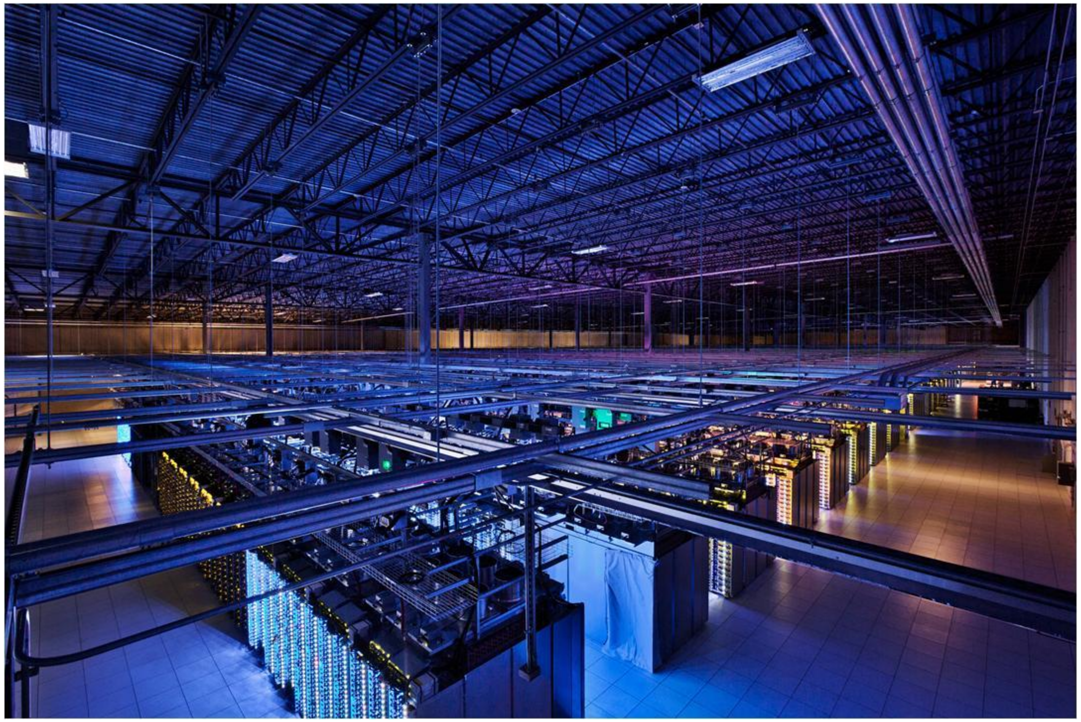
Centrum Wiskunde & Informatica

# Big Data for Data Science

## Data streams and low latency processing



event.cwi.nl/lsde

# DATA STREAM BASICS

# What is a data stream?

- Large data volume, likely structured, arriving at a very high rate
  - Potentially high enough that the machine cannot keep up with it
- Not (only) what you see on youtube
  - Data streams can have structure and semantics, they're not only audio or video

- Definition (Golab and Ozsu, 2003)
  - A data stream is a real-time, continuous, ordered (implicitly by arrival time of explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor it is feasible to locally store a stream in its entirety.

# Why do we need a data stream?

- Online, real-time processing
- Potential objectives
  - Event detection and reaction
  - Fast and potentially approximate online aggregation and analytics at different granularities
- Various applications
  - Network management, telecommunications
    Sensor networks, real-time facilities monitoring
  - Load balancing in distributed systems
  - Stock monitoring, finance, fraud detection
  - Online data mining (click stream analysis)

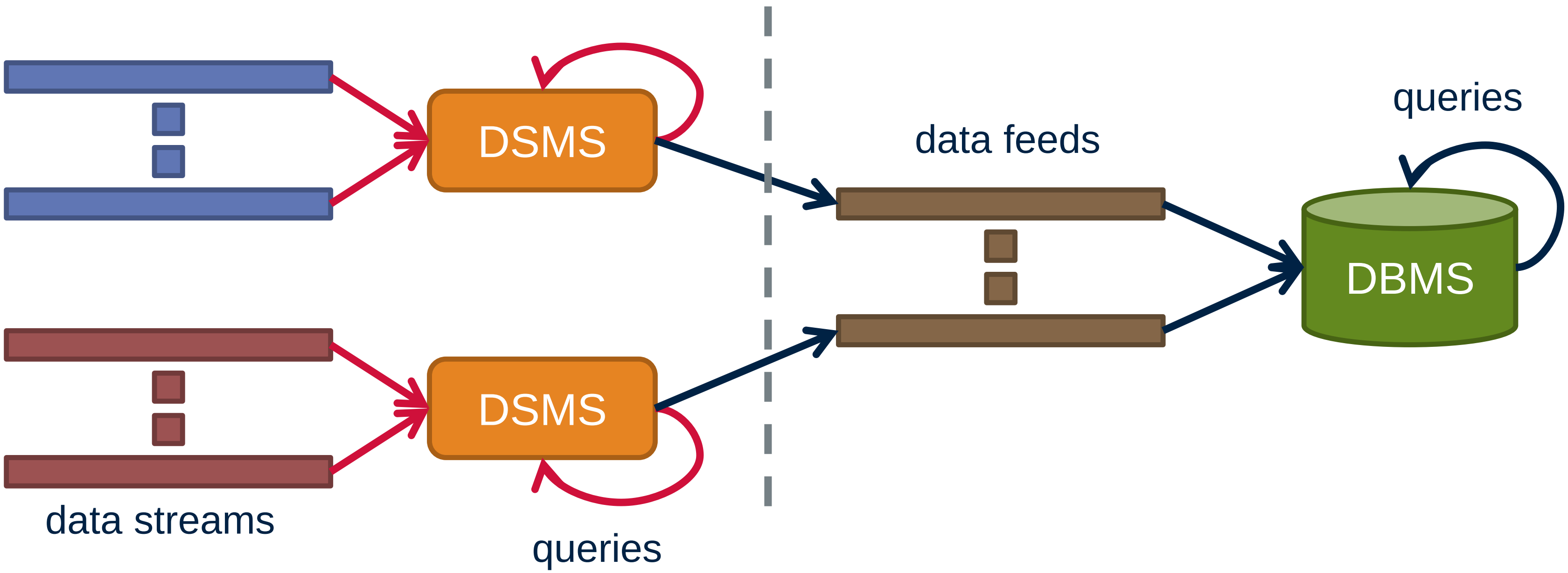
# Example uses

- Network management and configuration
  - Typical setup: IP sessions going through a router
  - Large amounts of data (300GB/day, 75k records/second sampled every 100 measurements)
  - Typical queries
    - What are the most frequent source-destination pairings per router?
    - How many different source-destination pairings were seen by router 1 but not by router 2 during the last hour (day, week, month)?
- Stock monitoring
  - Typical setup: stream of price and sales volume
  - Monitoring events to support trading decisions
  - Typical queries
    - Notify when some stock goes up by at least 5%
    - Notify when the price of XYZ is above some threshold and the price of its competitors is below than its 10 day moving average

# Structure of a data stream

- Infinite sequence of items (elements)
- One item: structured information, i.e., tuple or object
- Same structure for all items in a stream
- Timestamping
  - Explicit: date/time field in data
  - Implicit: timestamp given when items arrive
- Representation of time
  - Physical: date/time
  - Logical: integer sequence number

# Database management vs. data stream management




- Data stream management system (DSMS) at multiple observation points
  - Voluminous streams-in, reduced streams-out
- Database management system (DBMS)
  - Outputs of data stream management system can be treated as data feeds to database

# DBMS vs. DSMS

- DBMS
  - Model: persistent relations
  - Relation: tuple set/bag
  - Data update: modifications
  - Query: transient
  - Query answer: exact
  - Query evaluation: arbitrary
  - Query plan: fixed

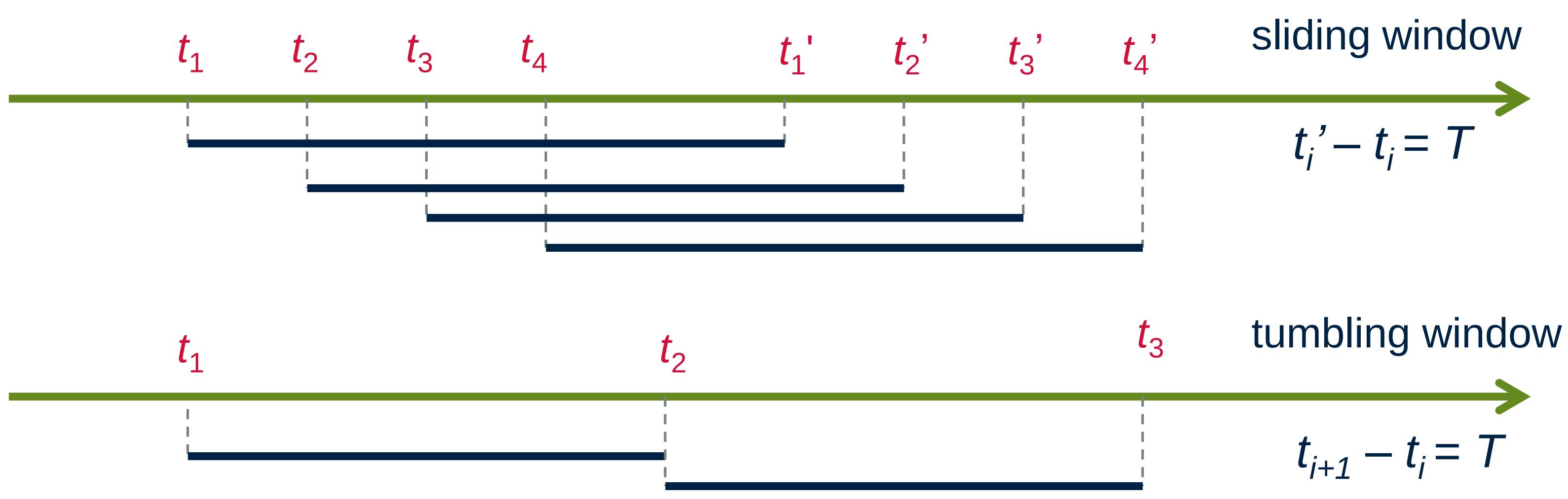
- DSMS
  - Model: transient relations
  - Relation: tuple sequence
  - Data update: appends
  - Query: persistent
  - Query answer: approximate
  - Query evaluation: one pass
  - Query plan: adaptive

# Windows

- Mechanism for extracting a finite relation from an infinite stream
- Various window proposals for restricting processing scope
  - Windows based on ordering attributes (e.g., time)
  - Windows based on item (record) counts
  - Windows based on explicit markers (e.g., punctuations) signifying beginning and end
  - Variants (e.g., some semantic partitioning constraint)

# Ordering attribute based windows

- Assumes the existence of an attribute that defines the order of stream elements/records (e.g., time)
- Let *T* be the window length (size) expressed in units of the ordering attribute (e.g., *T* may be a time window)



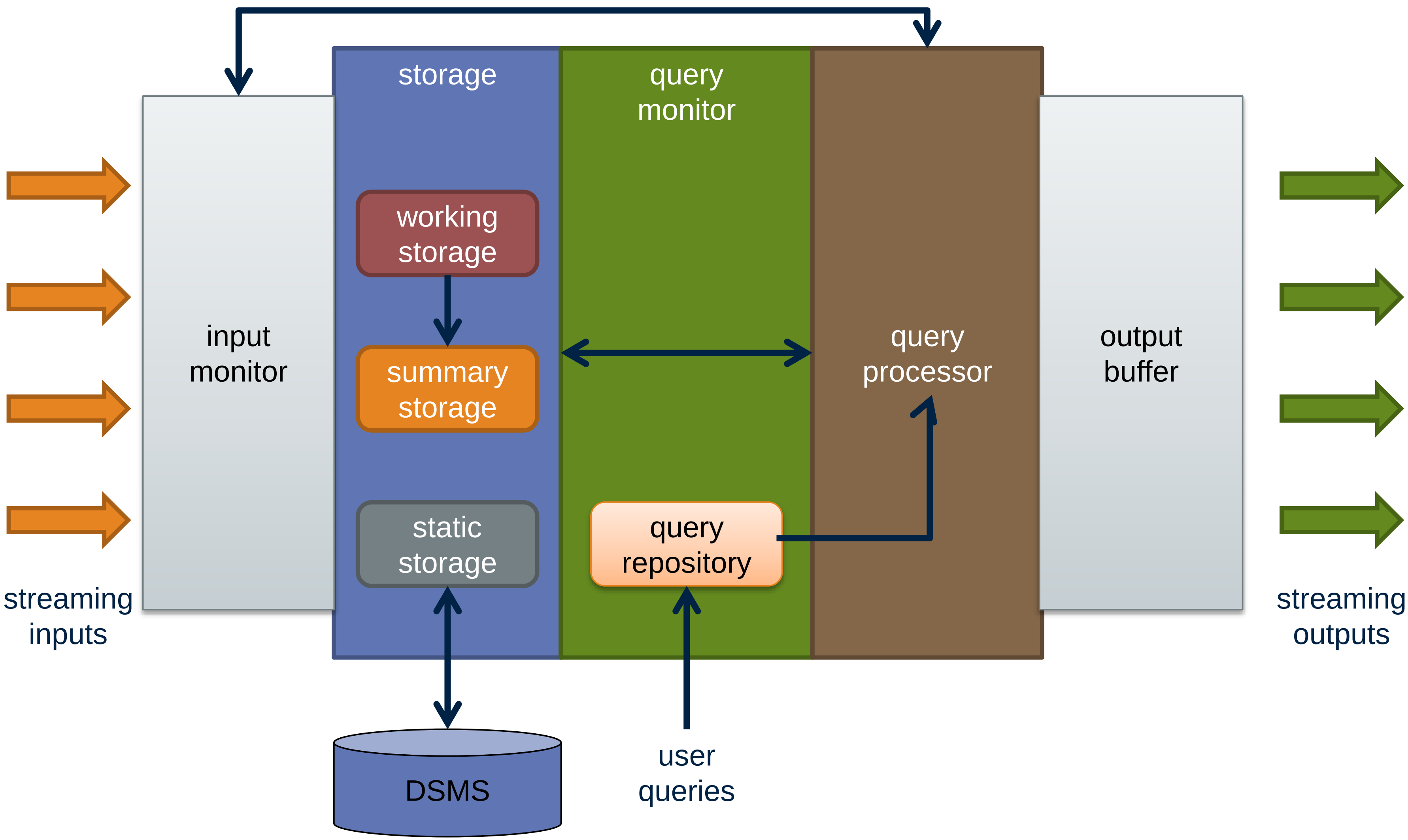
# Count-based windows

- Window of size *N* elements (sliding, tumbling) over the stream
- Problematic with non-unique timestamps associated with stream elements
- Ties broken arbitrarily may lead to non-deterministic output
- Potentially unpredictable with respect to fluctuating input rates
  - But dual of time based windows for constant arrival rates
  - Arrival rate $\lambda$ elements/time-unit, time-based window of length $T$, count-based window of size $N$; $N = \lambda T$

