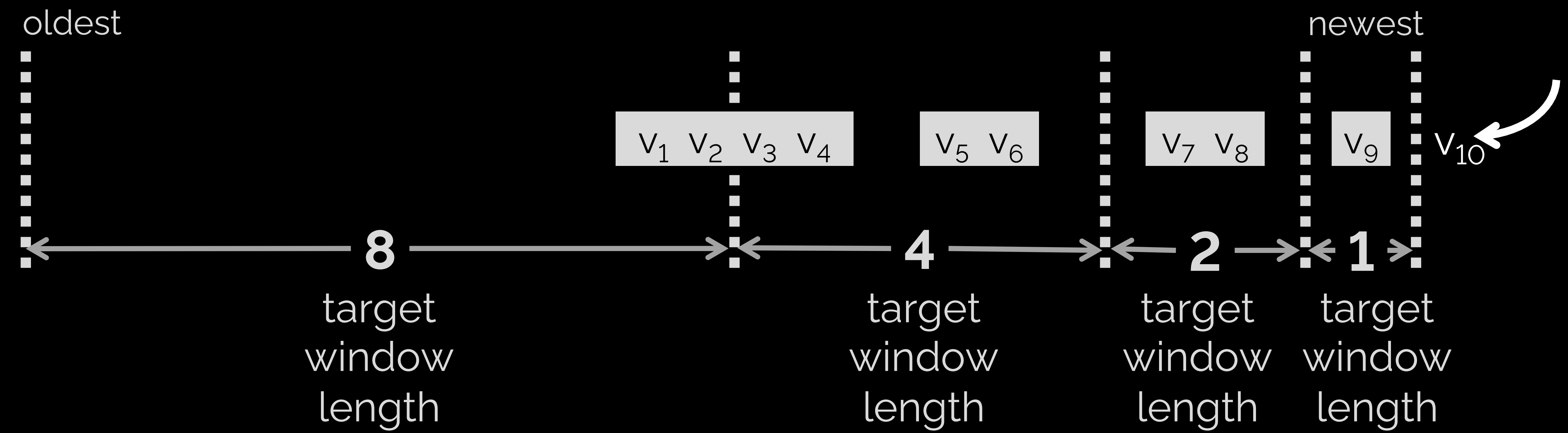
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



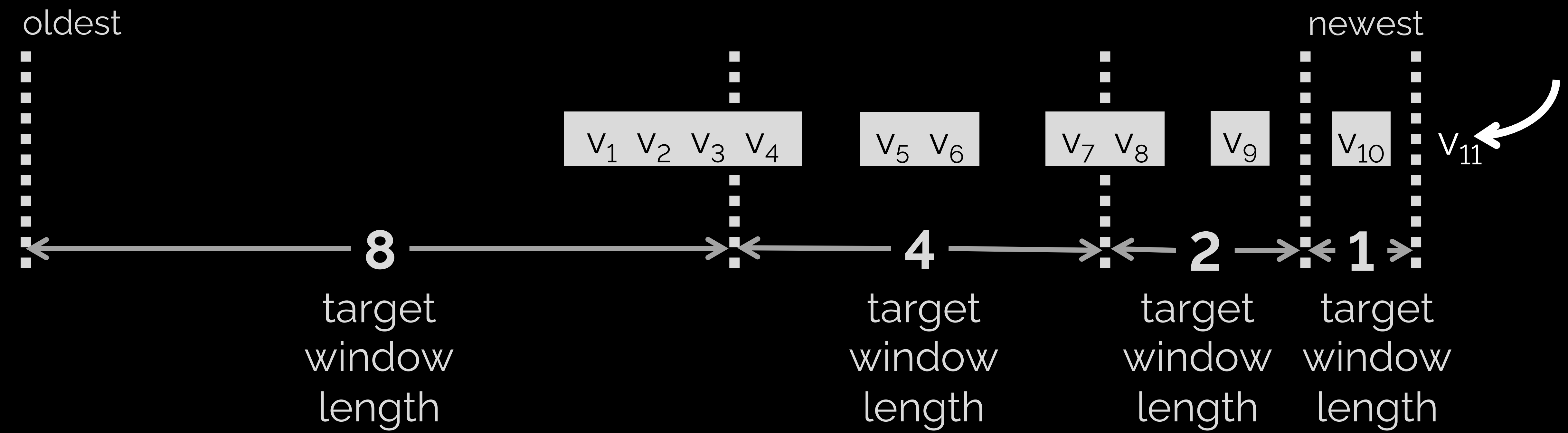
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



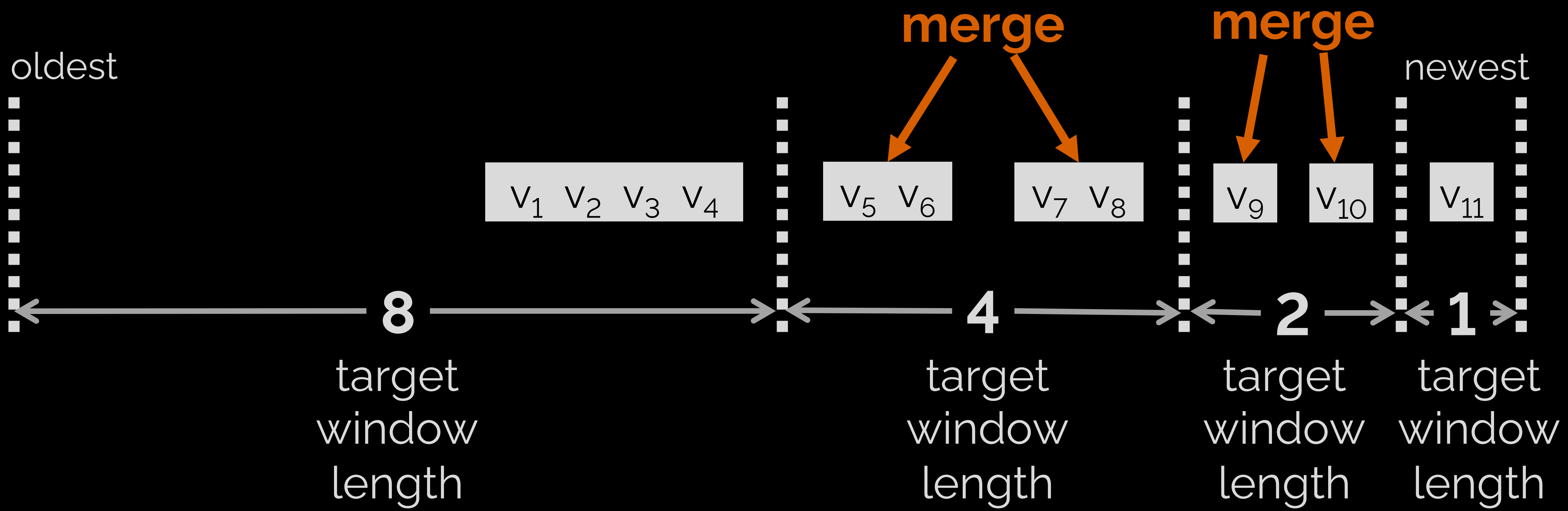
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



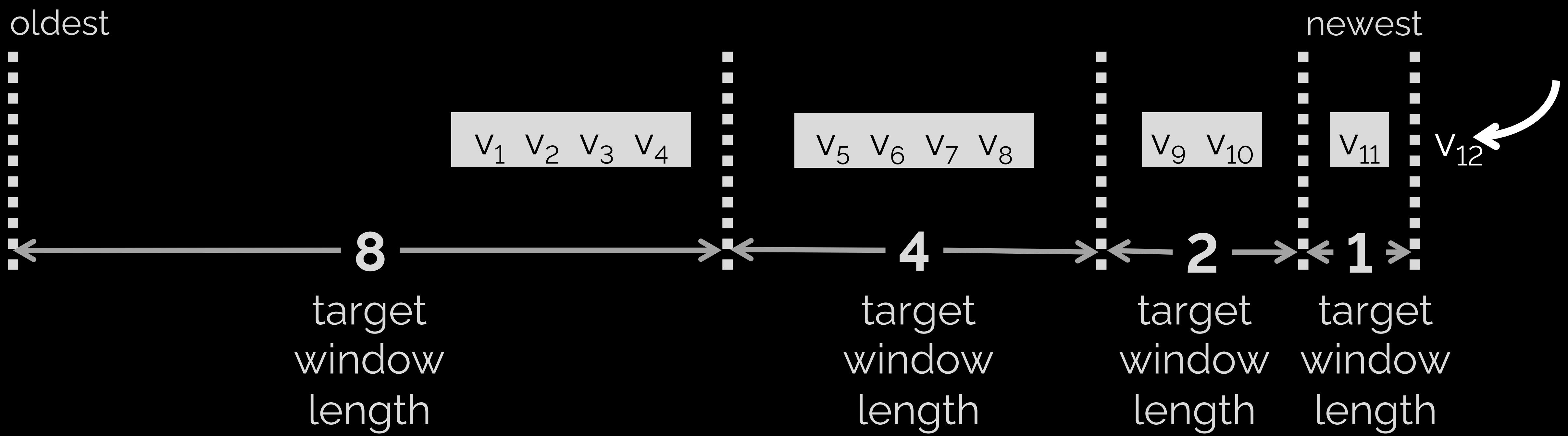
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



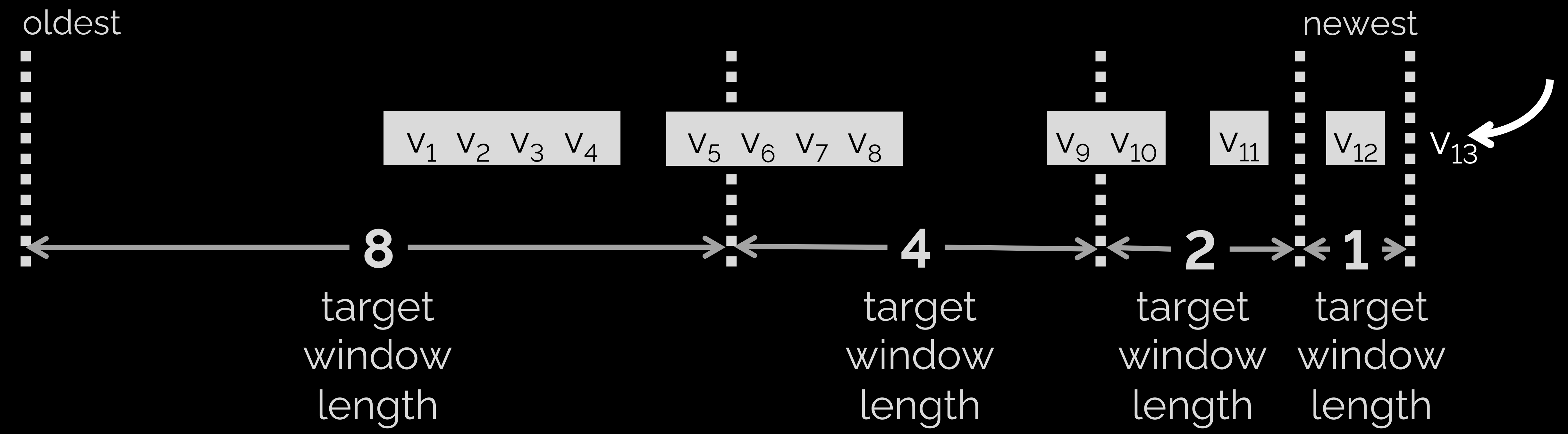
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



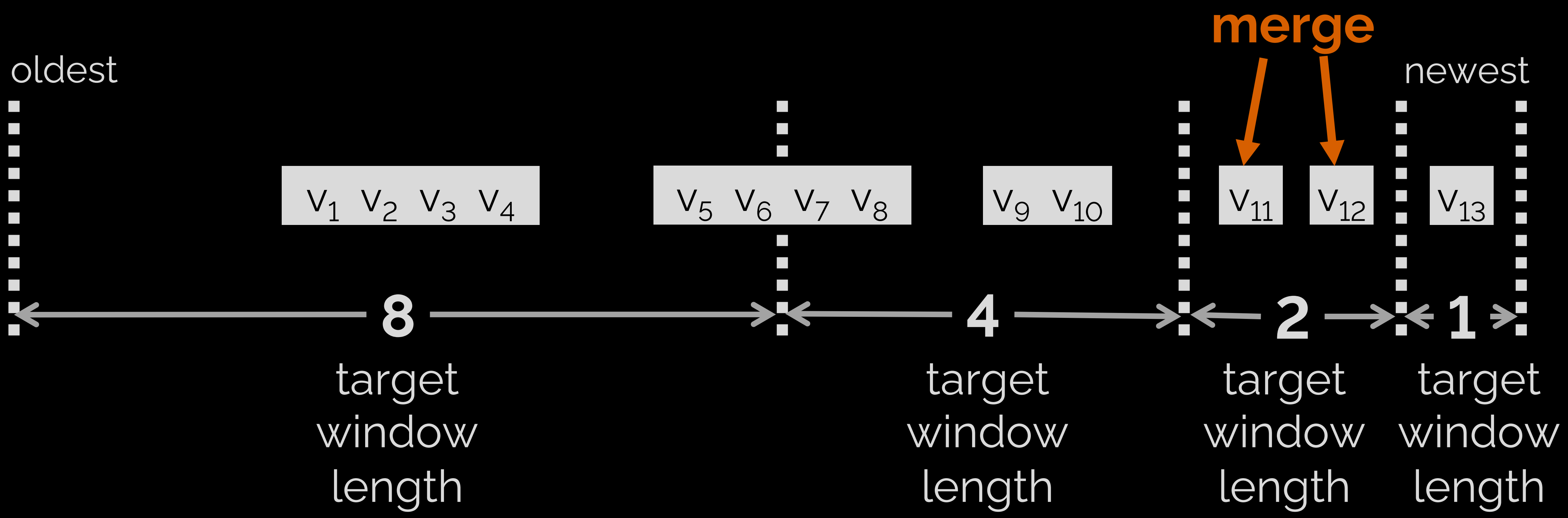
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



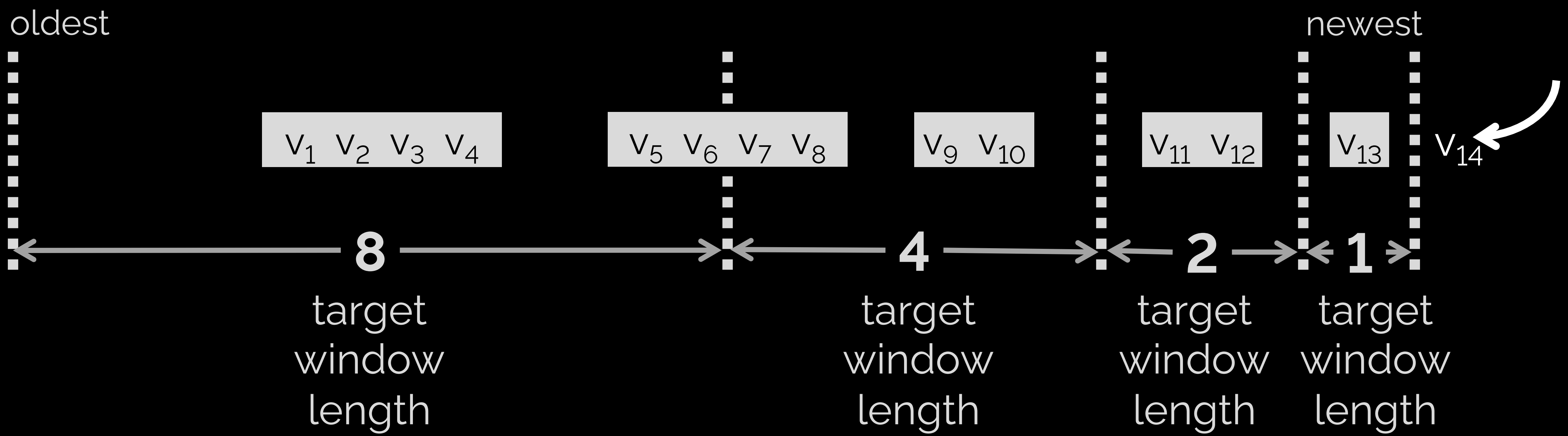
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



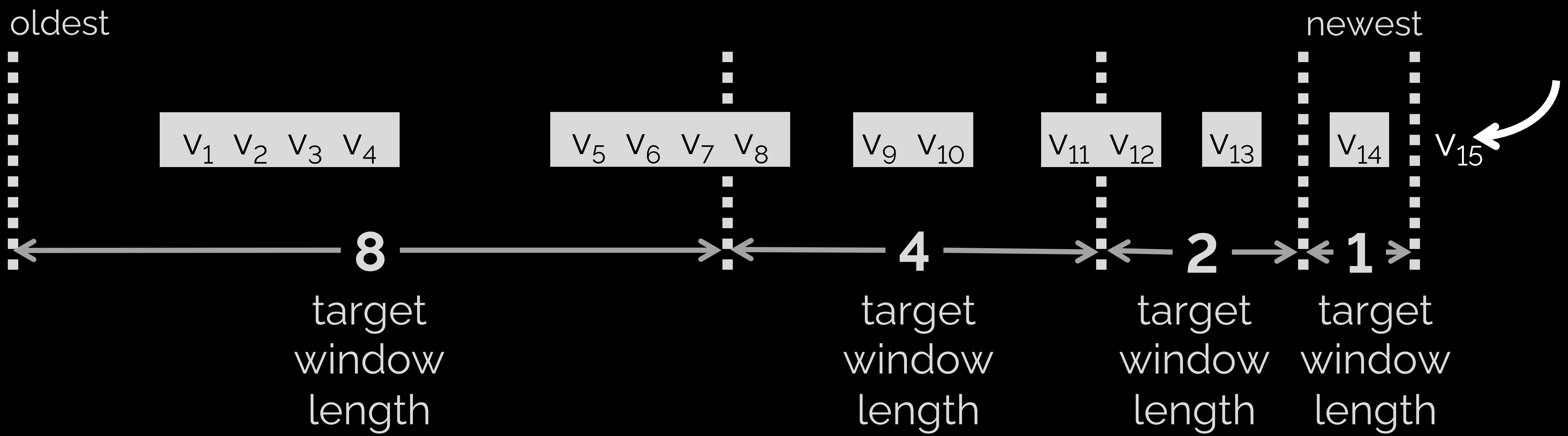
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