# Punctuation-based windows

- Application-inserted “end-of-processing”
  – Each next data item identifies “beginning-of-processing”
- Enables data item-dependent variable length windows
  – Examples: a stream of auctions, an interval of monitored activity
- Utility in data processing: limit the scope of operations relative to the stream
- Potentially problematic if windows grow too large
  – Or even too small: too many punctuations

# Putting it all together: architecting a DSMS



# STREAM MINING

# Data stream mining

- Numerous applications
  - Identify events and take responsive action in real time
  - Identify correlations in a stream and reconfigure system
- Mining query streams: Google wants to know what queries are more frequent today than yesterday
- Mining click streams: Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour
- Big brother
  - Who calls whom?
  - Who accesses which web pages?
  - Who buys what where?
  - All those questions answered in real time
- We will focus on frequent pattern mining

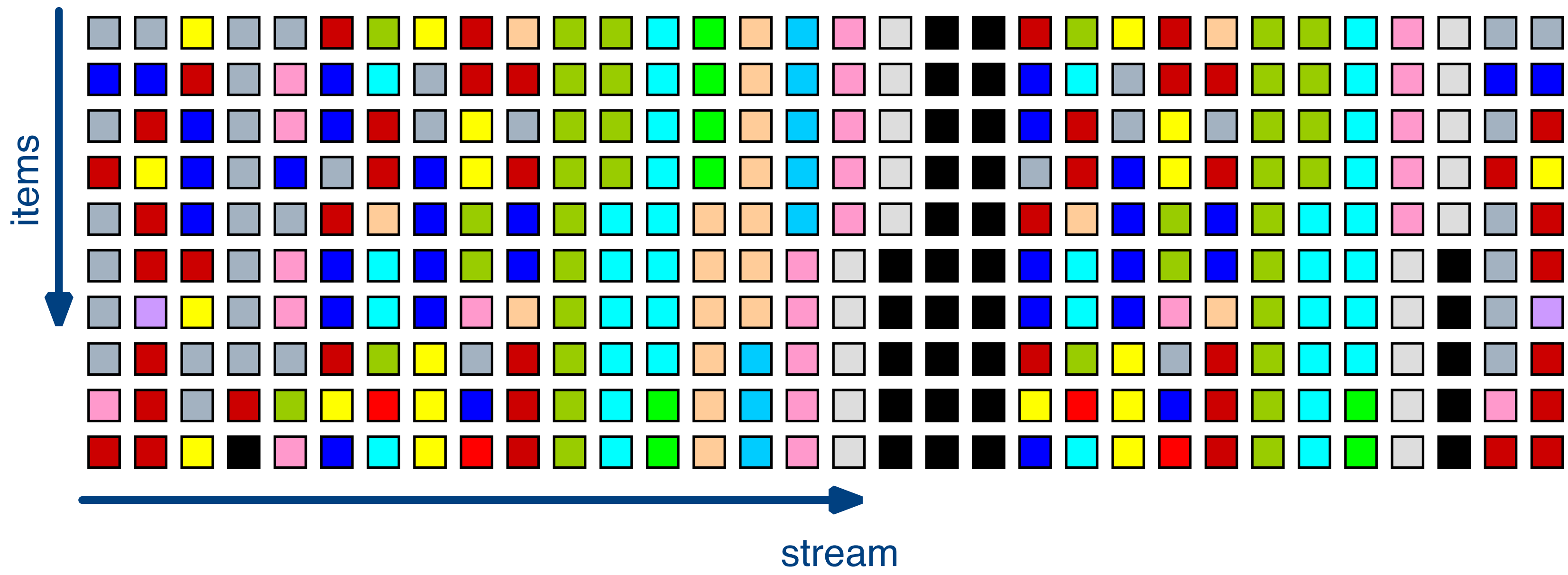
# Frequent pattern mining

- Frequent pattern mining refers to finding patterns that occur more frequently than a pre-specified threshold value
  - Patterns refer to items, itemsets, or sequences
  - Threshold refers to the percentage of the pattern occurrences to the total number of transactions
    - Termed as support
- Finding frequent patterns is the first step for association rules
  - $A \rightarrow B$: $A$ implies $B$
- Many metrics have been proposed for measuring how strong an association rule is
  - Most commonly used metric: confidence
  - Confidence refers to the probability that set $B$ exists given that $A$ already exists in a transaction
    - $\text{confidence}(A \rightarrow B) = \text{support}(A \wedge B) / \text{support}(A)$

# Frequent pattern mining in data streams

- Frequent pattern mining over data streams differs from conventional one
  - Cannot afford multiple passes
    - Minimised requirements in terms of memory
    - Trade off between storage, complexity, and accuracy
    - You only get one look
- Frequent items (also known as heavy hitters) and itemsets are usually the final output
- Effectively a counting problem
  - We will focus on two algorithms: lossy counting and sticky sampling

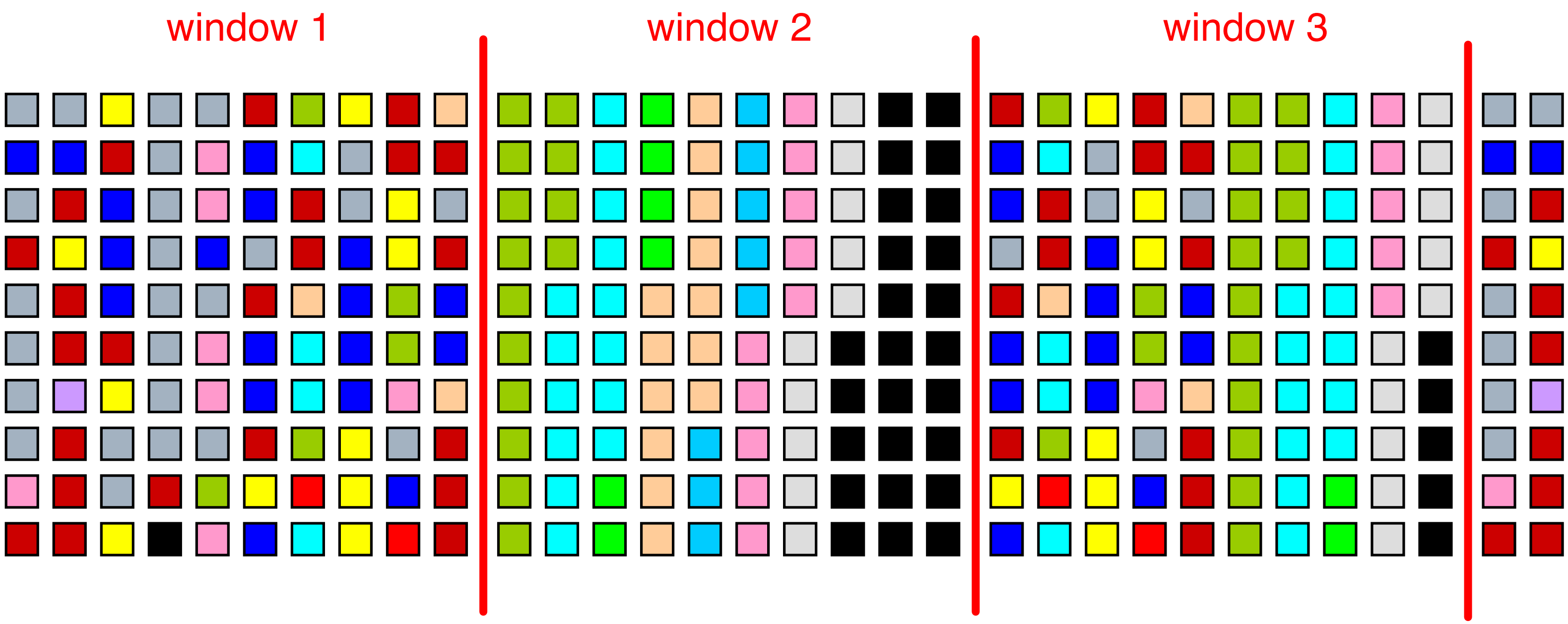
# The problem in more detail



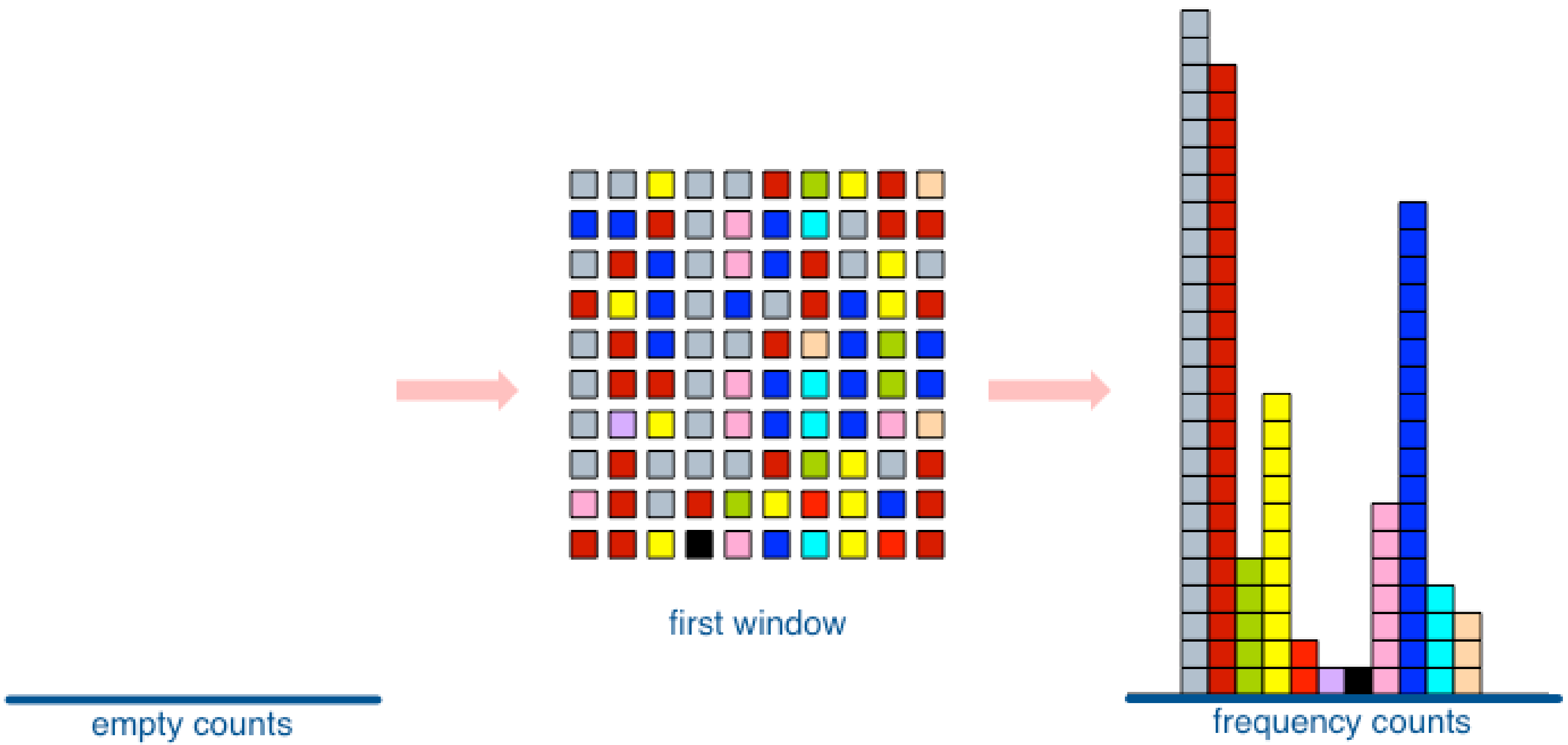

- Problem statement
  - Identify all items whose current frequency exceeds some support threshold *s* (*e.g.*, 0.1%)