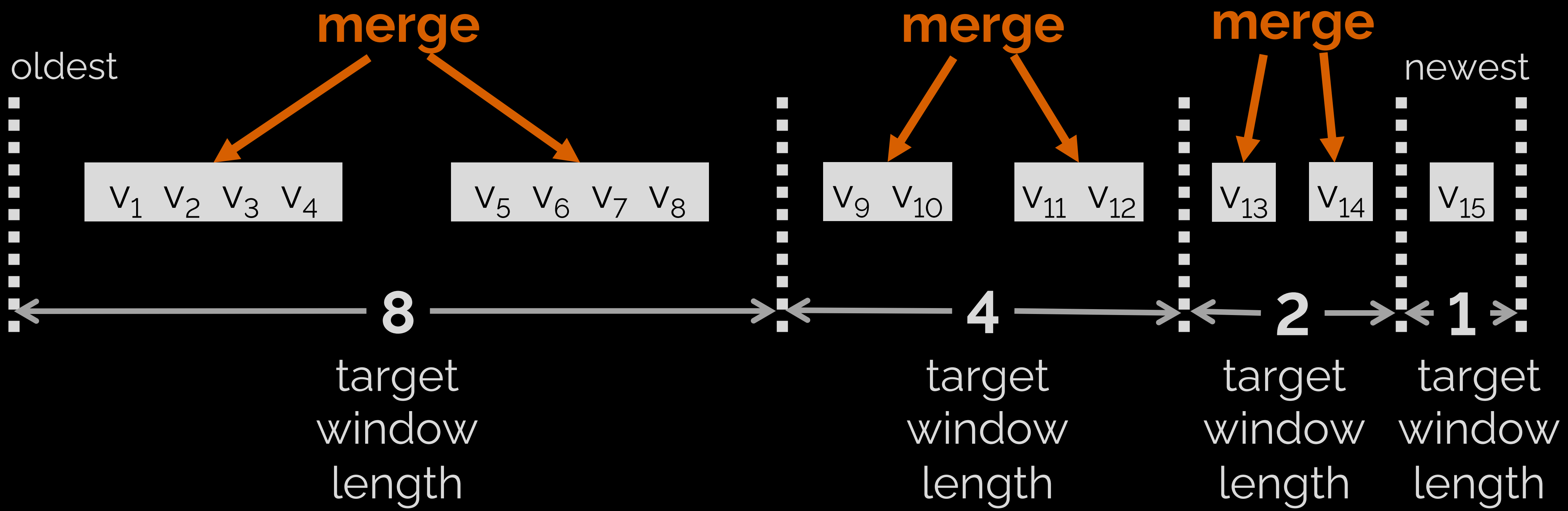
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



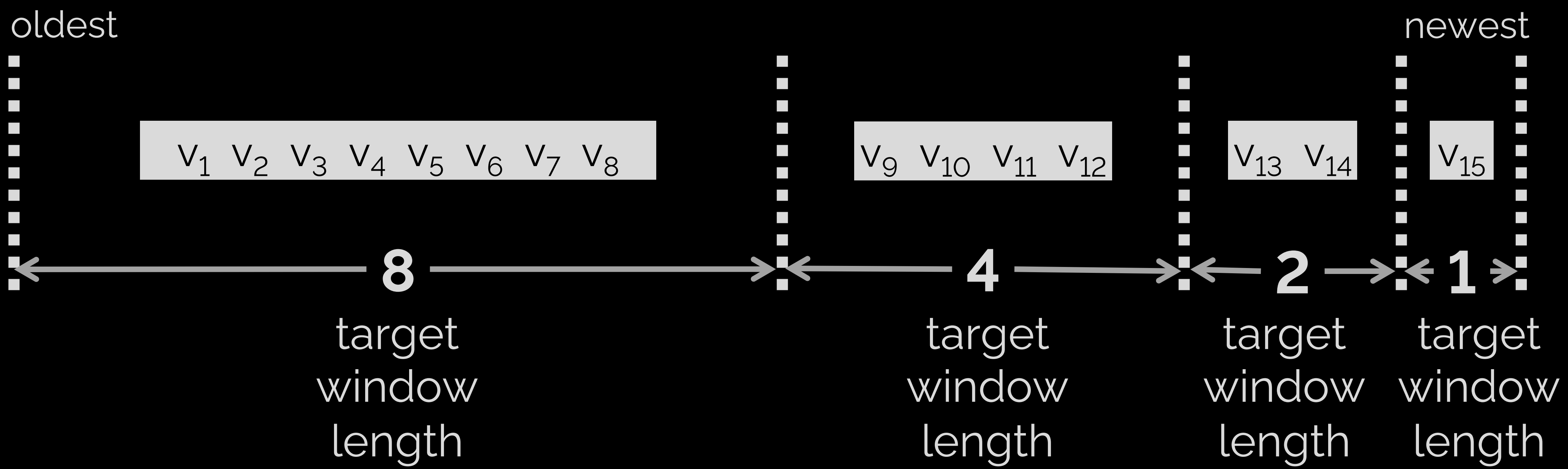
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



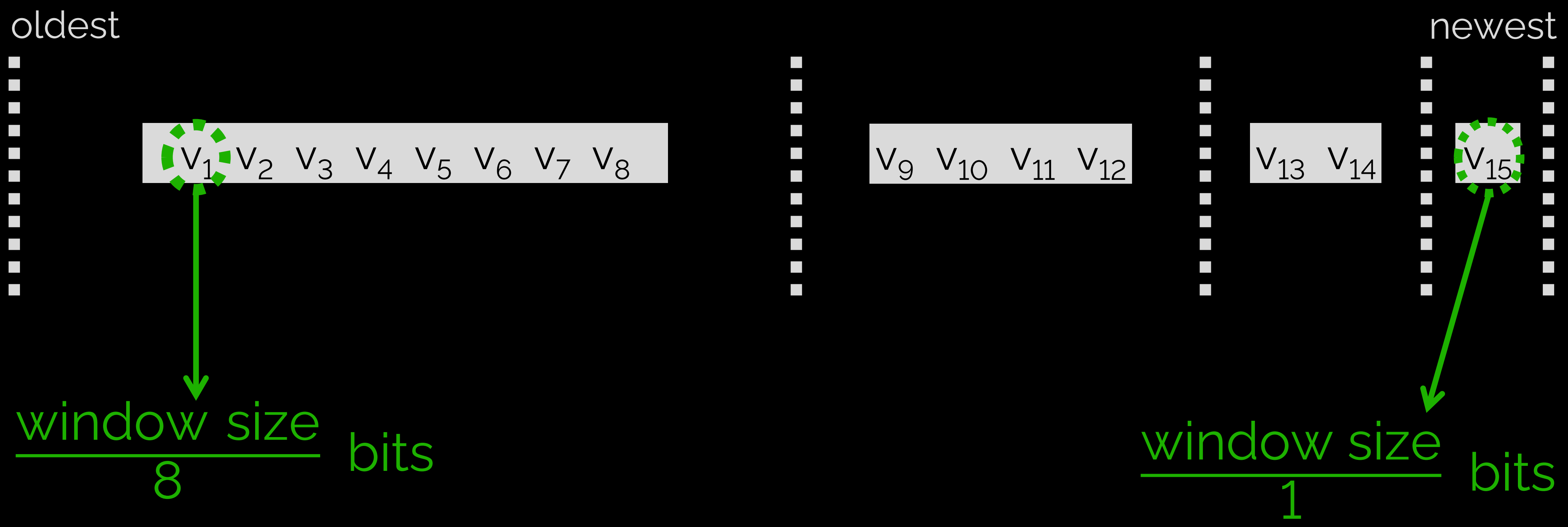
Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines



Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines

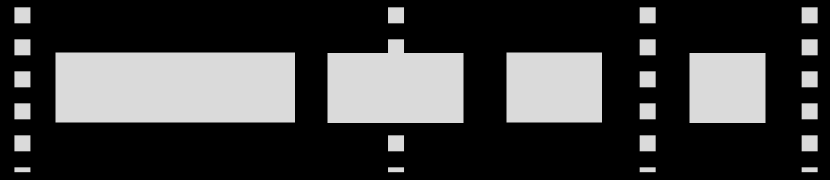


Ingest algorithm

1. When a value arrives, create new window of length 1 to hold it
2. Merge any consecutive windows contained in same pair of dotted lines

Why this algorithm works:

- At any point in time:
  - No more than one actual window inside any target window (between a pair of dotted lines)



  - Thus number of actual windows ≤ 2 x number of target windows
- By induction: # bits allocated to any datum is always within 2x of target

Outline

1. Time-decayed stream representation
2. Processing writes
3. Handling queries
4. Evaluation

Query interface: Time-range queries

query a summary over
the time-range $[T_1, T_2]$

Examples

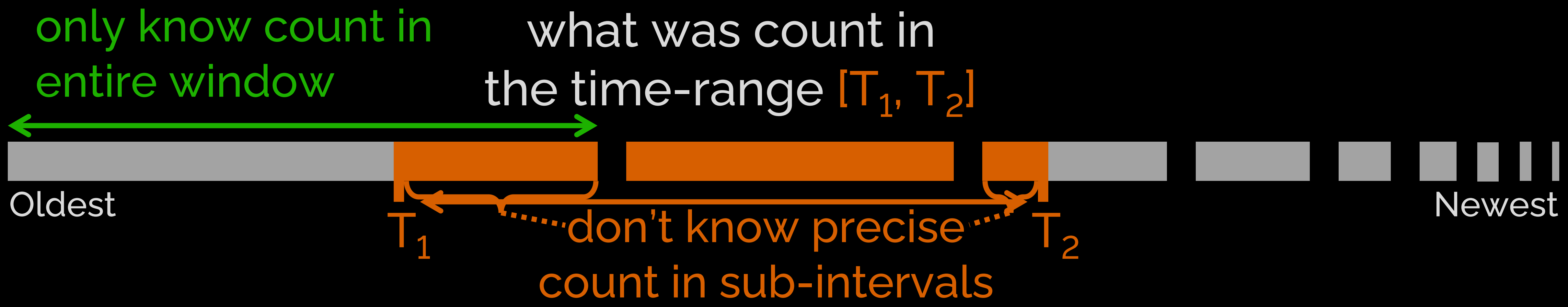
- What was average energy usage in Sep 2015?
- Fetch a random (time-decayed) sample over the last 1 year

Query interface: Time-range queries

query a summary over
the time-range $[T_1, T_2]$

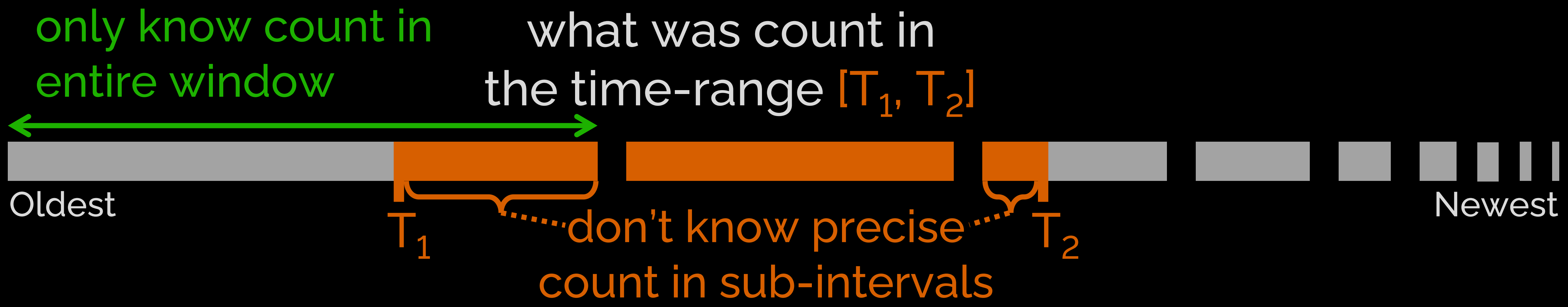
Time-ranges are allowed to be arbitrary, need not be window-aligned

Query interface: Time-range queries



Time-ranges are allowed to be arbitrary, need not be window-aligned

Query interface: Time-range queries



Time-ranges are allowed to be arbitrary, need not be window-aligned

Lack of window alignment introduces error

We use novel low-overhead statistical techniques to estimate answer & confidence interval

Query accuracy

Age = how far back in time query goes

- Lower age ⇒ more recent data, so better accuracy

Length = time-span query covers

- Longer length ⇒ more windows spanned, so better

Not suited for large age + small length

- e.g. query over the time range
  [10 years ago, 10 years ago + 3 seconds]

length \ **age**	new	med	old
long	✓	✓	✓
med	✓	✓	–
short	✓	–	× not suited

Outline

1. Time-decayed stream representation
2. Processing writes
3. Handling queries
4. Evaluation

Evaluation

On a single node: 224 GB RAM, 10 x 1 TB disk

Microbenchmarks: 1 PB on single node

Real applications

- Forecasting
- Outlier analysis
- Analyzing network traffic and data backup logs

Evaluation

On a single node: 224 GB RAM, 10 x 1 TB disk

Microbenchmarks: 1 PB on single node

Real applications

- Forecasting
- Outlier analysis
- Analyzing network traffic and data backup logs

1. Microbenchmarks: 1 PB on a single node

1 PB on a single node: Setup

Dataset: 1024 x 1 TB streams, each randomly generated

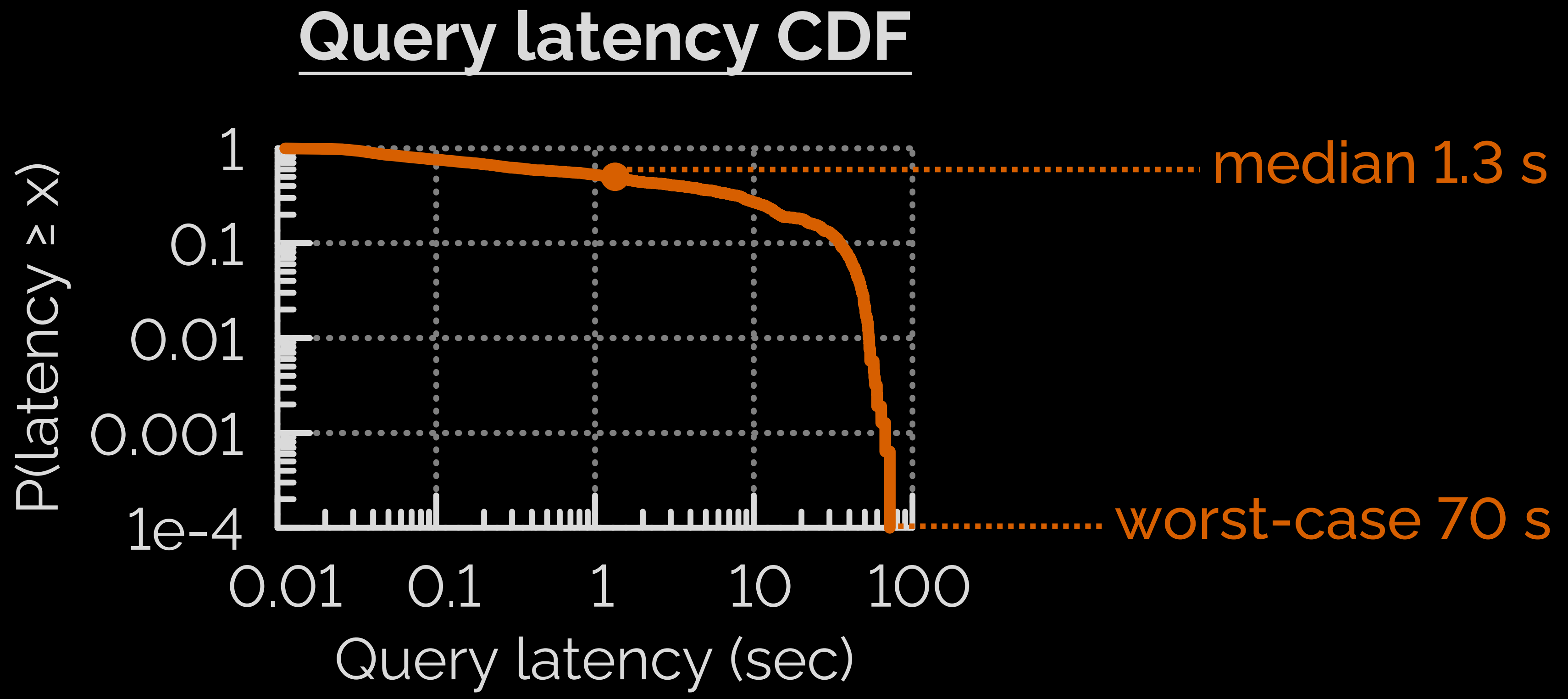
- Poisson, Pareto arrival process
- Uniform random values

Compacted 100x (down to 10 TB)

Queries: randomly generated

- Count, Sum, Frequency, Existence
- Each query picks a random stream and a random time interval
    - Spans up to 1 TB raw data

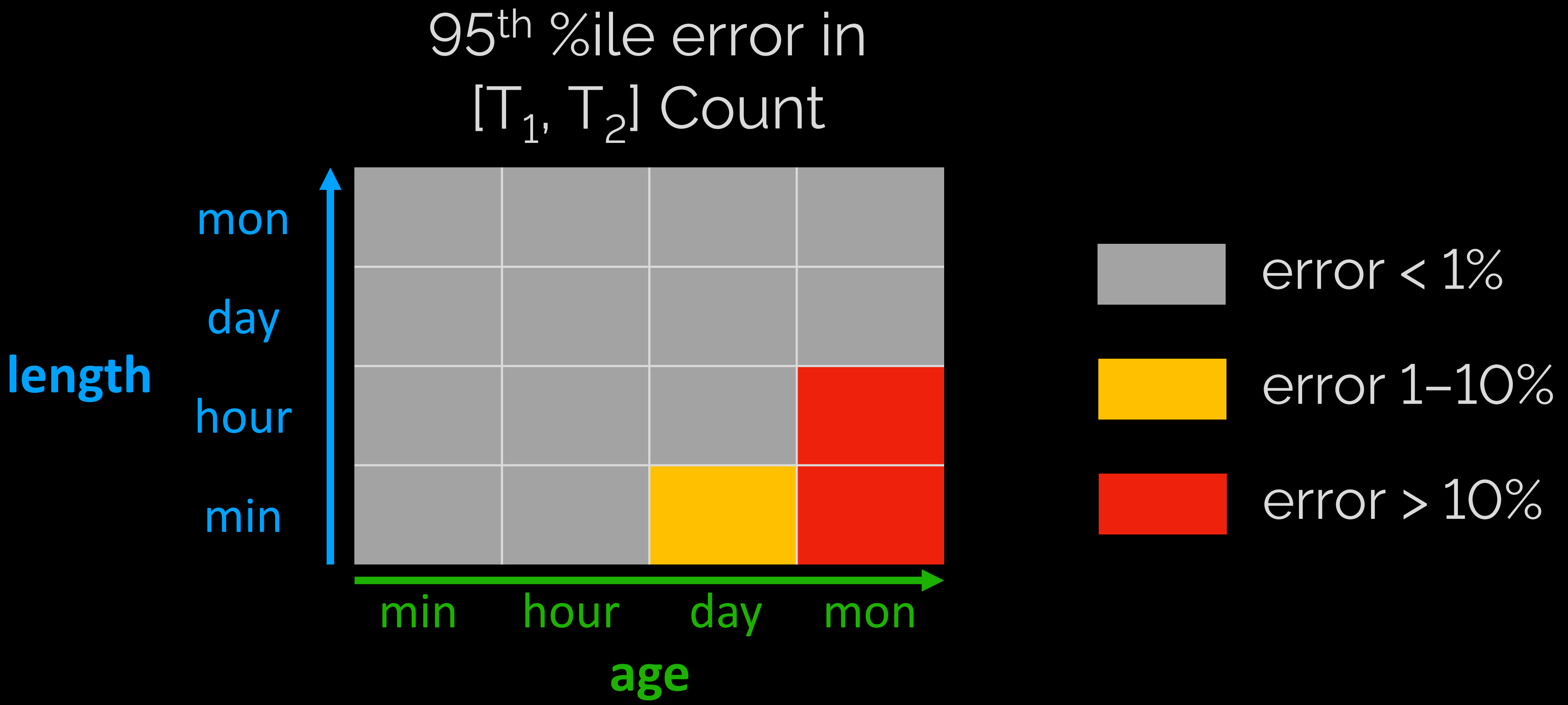
1 PB on a single node: Latency



Above latencies were with all caches disabled (conservative bound)

With caching: varies, < 1s 95th %ile w/ reasonable locality

1 PB on a single node (compacted 100x): Accuracy



1 PB on a single node (compacted 100x): Accuracy

Existence
(Bloom filter)

Count

Frequency
(count-min sketch)