# Lossy counting in action

- Divide the incoming stream into windows

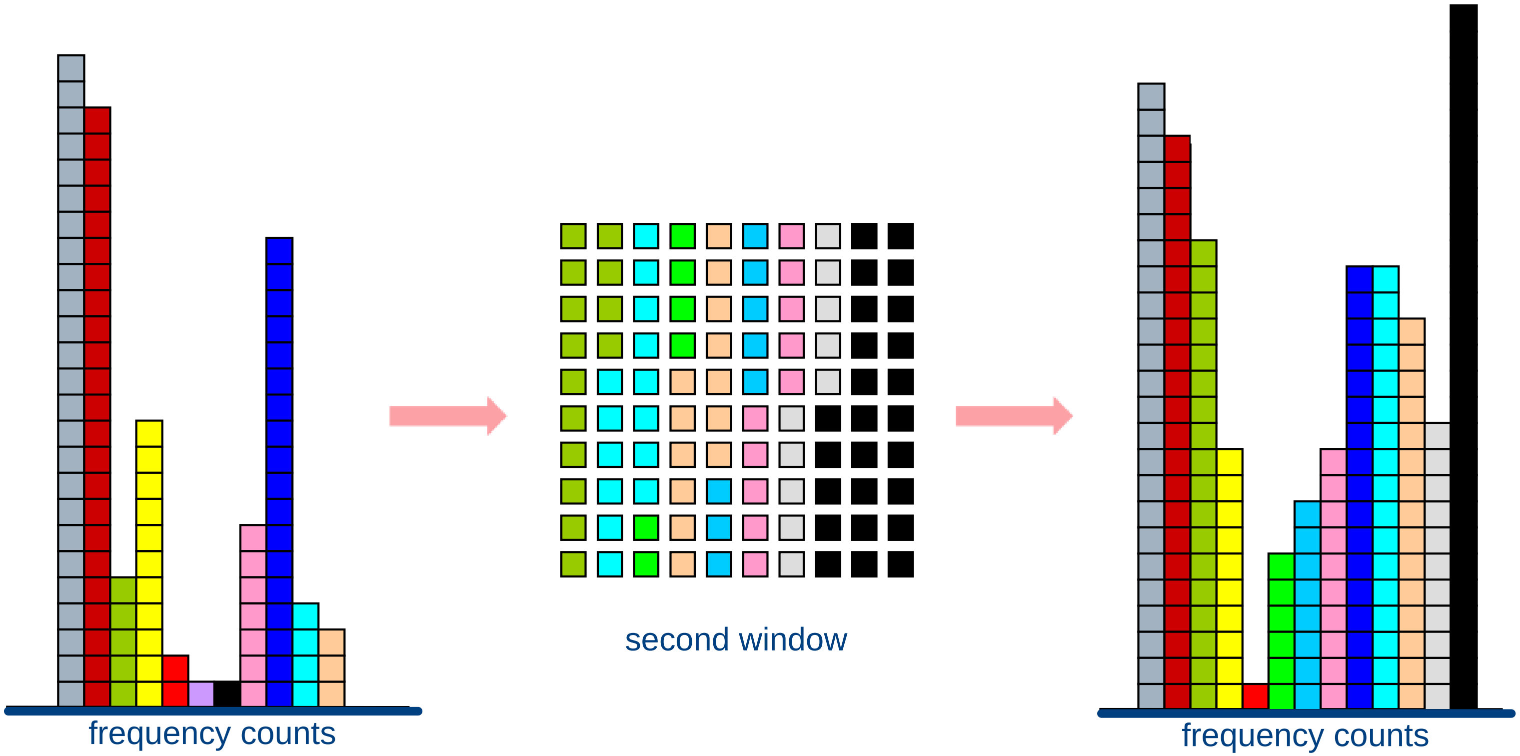


# First window comes in




- At window boundary, adjust counters

# Next window comes in




- At window boundary, adjust counters

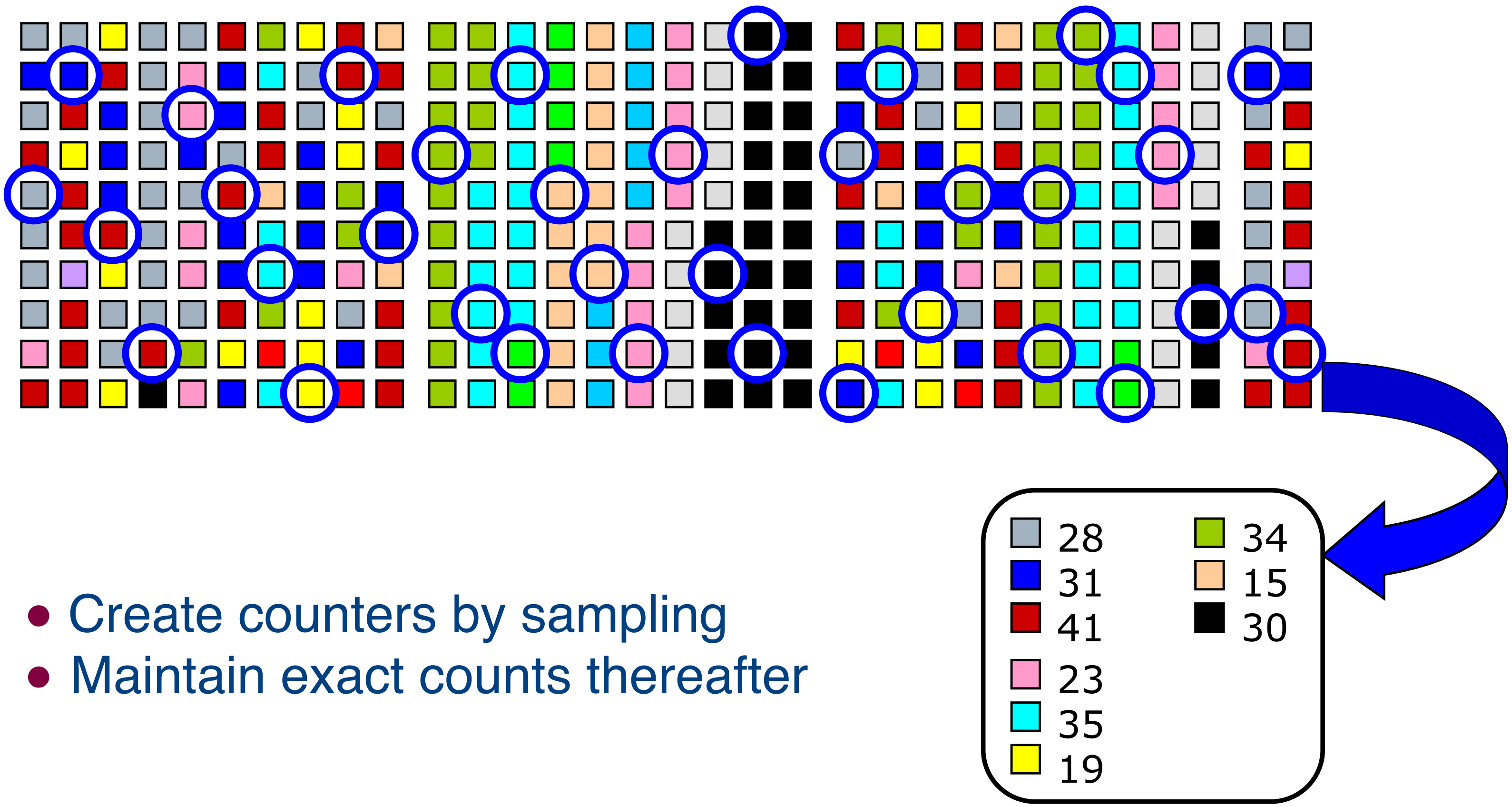
# Lossy counting algorithm

- Deterministic technique; user supplies two parameters
  - Support $s$; error $\varepsilon$
- Simple data structure, maintaining triplets of data items $e$, their associated frequencies $f$, and the maximum possible error $\Delta$ in $f$ : ($e$, $f$, $\Delta$)
- The stream is conceptually divided into buckets of width $w = 1/\varepsilon$
  - Each bucket labelled by a value $N/w$ where $N$ starts from 1 and increases by 1
- For each incoming item, the data structure is checked
  - If an entry exists, increment frequency
  - Otherwise, add new entry with $\Delta = b_{current} - 1$ where $b_{current}$ is the current bucket label
- When switching to a new bucket, all entries with $f + \Delta < b_{current}$ are released

# Lossy counting observations

- How much do we undercount?
  - If current size of stream is $N$
  - ...and window size is $1/\varepsilon$
  - ...then frequency error ≤ number of windows, *i*.e., $\varepsilon N$
- Empirical rule of thumb: set $\varepsilon$ = 10% of support $s$
  - Example: given a support frequency $s$ = 1%,
  - …then set error frequency $\varepsilon$ = 0.1%
- Output is elements with counter values exceeding $sN - \varepsilon N$
- Guarantees
  - Frequencies are underestimated by at most $\varepsilon N$
  - No false negatives
  - False positives have true frequency at least $sN-\varepsilon N$
- In the worst case, it has been proven that we need $1/\varepsilon \times \log(\varepsilon N)$ counters

# Sticky Sampling




- Create counters by sampling
- Maintain exact counts thereafter

Centrum Wiskunde & Informatica

# STORM AND LOW-LATENCY PROCESSING

# Low latency processing

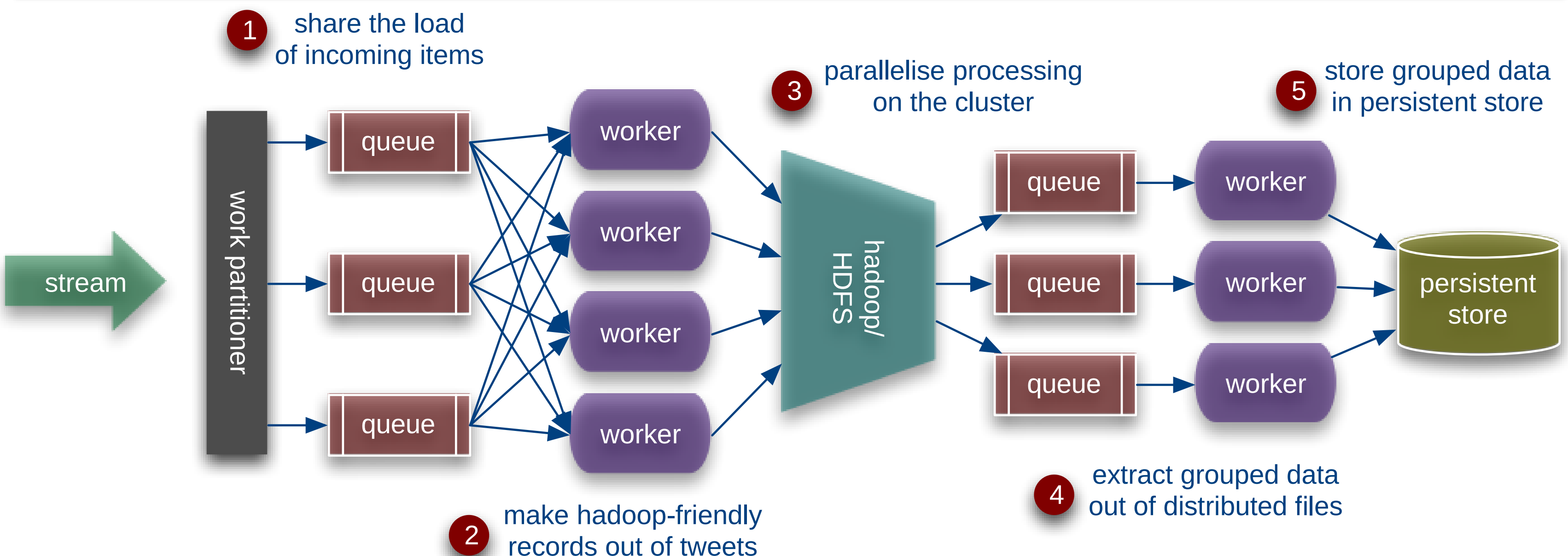
- Similar to data stream processing, but with a twist
  - Data is streaming into the system (from a database, or a network stream, or an HDFS file, or …)
  - We want to process the stream in a distributed fashion
  - And we want results as quickly as possible
- Numerous applications
  - Algorithmic trading: identify financial opportunities (e.g., respond as quickly as possible to stock price rising/falling
  - Event detection: identify changes in behaviour rapidly
- Not (necessarily) the same as what we have seen so far
  - The focus is not on summarising the input
  - Rather, it is on “parsing” the input and/or manipulating it on the fly

# The problem

- Consider the following use-case
- A stream of incoming information needs to be summarised by some identifying token
  - For instance, group tweets by hash-tag; or, group clicks by URL;
  - And maintain accurate counts
- But do that at a massive scale and in real time
- Not so much about handling the incoming load, but using it
  - That's where latency comes into play
- Putting things in perspective
  - Twitter's load is not that high: at 15k tweets/s and at 150 bytes/tweet we're talking about 2.25MB/s
  - Google served 34k searches/s in 2010: let's say 100k searches/s now and an average of 200 bytes/search that's 20MB/s
  - But this 20MB/s needs to filter PBs of data in less than 0.1s; that's an EB/s throughput

# A rough approach

- Latency
  - Each point 1 − 5 in the figure introduces a high processing latency
  - Need a way to transparently use the cluster to process the stream




- Bottlenecks
  - No notion of locality
    - Either a queue per worker per node, or data is moved around
  - What about reconfiguration?
    - If there are bursts in traffic we need to shutdown, reconfigure and redeploy

# Storm

- Started up as backtype; widely used in Twitter
- Open-sourced (you can download it and play with it!
  - http://storm-project.net/
- On the surface, Hadoop for data streams
  - Executes on top of a (likely dedicated) cluster of commodity hardware
  - Similar setup to a Hadoop cluster
    - Master node, distributed coordination, worker nodes
    - We will examine each in detail
- But whereas a MapReduce job will finish, a Storm job—termed a topology—runs continuously
  - Or rather, until you kill it

# Storm topologies

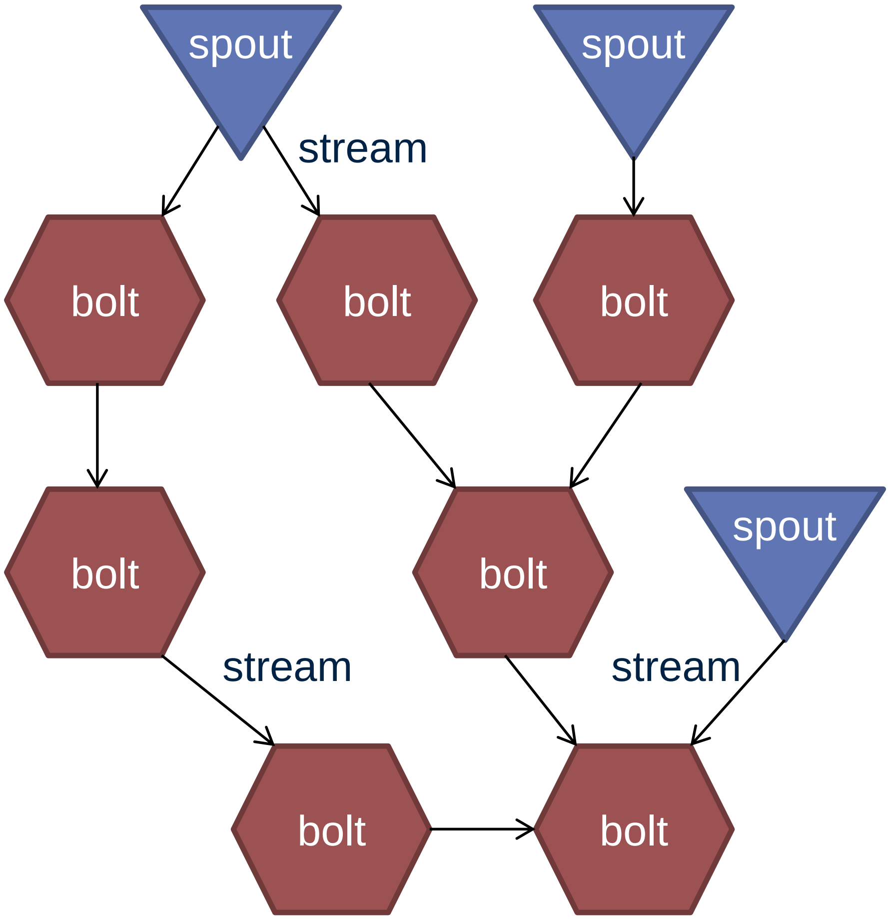
- A Storm topology is a graph of computation
  - Graph contains nodes and edges
  - Nodes model processing logic (i.e., transformation over its input)
  - Directed edges indicate communication between nodes
  - No limitations on the topology; for instance one node may have more than one incoming edges and more than one outgoing edges
- Storm processes topologies in a distributed and reliable fashion

# Streams, spouts, and bolts

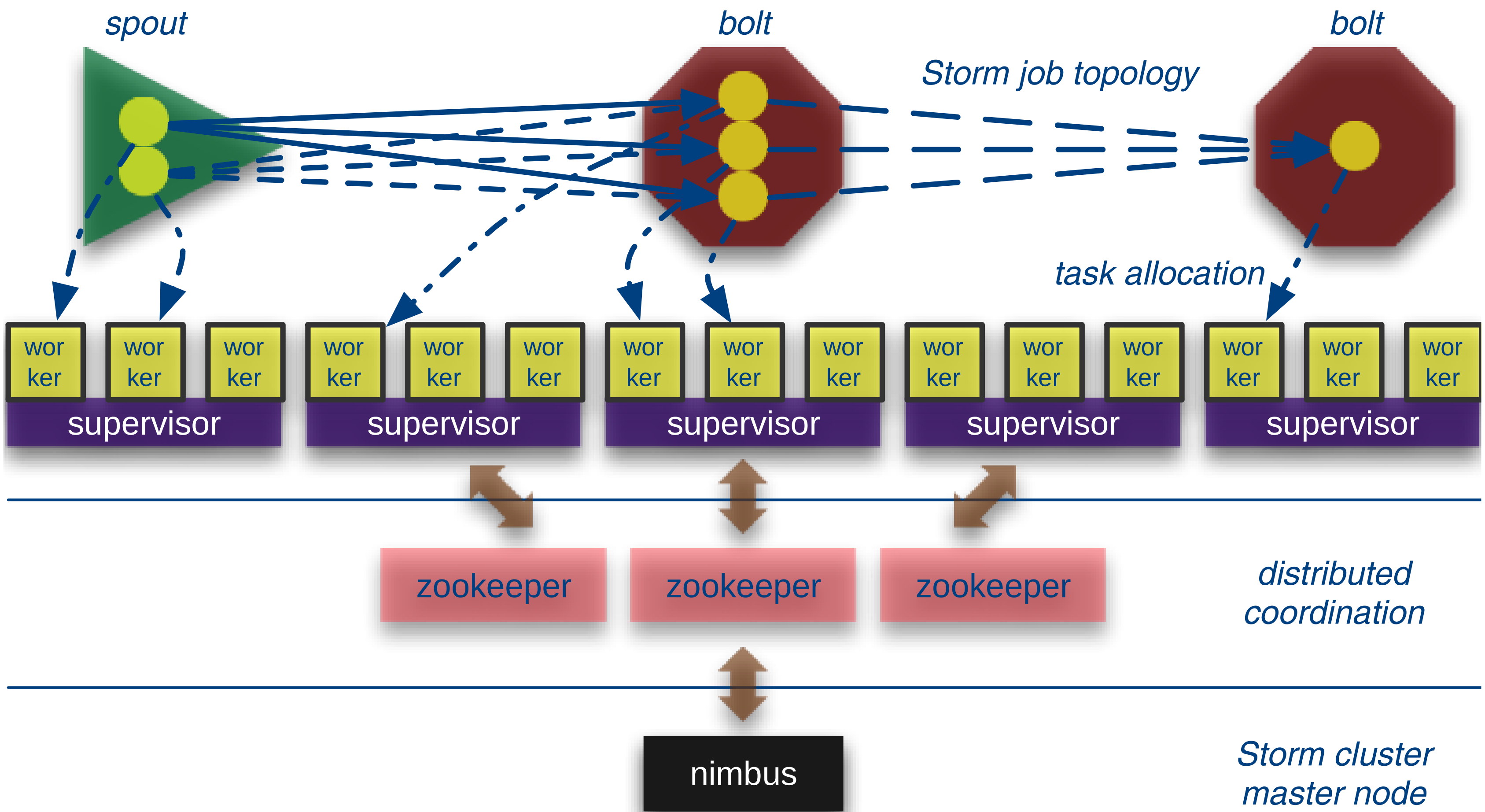
- Streams
  - The basic collection abstraction: an unbounded sequence of tuples
  - Streams are transformed by the processing elements of a topology

- Spouts
  - Stream generators
  - May propagate a single stream to multiple consumers

- Bolts
  - Subscribe to streams
  - Streams transformers
  - Process incoming streams and produce new ones

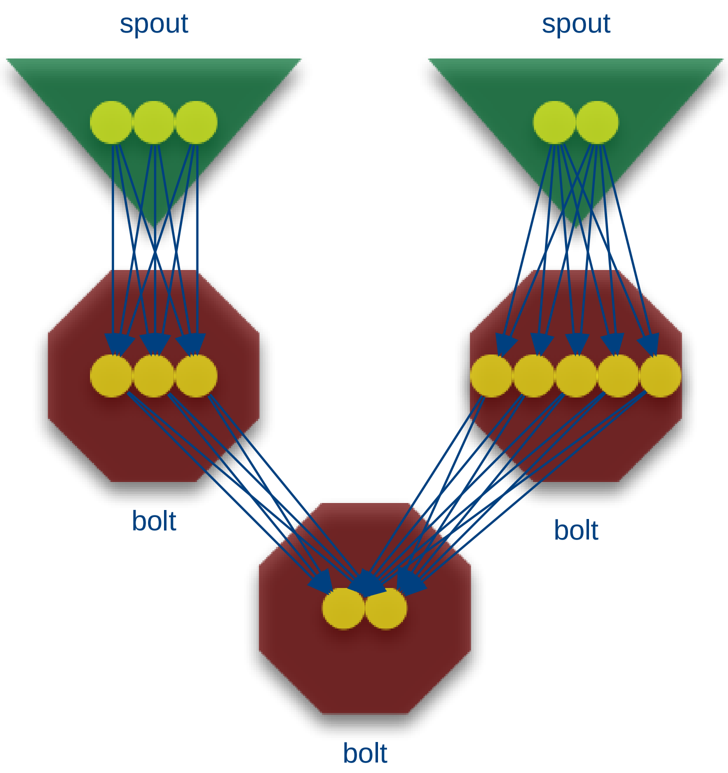


# Storm architecture



# From topology to processing: stream groupings

- Spouts and bolts are replicated in taks, each task executed in parallel by a worker
  - User-defined degree of replication
  - All pairwise combinations are possible between tasks
- When a task emits a tuple, which task should it send to?
- Stream groupings dictate how to propagate tuples
  - Shuffle grouping: round-robin
  - Field grouping: based on the data value (e.g., range partitioning)



# Putting it all together: word count

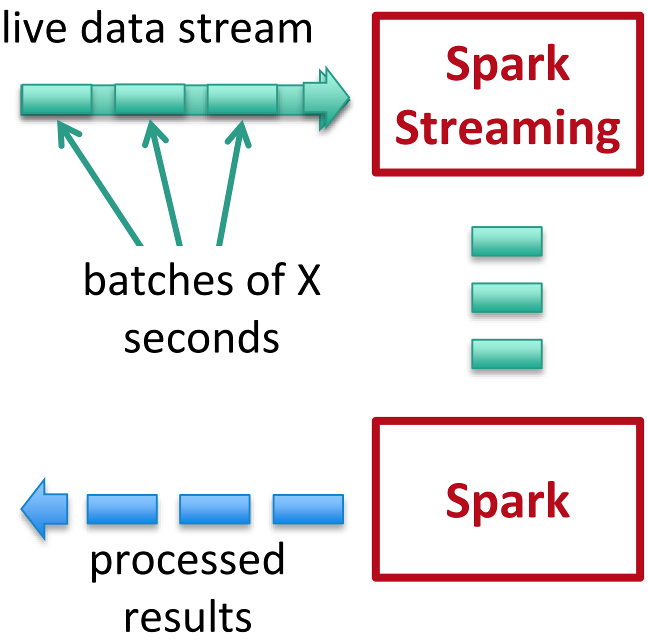
```
// instantiate a new topology
TopologyBuilder builder = new TopologyBuilder();
// set up a new spout with five tasks
builder.setSpout("spout", new RandomSentenceSpout(), 5);
// the sentence splitter bolt with eight tasks
builder.setBolt("split", new SplitSentence(), 8)
    .shuffleGrouping("spout"); // shuffle grouping for the ouput
// word counter with twelve tasks
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("split", new Fields("word")); // field grouping
// new configuration
Config conf = new Config();
// set the number of workers for the topology; the 5x8x12=480 tasks
// will be allocated round-robin to the three workers, each task
// running as a separate thread
conf.setNumWorkers(3);
// submit the topology to the cluster
StormSubmitter.submitTopology("word-count", conf, builder.createTopology());
```

# SPARK STREAMING

# Discretized Stream Processing

Run a streaming computation as a series of very small, deterministic batch jobs → "MICRO BATCH" approach

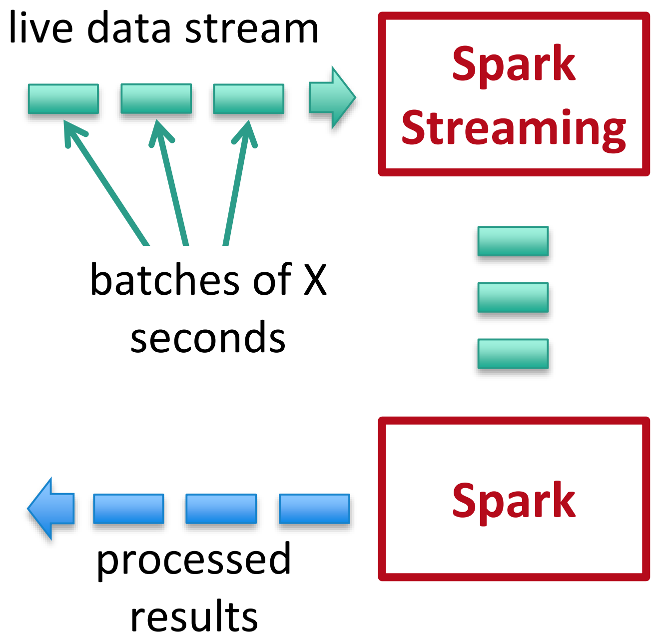
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



# Discretized Stream Processing

Run a streaming computation as a series of very small, deterministic batch jobs → "MICRO BATCH" approach

- Batch sizes as low as ½ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system