2. Real application: Time-series forecasting w/ Prophet

Time-series forecasting w/ Prophet

Prophet: open-source forecasting library from Facebook

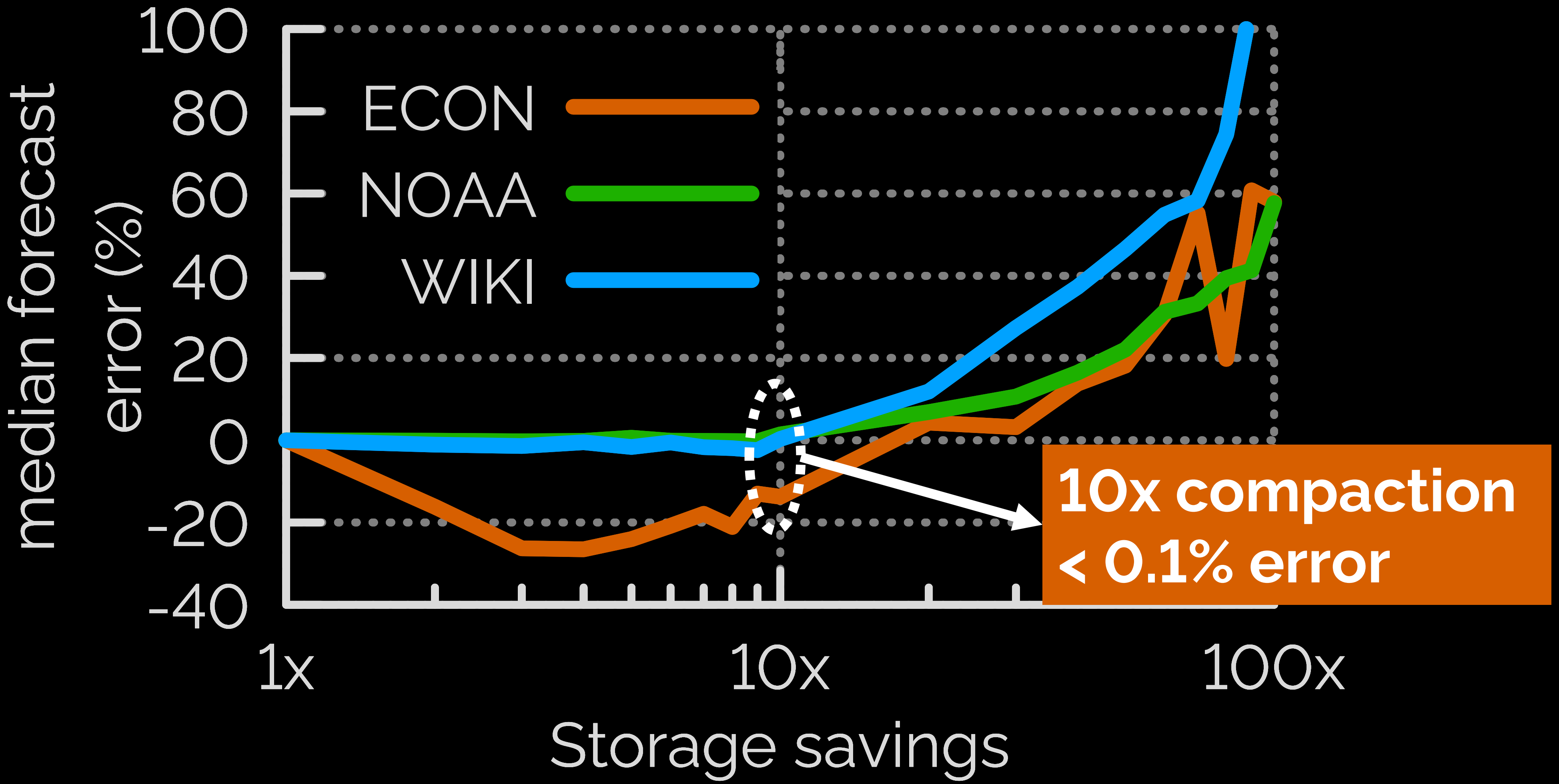
Tested three datasets

- WIKI: visit counts for Wikipedia pages
- NOAA: global surface temperature readings
- ECON: log of US economic indicators

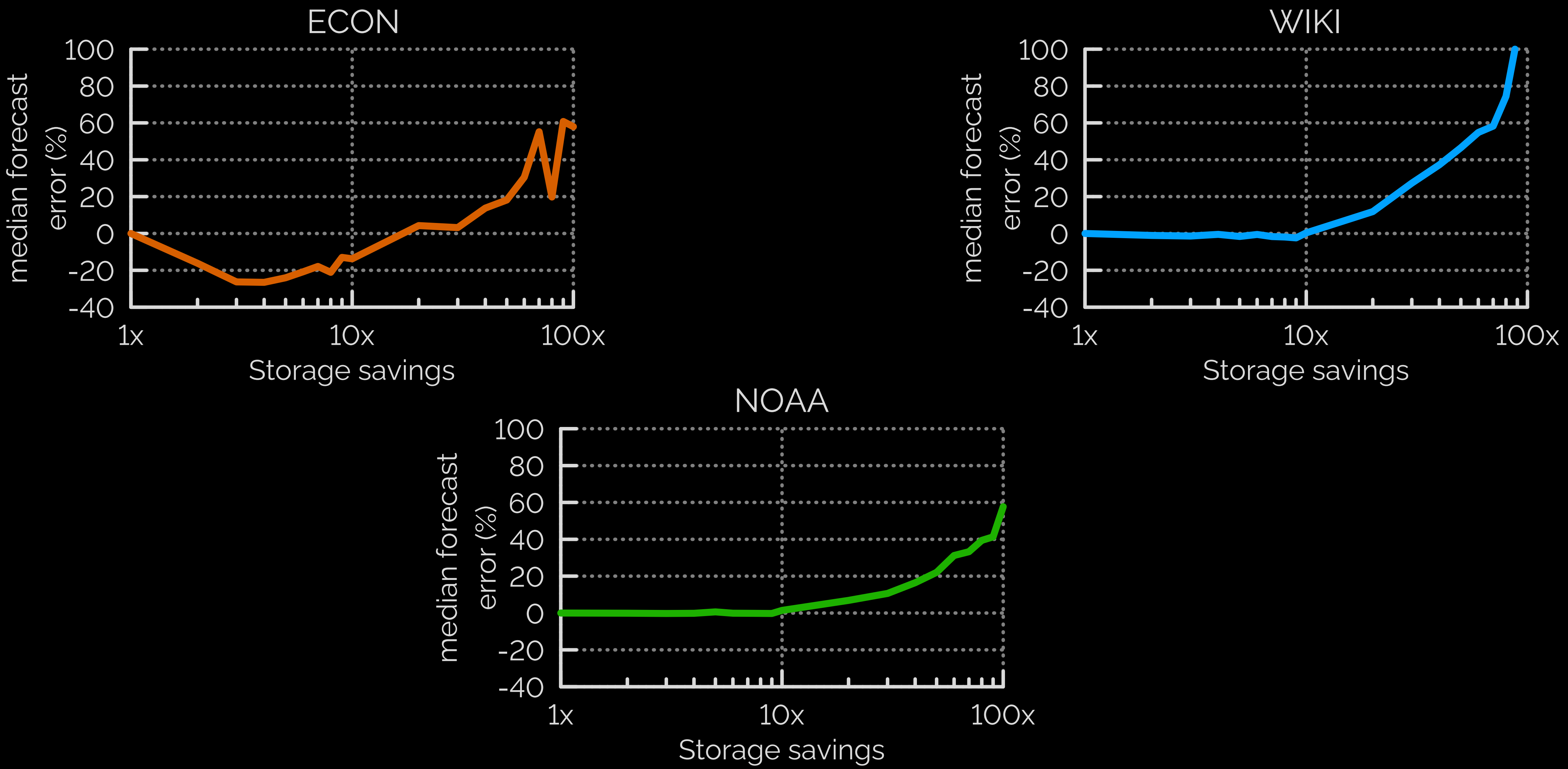
On each time-series in each dataset, compared forecast accuracy of