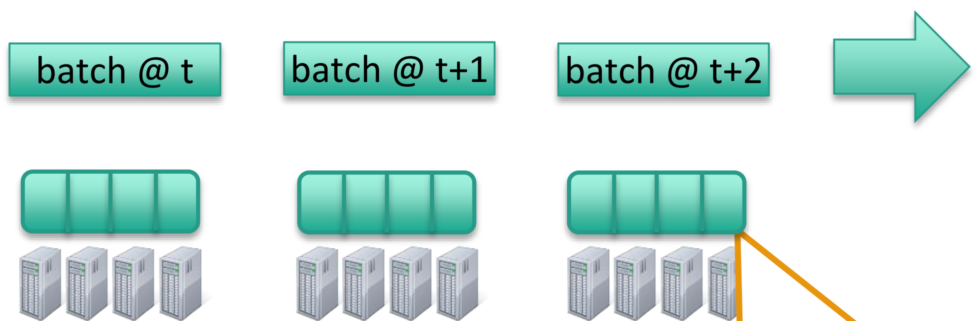
# Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
```

**DStream**: a sequence of RDDs representing a stream of data

Twitter Streaming API

tweets DStream



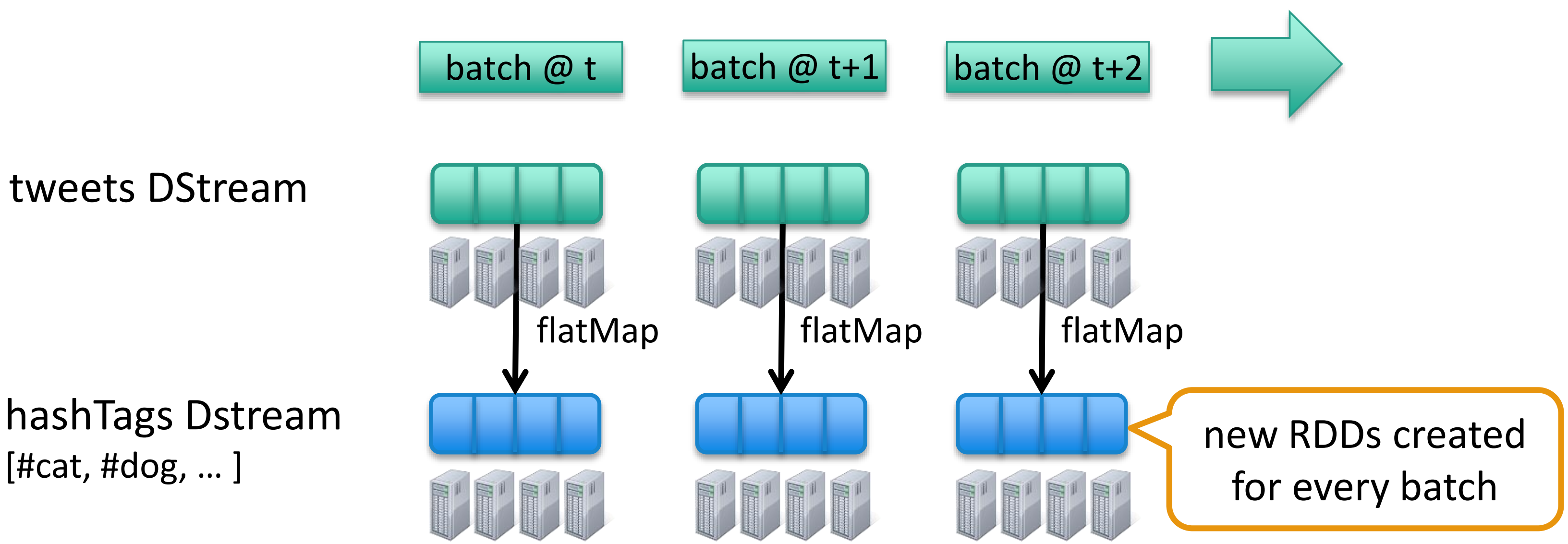

stored in memory as an RDD
(immutable, distributed)

# Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

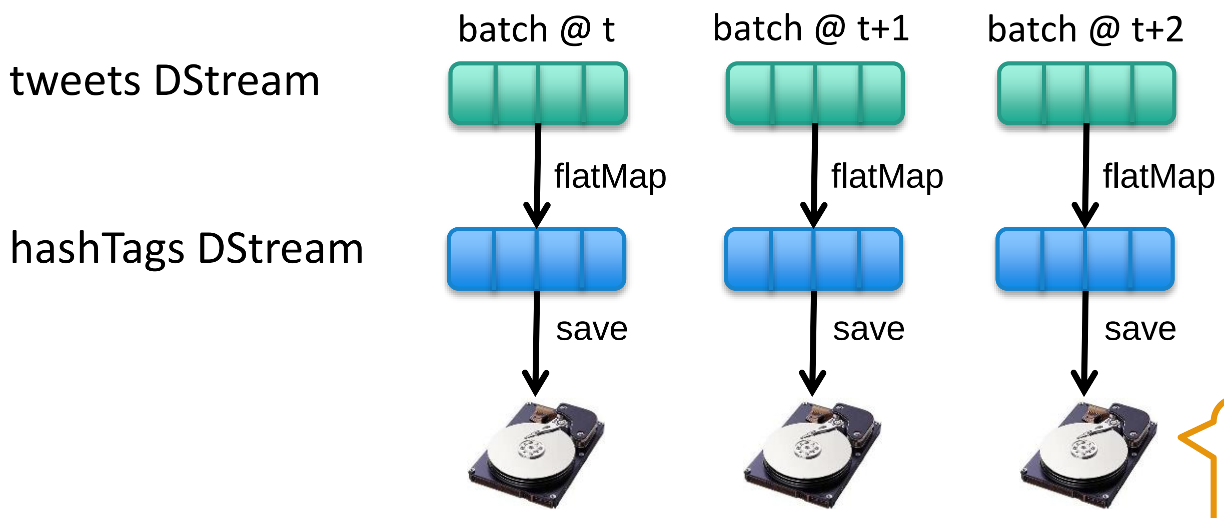
**transformation**: modify data in one DStream to create another DStream



# Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

**output operation**: to push data to external storage

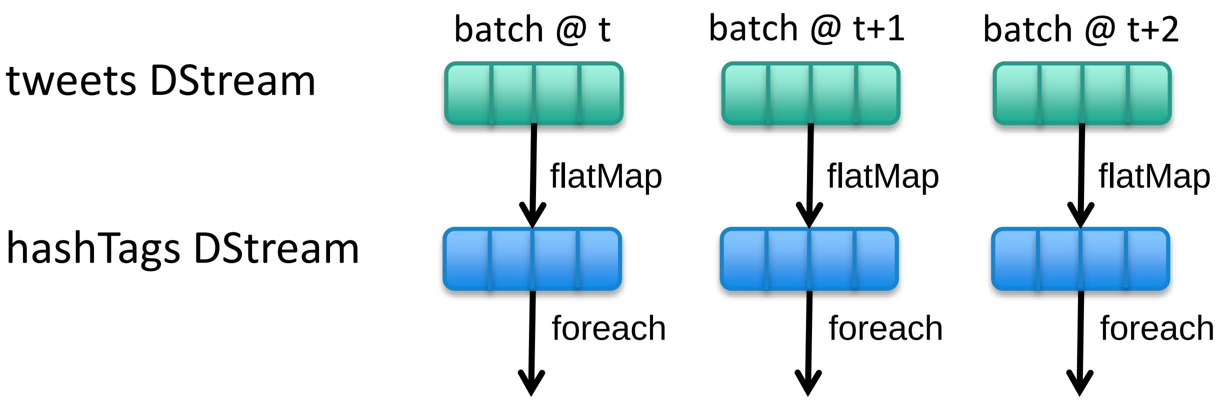



every batch saved to HDFS

# Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.foreach(hashTagRDD => { ... })
```

**foreach**: do whatever you want with the processed data




Write to database, update analytics UI, do whatever you want

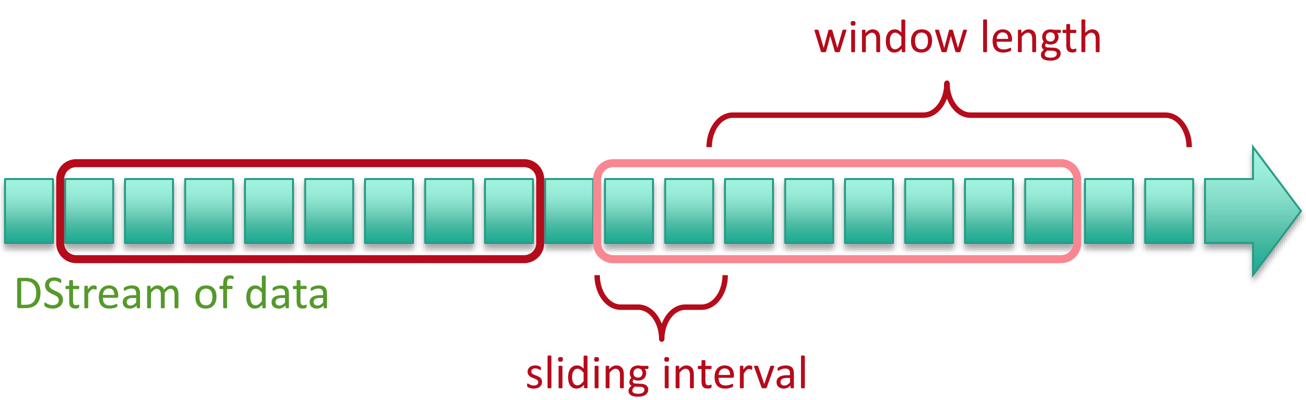
# Window-based Transformations

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
```

sliding window operation

window length

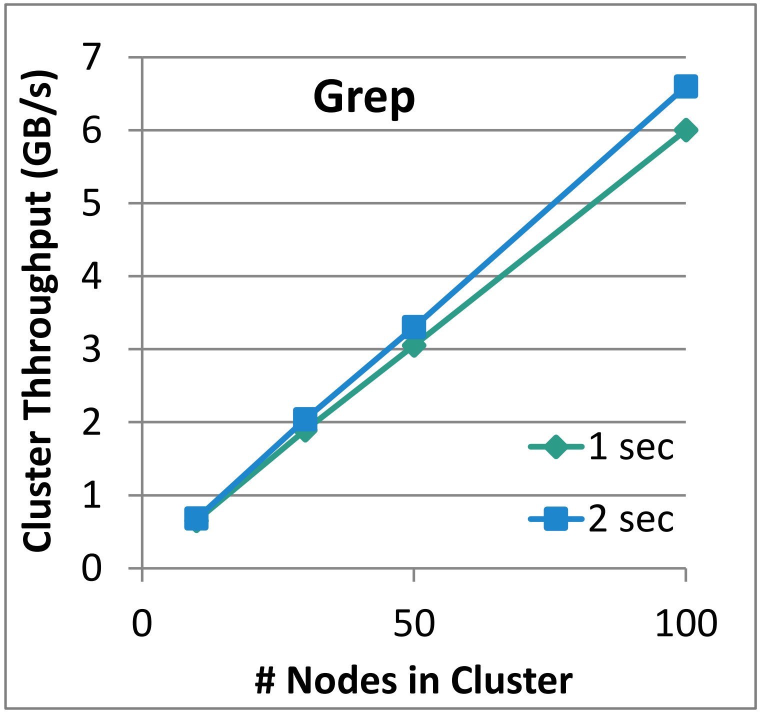
sliding interval

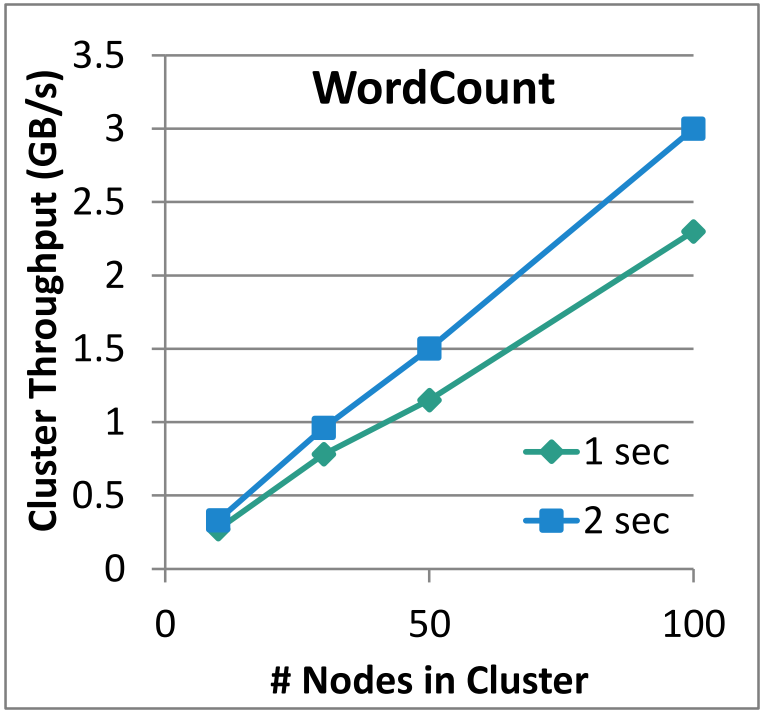


# Performance

Can process **6 GB/sec (60M records/sec)** of data on 100 nodes at **sub-second** latency

- Tested with 100 text streams on 100 EC2 instances with 4 cores each

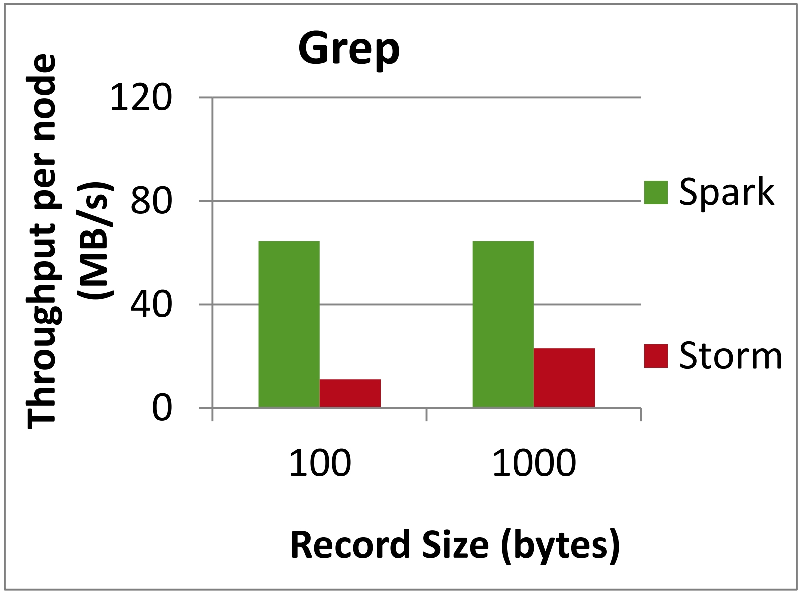


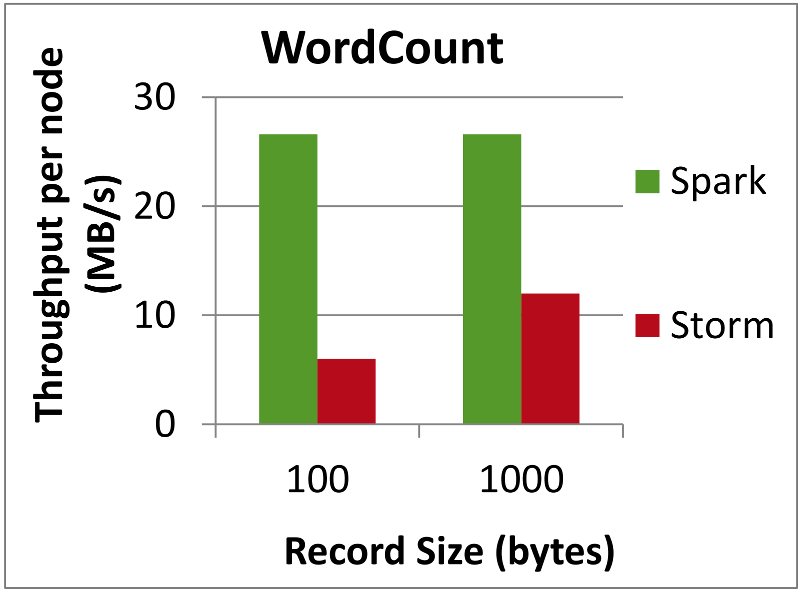



# Comparison with Storm and S4

Higher throughput than Storm

- Spark Streaming: **670k** records/second/node
- Storm: **115k** records/second/node
- Apache S4: 7.5k records/second/node



# Unifying Batch and Stream Processing Models

Spark program on Twitter log file using RDDs

```
val tweets = sc.hadoopFile("hdfs://...")
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFile("hdfs://...")
```

Spark Streaming program on Twitter stream using DStreams

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

# Vision - *one stack to rule them all*

- Explore data interactively using Spark Shell to identify problems

```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = filtered.map(...)
...
```

- Use same code in Spark stand-alone programs to identify problems in production logs

```
object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}
```

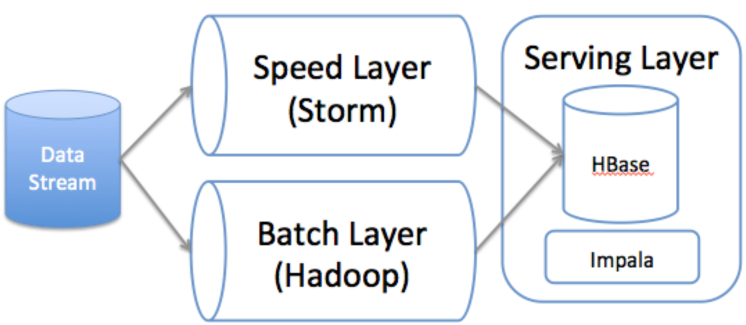
- Use similar code in Spark Streaming to identify problems in live log streams

```
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}
```

# LAMBDA ARCHITECTURE

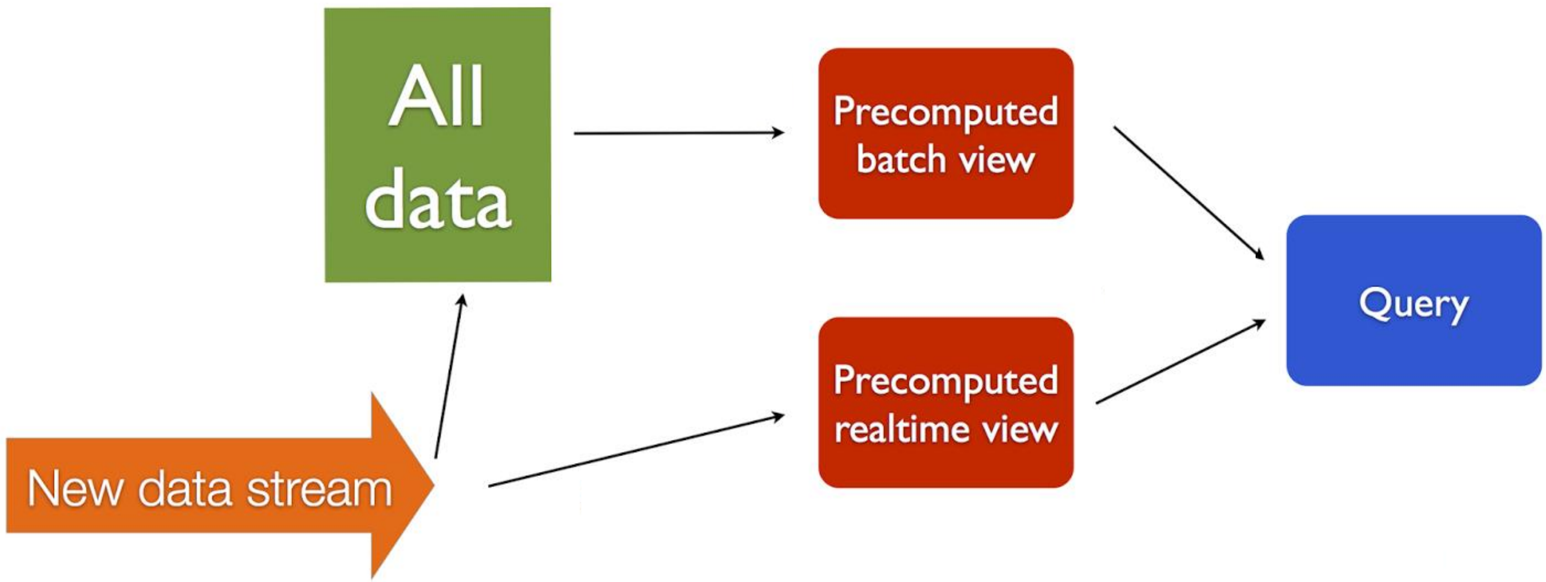
# Lambda Architecture

- apply the (λ) Lambda philosophy in designing big data system
- equation "query = function(all data)" which is the basis of all data systems
- proposed by Nathan Marz (http://nathanmarz.com/)
  - software engineer from Twitter in his "Big Data" book.
- three design principles:
  1. human fault-tolerance – the system is unsusceptible to data loss or data corruption because at scale it could be irreparable.
  2. data immutability – store data in it's rawest form immutable and for perpetuity.
     - INSERT/ SELECT/DELETE but no UPDATE !)
  3. recomputation – with the two principles above it is always possible to (re)-compute results by running a function on the raw data




*D'Oh! Two pipelines!*

# Lambda Architecture




"Lambda Architecture"

# GOOGLE DATAFLOW

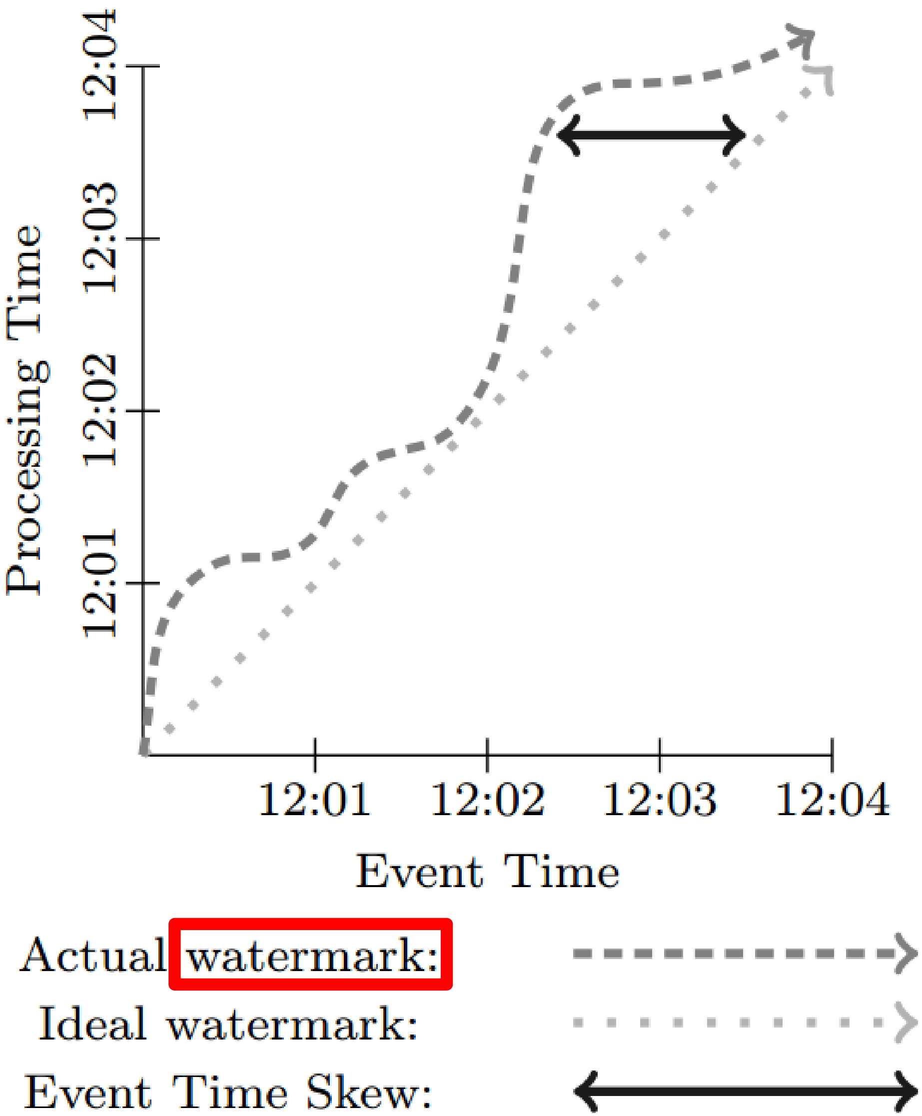
# Google DataFlow

- Allows for the calculation of
  - event-time ordered results,
  - windowed by features of the data themselves,
  - over an **unbounded**, unordered data source,
  - correctness, latency, and cost **tunable** across a broad spectrum of combinations.

- Decomposes pipeline implementation across four related dimensions, providing clarity, composability, and flexibility:
  - **What** results are being computed.
  - **Where** in event time they are being computed.
  - **When** in processing time they are materialized.
  - **How** earlier results relate to later refinements.

- Separates the **logical data processing** from the underlying **physical implementation**,
  - allowing the choice of
    - batch
    - micro-batch, or
    - streaming engine to become one of simply correctness, latency, and cost.

# DataFlow: Time




Two kinds of time

- **Event Time**, which is the time at which the event itself actually occurred
- **Processing Time**, which is the time at which an event is handled by the processing pipeline.

**watermark = time before which the system (thinks it) has processed all events**

# DataFlow: Processing Model

Generalized MapReduce:

- **ParDo (doFcn)** pretty much = **"Map"**
  - Each input element to be processed (which itself may be a finite collection) is provided to a user-defined function (called a DoFn in Dataflow), which can yield zero or more output elements per input.
  - For example, consider an operation which expands all prefixes of the input key, duplicating the value across them:
    - Input: (fix, 1),(fit, 2) · · ·

      ➔ ParDo(ExpandPrefixes) ➔
    - Output: (f, 1),(fi, 1),(fix, 1),(f, 2),(fi, 2),(fit, 2)
- **GroupByKey** more or less ~ **"Reduce"**
  - for key-grouping (key, value) pairs.
  - In the example:
    - Input: (f, 1),(fi, 1),(fix, 1),(f, 2),(fi, 2),(fit, 2)

      ➔ GroupByKey ➔
    - Output: (f, [1, 2]),(fi, [1, 2]),(fix, [1]),(fit, [2])

# DataFlow: Windowing Model

Many possible window definitions, define one using two methods:

- **AssignWindows**(**T datum**) ➔ Set<Windows>
- **MergeWindows**(**Set<Windows>**) ➔ Set<Windows>

Example:

- Input: (k, v1, 12:00, [0, ∞)),(k, v2, 12:01, [0, ∞)) · · ·

➔ AssignWindows( Sliding(2min, 1min)) ➔

- Output:

  (k, v1, 12:00, [11:59, 12:01)),

  (k, v1, 12:00, [12:00, 12:02)),

  (k, v2, 12:01, [12:00, 12:02)),

  (k, v2, 12:01, [12:01, 12:03))

# Data Model

- MapReduce

  (Key,Value)

- DataFlow

  (Key, Value, EventTime, Window)

Centrum Wiskunde & Informatica

# DataFlow: Windowing Model