- Model trained on all data
- Model trained on time-decayed sample of data

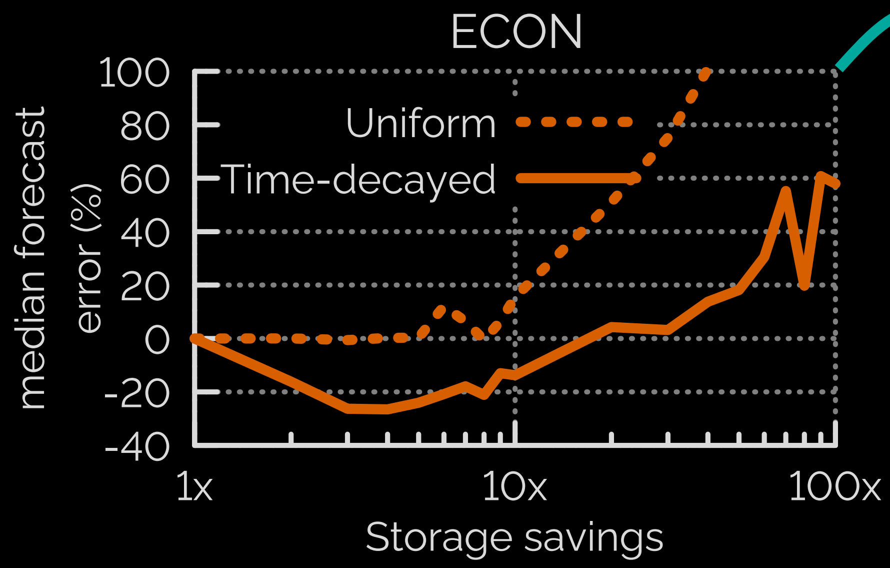
Time-series forecasting w/ Prophet

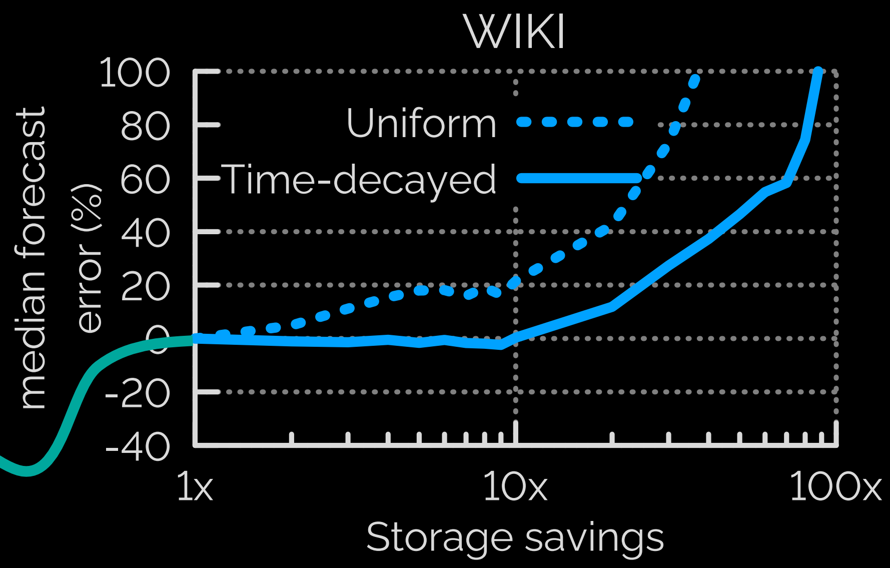


Time-series forecasting w/ Prophet

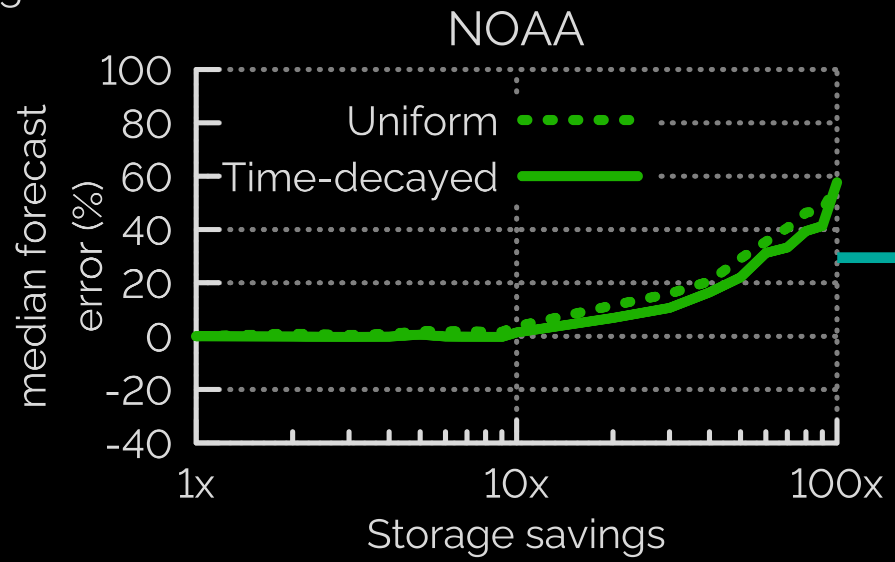


Time-series forecasting w/ Prophet





substantial improvement



difference not as stark because of predictable dataset

More details in paper

Landmarks

System design

System configuration

Statistical techniques for sub-window queries

More details in paper

Landmarks

System design

System configuration

Statistical techniques for sub-window queries

Landmarks

Mechanism for protecting specific values from decay

Values declared as landmarks are

- Always stored at full resolution
- Seamlessly combined with decayed data when answering queries

Example application: outlier analysis

SummaryStore: approximate store for stream analytics

Contributions

- Abstraction: time-decayed summaries + landmarks
- Data ingest mechanism
- Low-overhead statistical techniques bounding query error

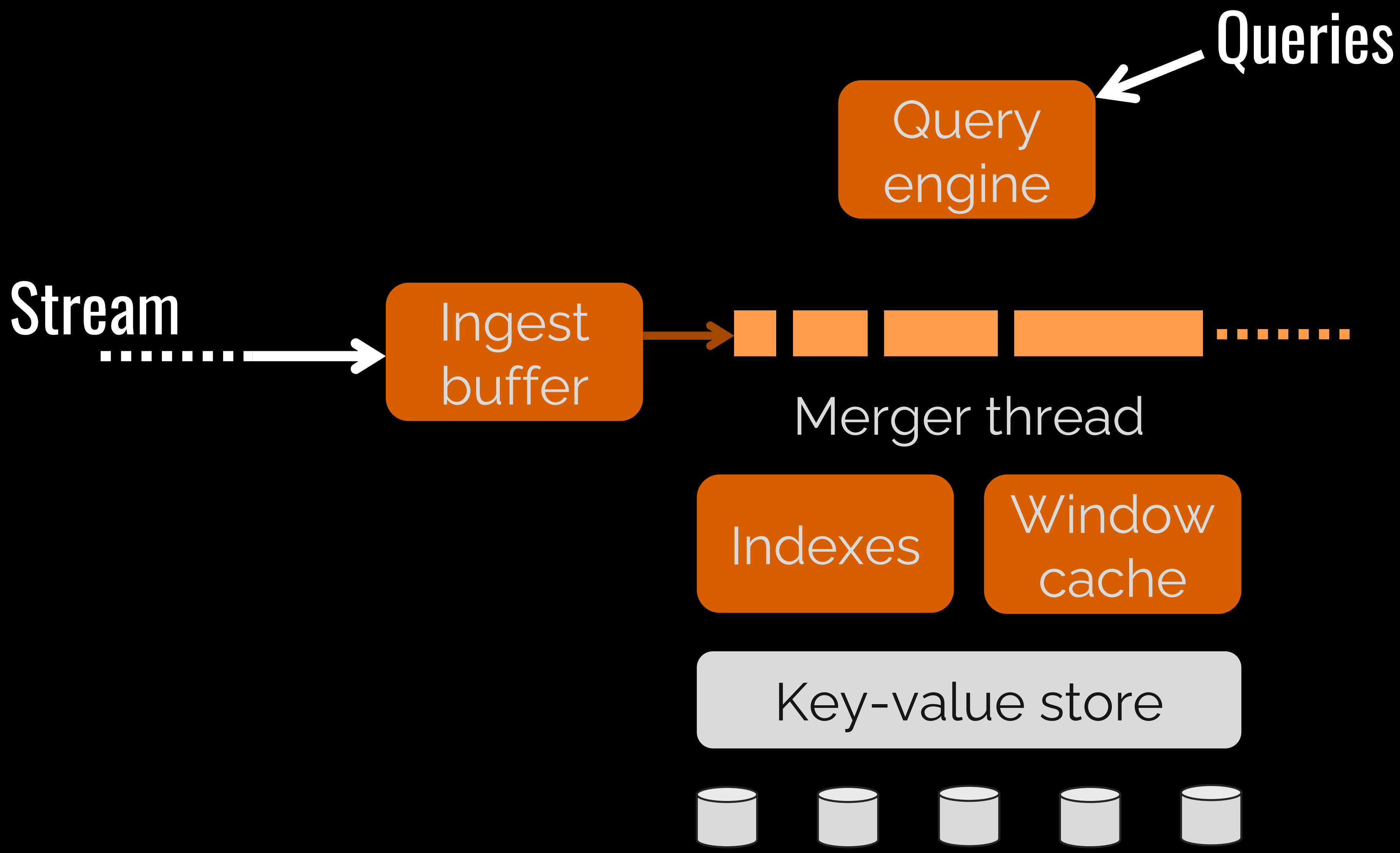
Works well in real applications and microbenchmarks:

- 10-100x compaction, warm-cache latency < 1s, low error
- 1 PB on a single node (summarized down to 10 TB)

Project details at https://bit.do/summarystore

Backup slides

System architecture



Configuration

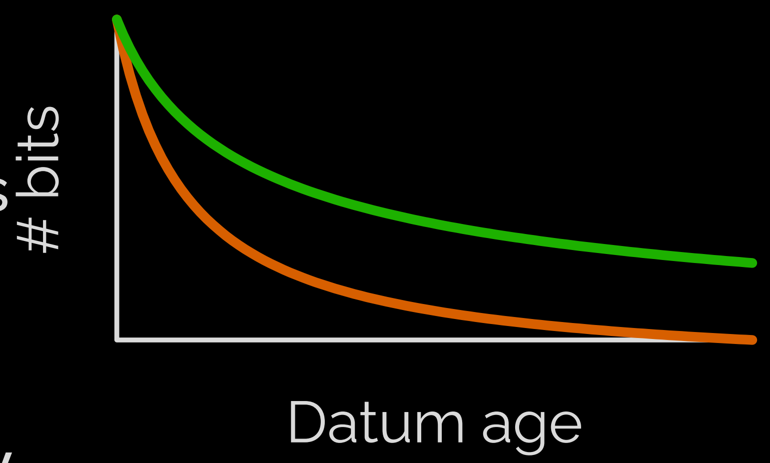
For each stream, users configure

1. List of summaries to maintain per-window
   - Built-in: Sum, Count, Histogram, Bloom Filter, Random Sample, …
   - Pluggable interface to add more
     - Allows adapting existing non-streaming summarization techniques