```
PCollection<KV<String, Integer>> input = IO.read(...);
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(Sessions.withGapDuration(
      Duration.standardMinutes(30))))
  .apply(Sum.integersPerKey());
```

AssignWindows( Sliding(2m, 1m))

- Output:

    (k, v1, 12:00, [11:59, 12:01)),

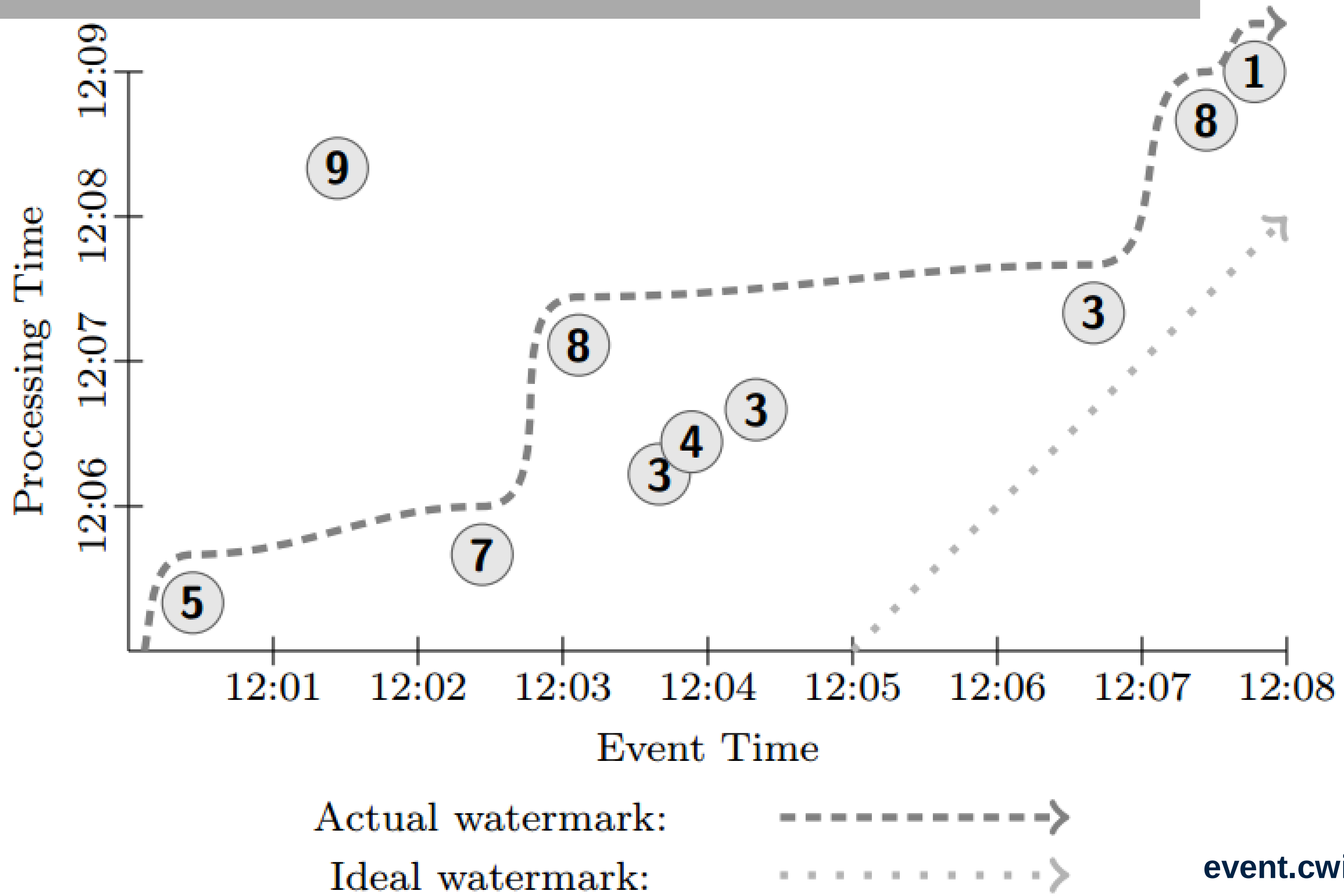
    (k, v1, 12:00, [12:00, 12:02)),

    (k, v2, 12:01, [12:00, 12:02)),

    (k, v2, 12:01, [12:01, 12:03))

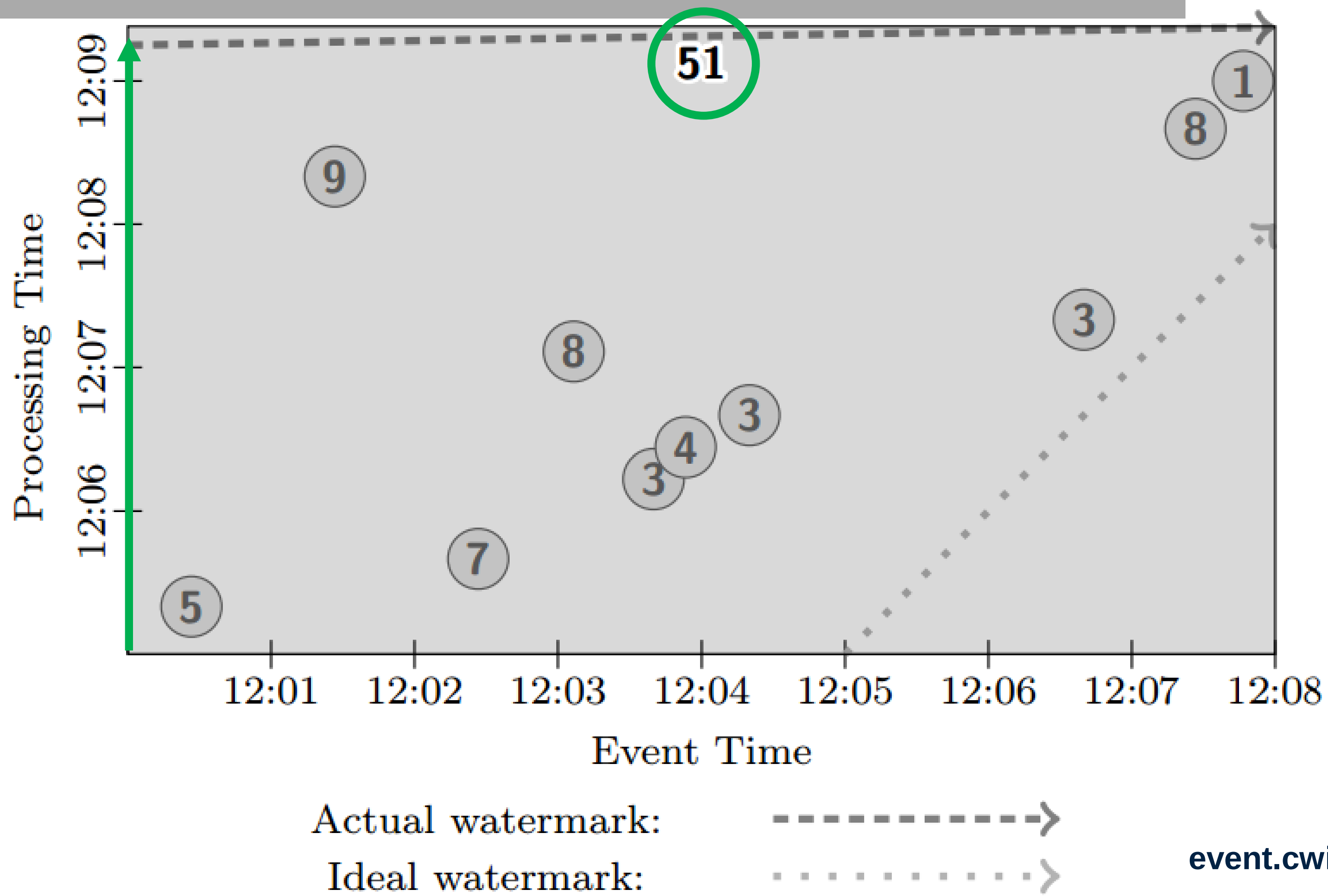
# Example. When do results get computed?

```
PCollection<KV<String, Integer>> output = input
  .apply(Sum.integersPerKey());
```



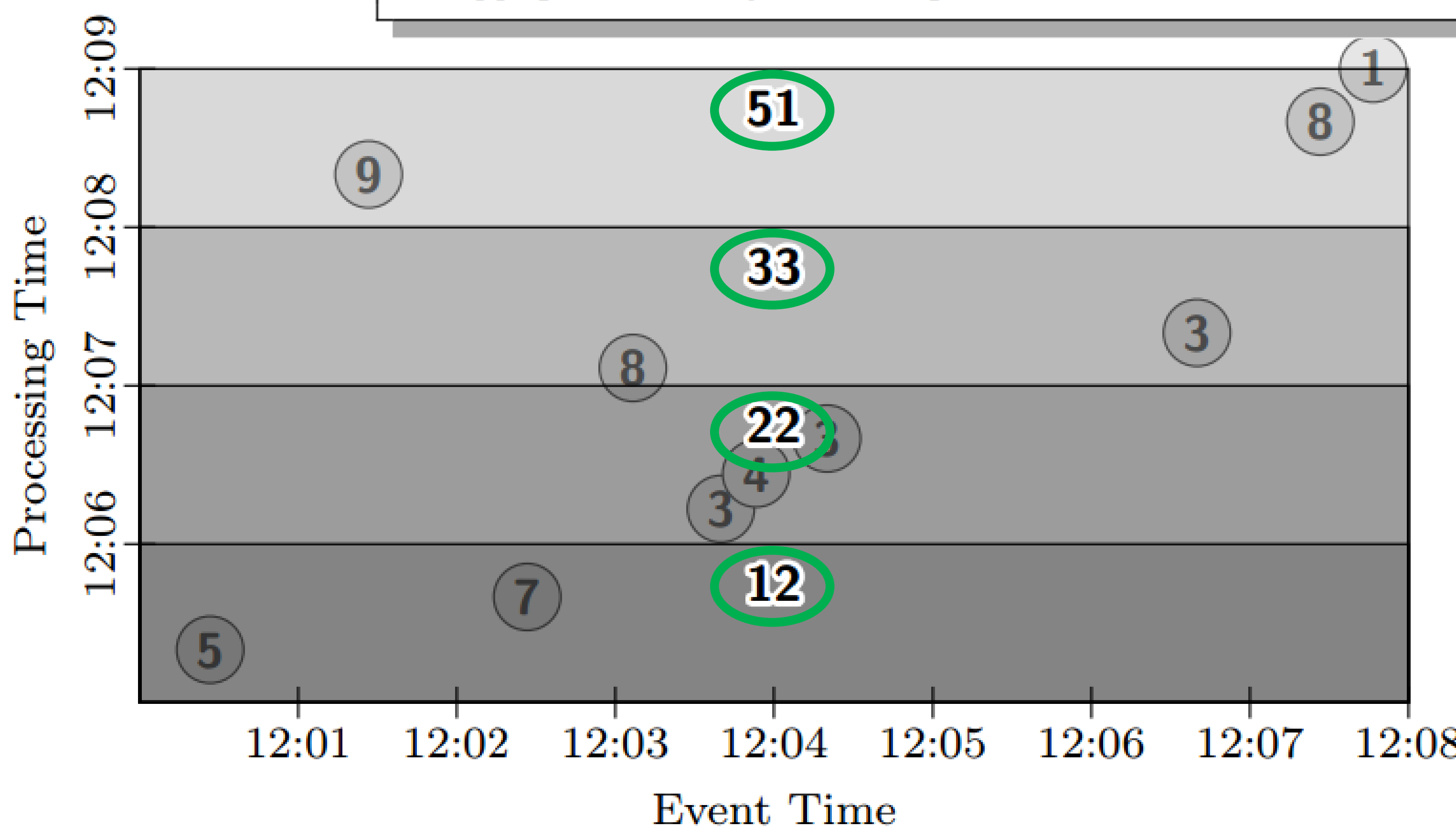
# Triggering: classical batch execution

```
PCollection<KV<String, Integer>> output = input
  .apply(Sum.integersPerKey());
```



# GlobalWindows, AtPeriod, Accumulating

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
               .accumulating())
  .apply(Sum.integersPerKey());
```