Configuration

For each stream, users configure

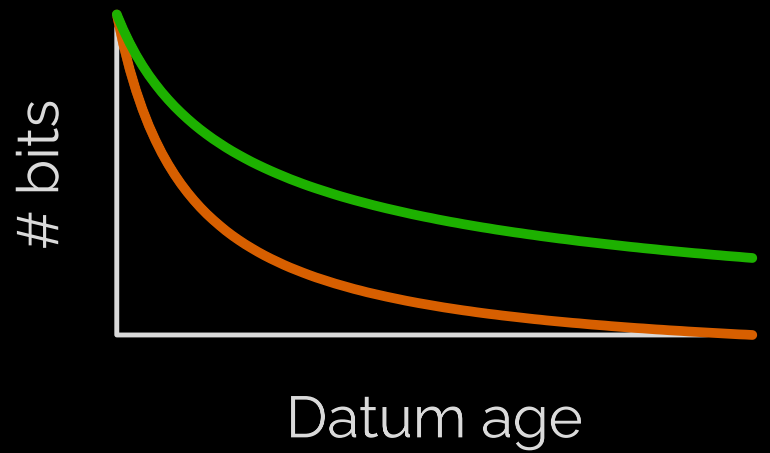
1. List of summaries to maintain per-window
2. Sequence of window lengths: controls decay
   - With window lengths = 1, 2, 4, 8, 16, 32, …; after N inserts
     - Storage footprint = $O(\log_2 N)$
     - # bits to nth oldest element = $O(1 / n)$
   - With window lengths = 1, 2, 3, 4, 5, 6, …; after N inserts
     - Storage footprint = $O(\sqrt{N})$
     - # bits to nth oldest element = $O(1 / \sqrt{n})$
   - Choice affects storage compaction, accuracy



Configuration

For each stream, users configure

1. List of summaries to maintain per-window
2. Sequence of window lengths: controls decay
   - Don't actually need to provide full list of window lengths
   - Parametric family of **power-law decay** functions
   - Simple 4 parameter API



Configuration

	Individual stream size		
PowerLaw (p, q, R, S)	*10 GB*	*100 GB*	*1000 GB*
(1, 1, 88, 1)	1.1x	3.4x	11x
(1, 1, 16, 1)	2.5x	7.9x	25x
(1, 1, 8, 1)	3.5x	11x	35x
(1, 1, 4, 1)	5x	16x	50x
(1, 1, 1, 1)	10x	32x	100x
(1, 2, 48, 1)	22x	100x	480x
(1, 2, 5, 1)	100x	460x	2200x
Exponential (b, R, S)	*10 GB*	*100 GB*	*1000 GB*
(2, 88, 1)	120x	1100x	9700x
(2, 32, 1)	320x	2800x	25000x
(2, 1, 1)	8600x	77000x	700000x
(3, 1, 1)	14000x	120000x	1100000x

Table 5: Storage compaction evolution w/ decay configurations. **Column name = size of raw data, increasing over time; compaction = (size of raw data)/(size of SummaryStore). The parameters of the power-law decay function map to different window lengths and consequently different compactions; admins can refer to table as rule-of-thumb for configuring.**

SummaryStore API

CreateStream(decay function, [list of summary operators])

DeleteStream(streamID)

Append(streamID, [timestamp], value)

BeginLandmark(streamID)

EndLandmark(streamID)

Query(stream, T_1, T_2, operator, params)

QueryLandmark(stream, T_1, T_2)

Statistical techniques for sub-window queries

Query	Method for Error Estimation
count[a, a+t] (generic)	$N\left(C\frac{t}{T}, \left(\frac{\sigma_t}{\mu_t}\right)^2 \frac{T}{\mu_t}\frac{t}{T}\left(1-\frac{t}{T}\right)\right)$
count[a, a+t] (Poisson)	Binomial $\left(C, \frac{t}{T}\right)$
sum[a, a+t]	$N\left(S\frac{t}{T}, \left(\frac{\sigma_t^2}{\mu_t^2}+\frac{\sigma_v^2}{\mu_v^2}\right)\frac{T\mu_v^2}{\mu_t}\frac{t}{T}\left(1-\frac{t}{T}\right)\right)$
membership(v)[a, a+t]	For Bloom filter with FP probability p: $p\frac{t}{T}$
membership(v)[a, a+t]	For CMS: Pr (Hypergeom$(C, S, V) > 0)$
frequency(v)[a, a+t]	Hypergeom(C, S, V)
S = normal distribution of count[a, a+t] (generic) in first row V = distribution over frequency(v)[entire window]; v refers to values	

Table 6: Statistical methods for sub-window queries.

Statistical techniques for sub-window queries

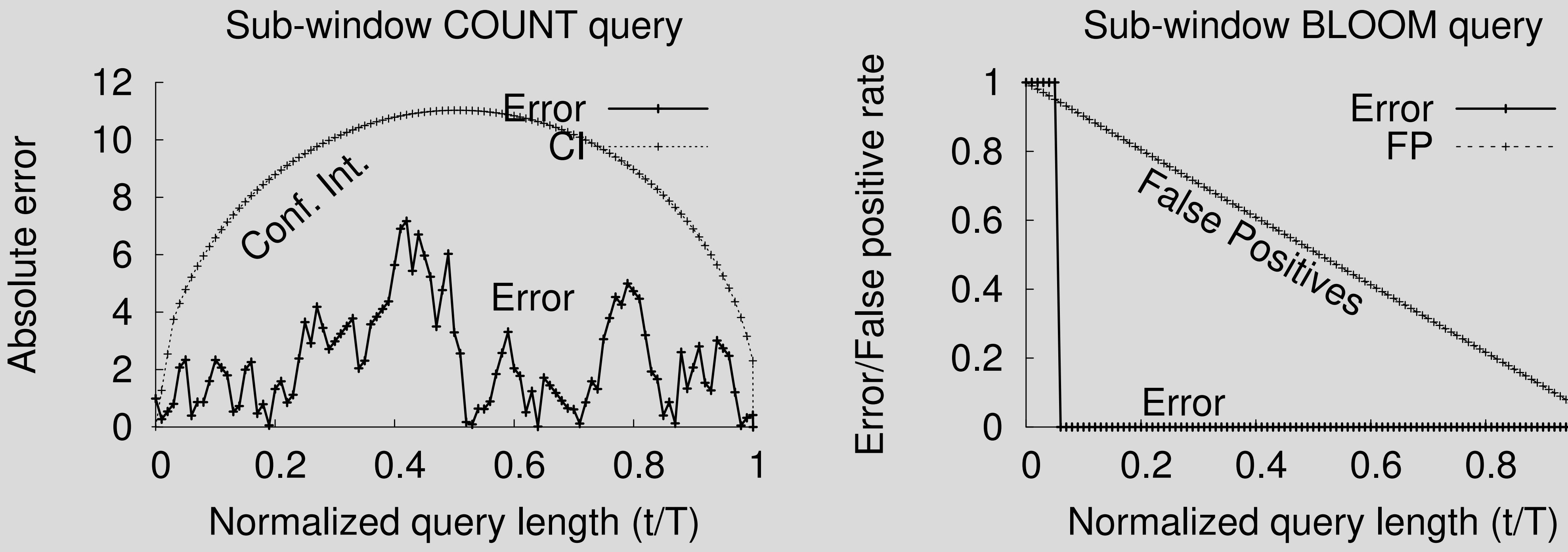


Figure 12: Sub-window answers and error estimates.

Combining decayed and landmark data when answering queries

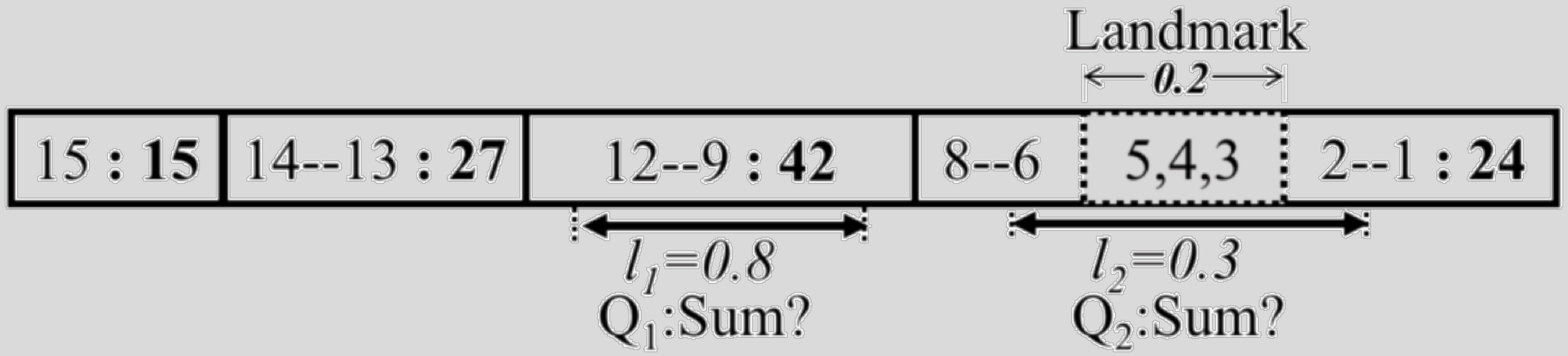


Figure 4: Challenge in answering sub-window queries for Summarized and Landmark windows. Q_1 asks Sum over summarized only, Q_2 over summarized and landmark. For the first window, 24 refers to the sum for 8—6 and 2—1; {3,4,5} are excluded from summaries and included in the landmark window.

Evaluation: 1 PB at 100x compaction

Error with infinite variance Pareto arrivals

Error in [T_1, T_2] Count

	min	hour	day	mon
mon	0.0	0.0	0.0	0.0
day	0.0	0.0	0.0	0.0
hour	0.0	0.0	0.0	5.2e4
min	0.001	0.003	0.01	7.7e4

not suited

Evaluation: 1 PB at 100x compaction

Error with Poisson arrivals

Error in $[T_1, T_2]$ Count

Length \ Age	min	hour	day	mon
mon	0.0	0.0	0.0	0.0
day	0.0	0.0	0.0	0.0
hour	0.0	0.0	0.0	0.0
min	0.0	0.0	0.002	0.001

Evaluation: 1 PB at 100x compaction

Latency breakdown

Latency (seconds)

	min	hour	day	mon
mon	54	50	47	34
day	15	14	9	4.5
hour	1.8	1.2	0.6	0.2
min	0.3	0.1	0.05	0.03

Evaluation: Smaller dataset, lower compaction

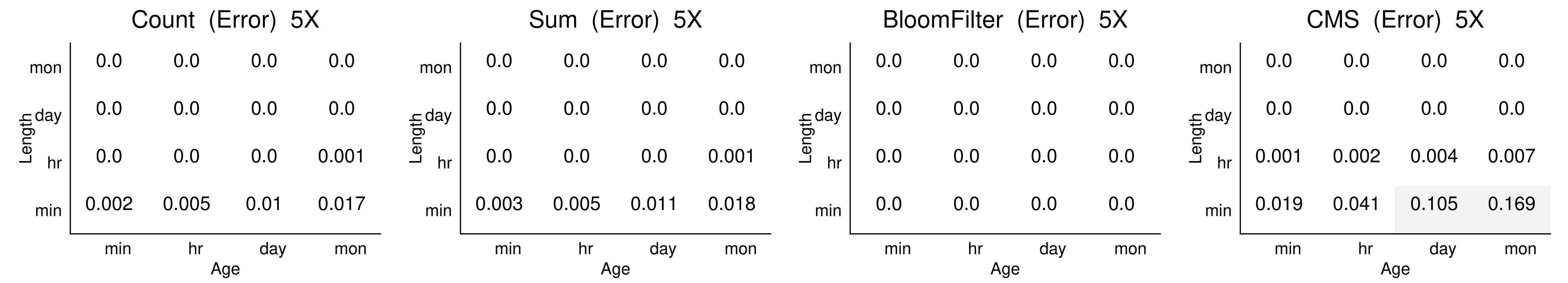
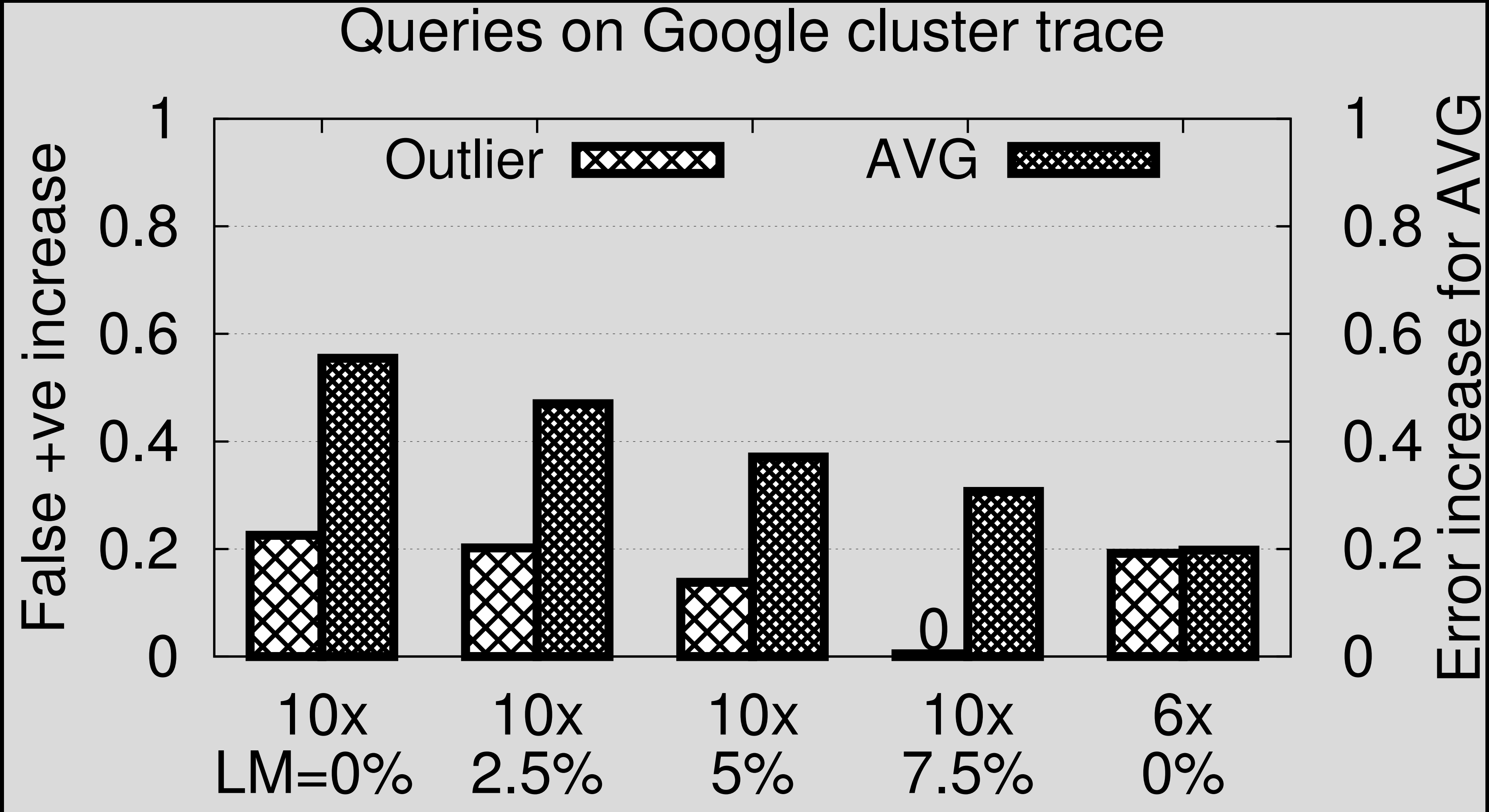


Figure 11: Error heatmap. **Slow Velocity Poisson(λ = 10), 5 GB/stream, Lower Compaction (5x)**

Evaluation: Outlier analysis on Google cluster trace



Deleted slides

Design goals

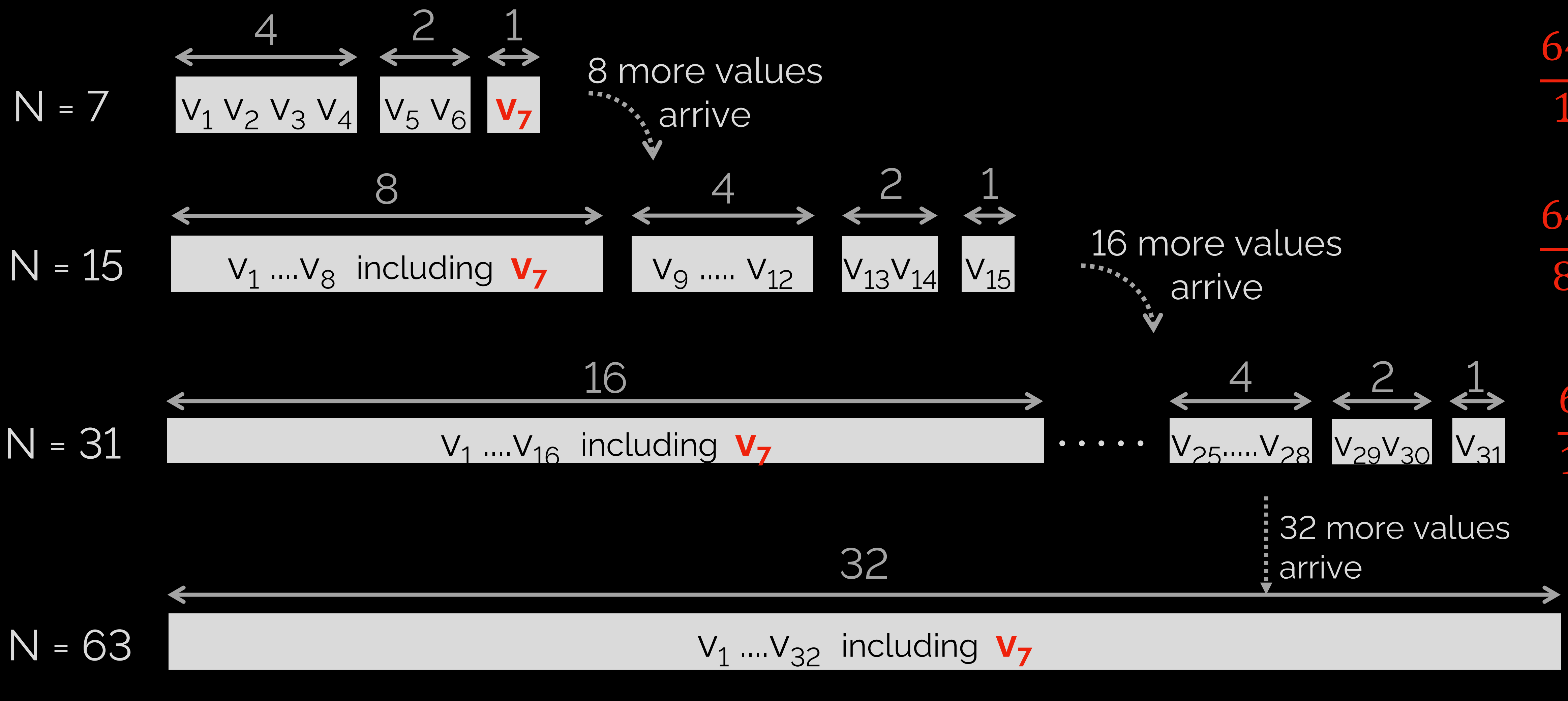
1. Allow reusing work on non-streaming approx
   - Substantial body of work on approximating static datasets
   - E.g. sampling [BlinkDB, QuickR], histograms [SQL DBs], other sketches, …
   - No clear winner, each suitable for different applications
   - Orthogonal to policy on time

2. Support configurable time-decay
   - Tune storage, accuracy

Our solution: decay through **windowed summarization**

Data decays as it ages

Assuming each window is 64 bits, # bits used for v_7 =



$$\frac{64}{1} = 64$$

$$\frac{64}{8} = 8$$

$$\frac{64}{16} = 4$$

$$\frac{64}{32} = 2$$