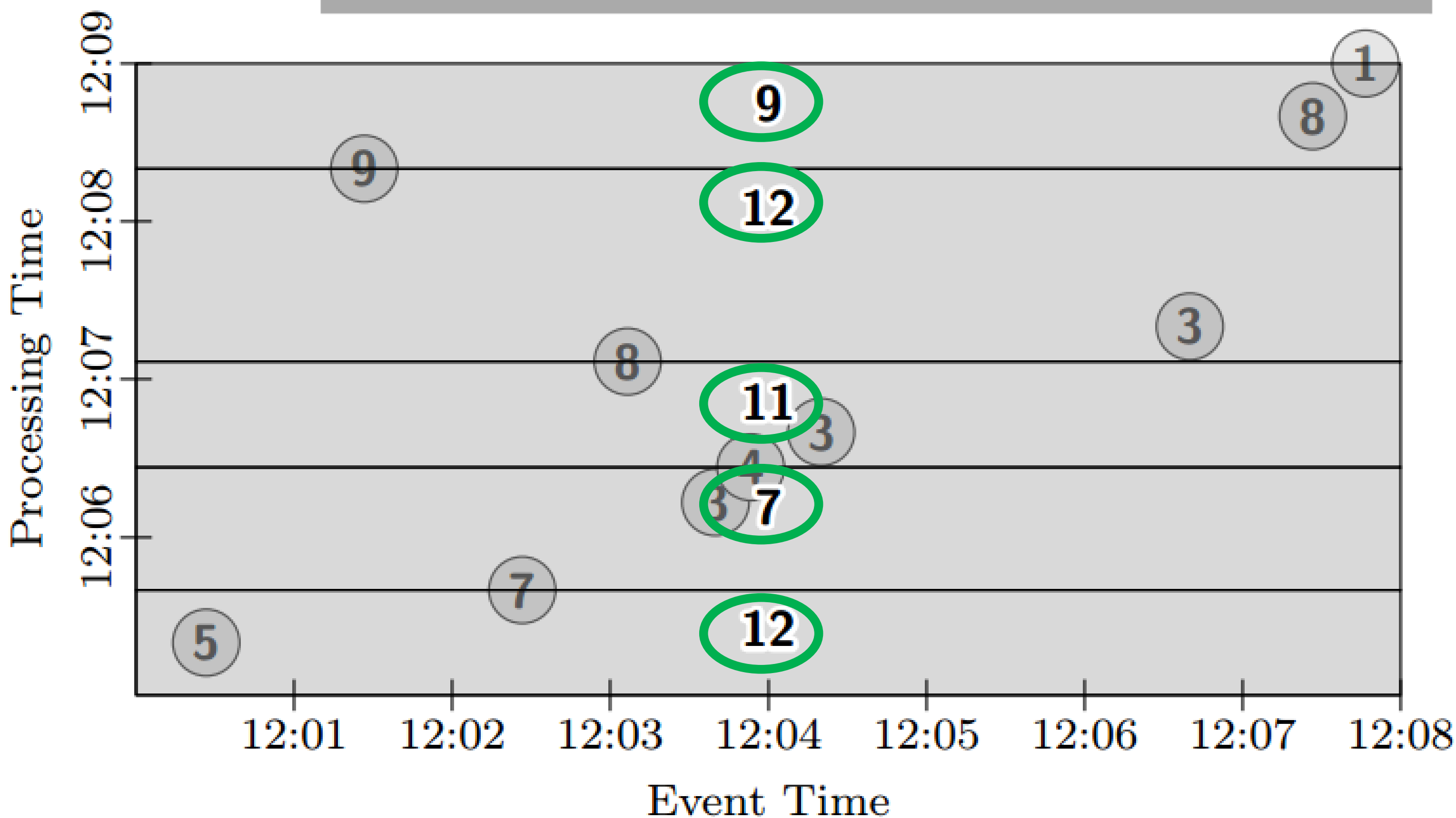


CWI

Centrum Wiskunde & Informatica

# GlobalWindows, AtCount, Discarding

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtCount(2)))
                  .discarding())
  .apply(Sum.integersPerKey());
```

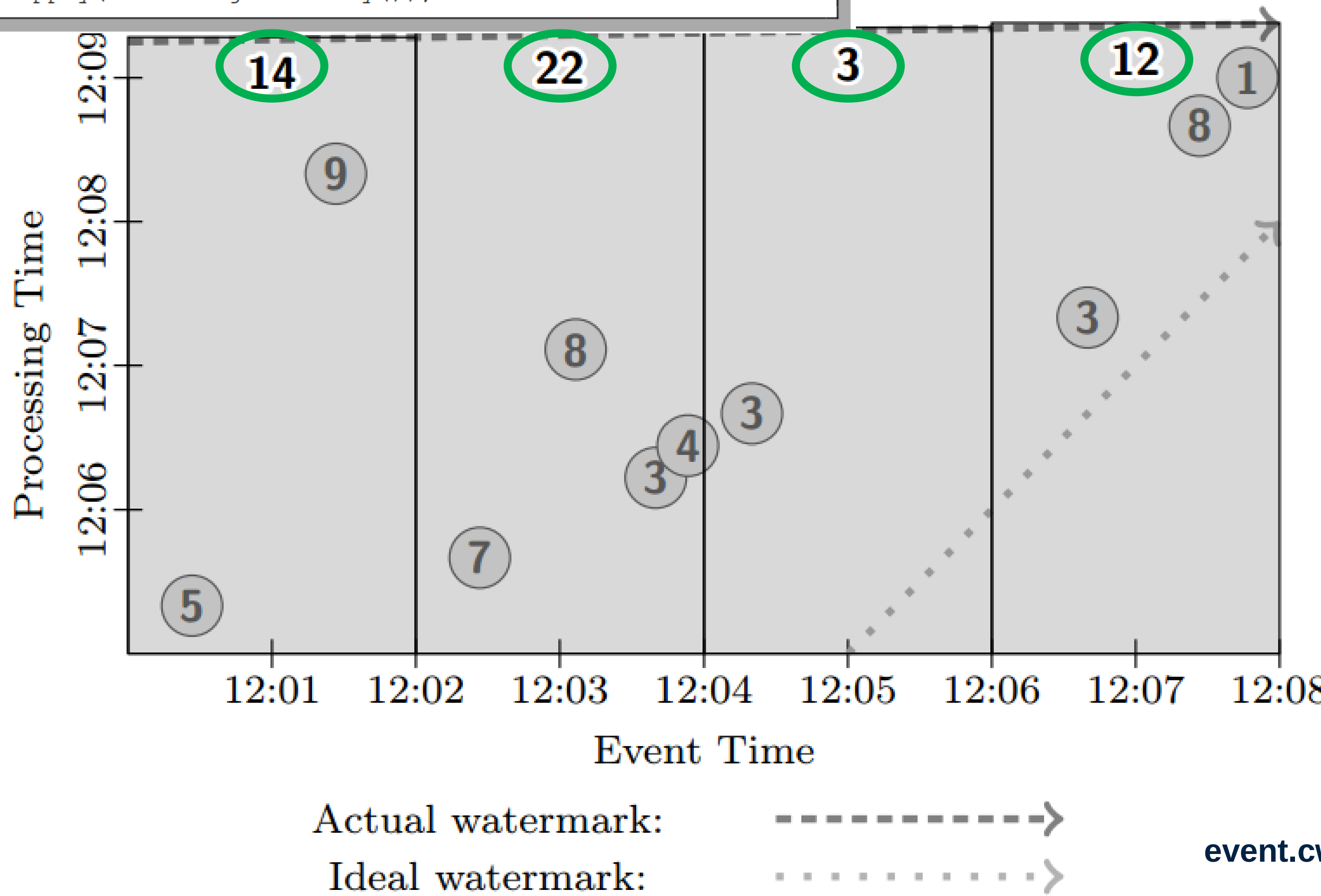


CWI

Centrum Wiskunde & Informatica

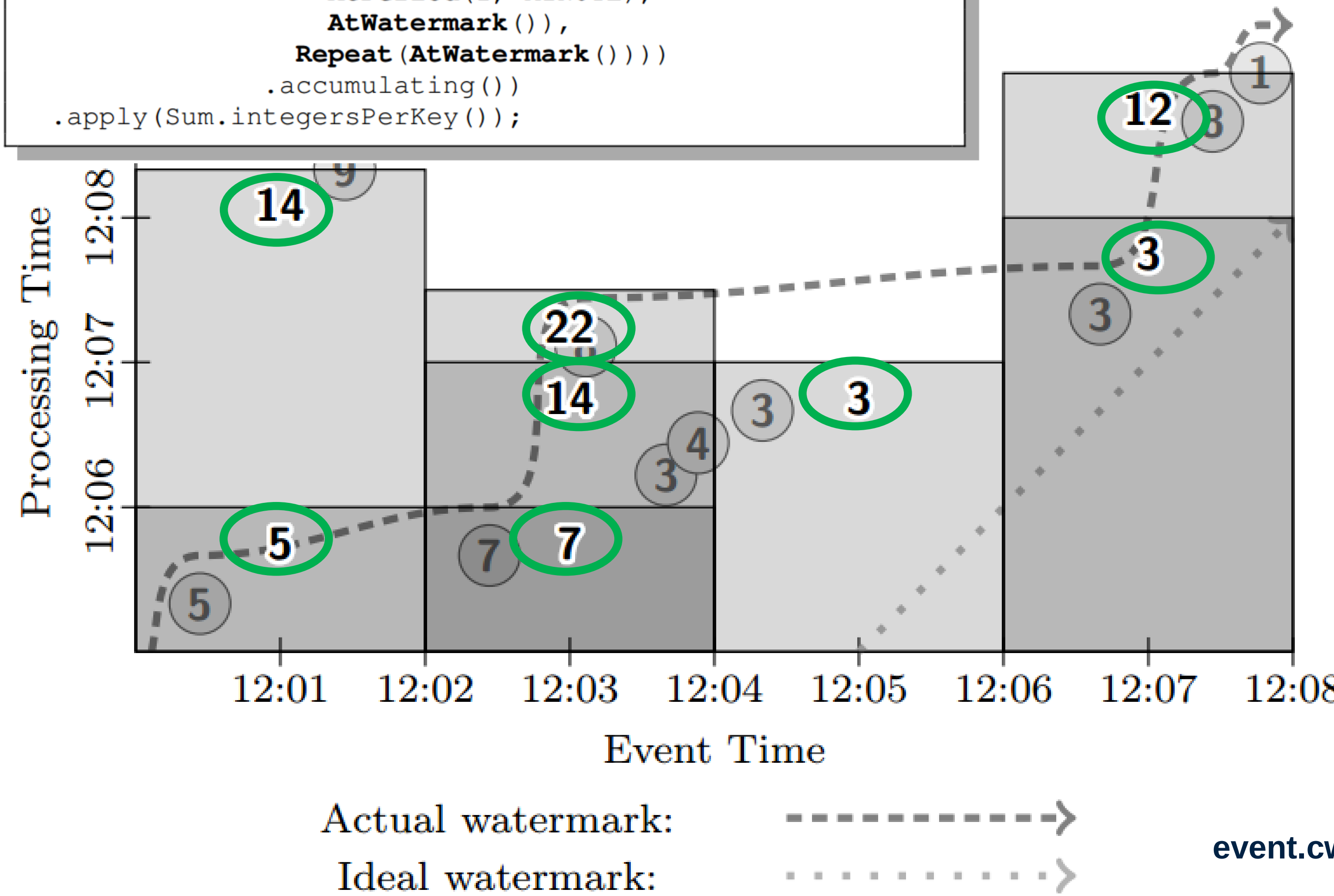
# Triggering: FixedWindows, Batch

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(FixedWindows.of(2, MINUTES))
               .trigger(Repeat(AtWatermark()))
               .accumulating())
  .apply(Sum.integersPerKey());
```



# FixedWindows, Streaming, Partial

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(FixedWindows.of(2, MINUTES))
               .trigger(SequenceOf(
                 RepeatUntil(
                   AtPeriod(1, MINUTE),
                   AtWatermark()),
                 Repeat(AtWatermark())))
               .accumulating())
  .apply(Sum.integersPerKey());
```

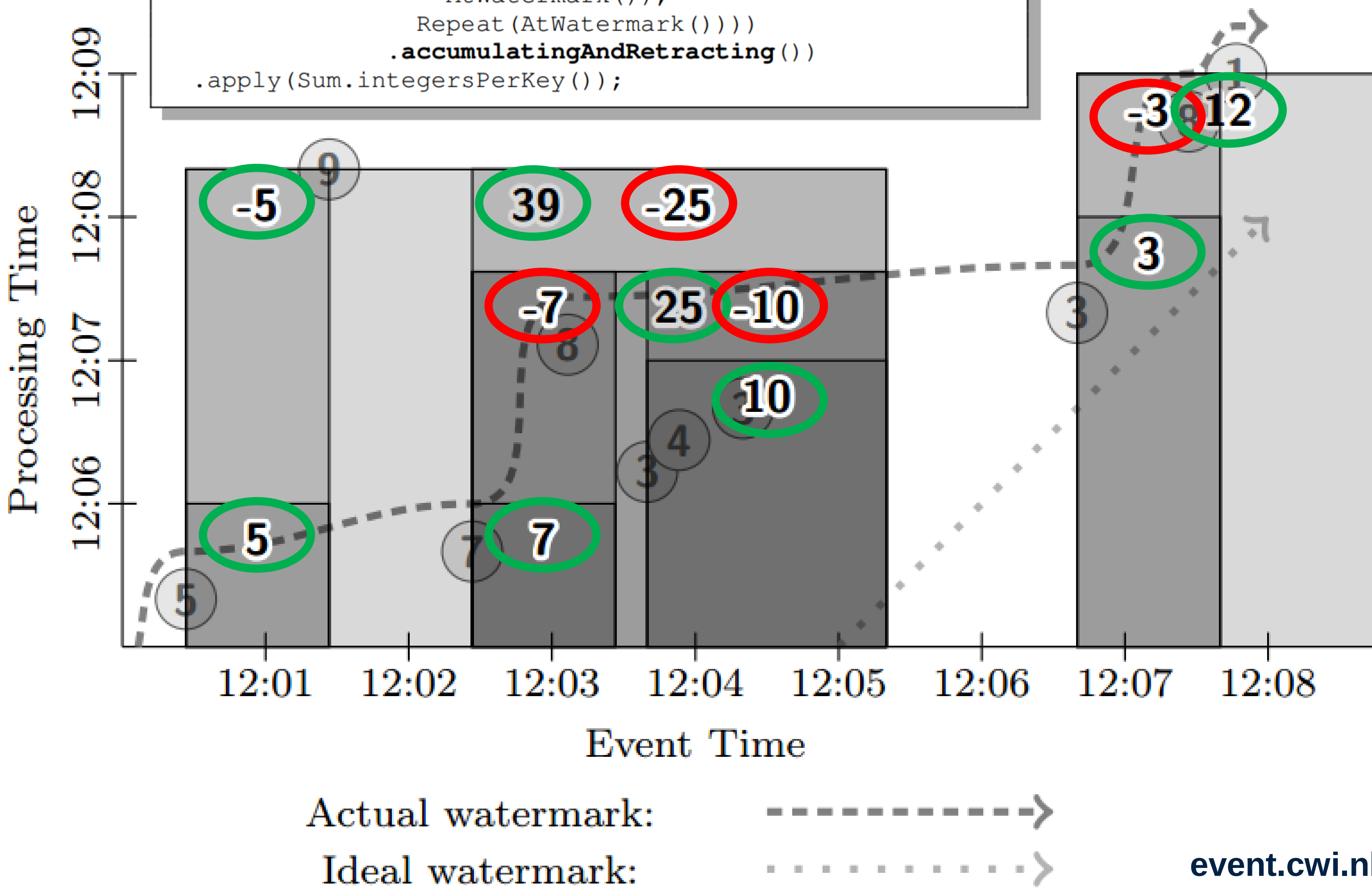



event.cwi.nl/lsde

Centrum Wiskunde & Informatica

# FixedWindows, Streaming, Retracting

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(Sessions.withGapDuration(1, MINUTE))
               .trigger(SequenceOf(
                 RepeatUntil(
                   AtPeriod(1, MINUTE),
                   AtWatermark()),
                 Repeat(AtWatermark())))
               .accumulatingAndRetracting())
  .apply(Sum.integersPerKey());
```



# Summary

- Introduced the notion of data streams and data stream processing
  - DSMS: persistent queries, transient data (opposite of DBMS)
- Described use-cases and algorithms for stream mining
  - Lossy counting
- Introduced frameworks for low-latency stream processing
  - Storm
    - Stream engine, not very Hadoop integrated (separate cluster)
  - Spark Streaming
    - “Micro-batching”, re-use of RDD concept
  - Google Dataflow
    - Map-Reduce++ with streaming built-in (advanced windowing)
    - Finegrained control over the freshness of computations
    - Avoids “Lambda Architecture” – two systems for batch and streaming