Dataflow Model

A Practical Approach to Balancing Correctness, Latency and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing

Raymond Tay 17 Nov 2020

Unified Model

Common Framework to describe (parallel) computation independent of the engine

- Allows for calculation of event-time ordered
- Decomposes pipeline implementation across 4 dimensions
  - **What** results are being computed
  - **Where** in event time they are being computed
  - **When** in processing time they're materialised
  - **How** earlier results relate to later refinements
- Separation of Concerns
  - Description of pipeline is "separated" from implementation of pipeline

The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak,
Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills,
Frances Perry, Eric Schmidt, Sam Whittle
Google
{takidau, robertwb, chambers, chernyak, rfernand,
relax, sgmc, millsd, fjp, cloude, samuelw}@google.com

ABSTRACT

Unbounded, unordered, global-scale datasets are increasingly common in day-to-day business (e.g. Web logs, mobile usage statistics, and sensor networks). At the same time, consumers of these datasets have evolved sophisticated requirements, such as event-time ordering and windowing by features of the data themselves, in addition to an insatiable hunger for faster answers. Meanwhile, practicality dictates that one can never fully optimize along all dimensions of correctness, latency, and cost for these types of input. As a result, data processing practitioners are left with the quandary

1. INTRODUCTION

Modern data processing is a complex and exciting field. From the scale enabled by MapReduce [16] and its successors (e.g Hadoop [4], Pig [18], Hive [29], Spark [33]), to the vast body of work on streaming within the SQL community (e.g. query systems [1, 14, 15], windowing [22], data streams [24], time domains [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [5], modern consumers of data wield remarkable amounts of power in shaping and taming massive-scale disorder into organized structures with far

Unified Model

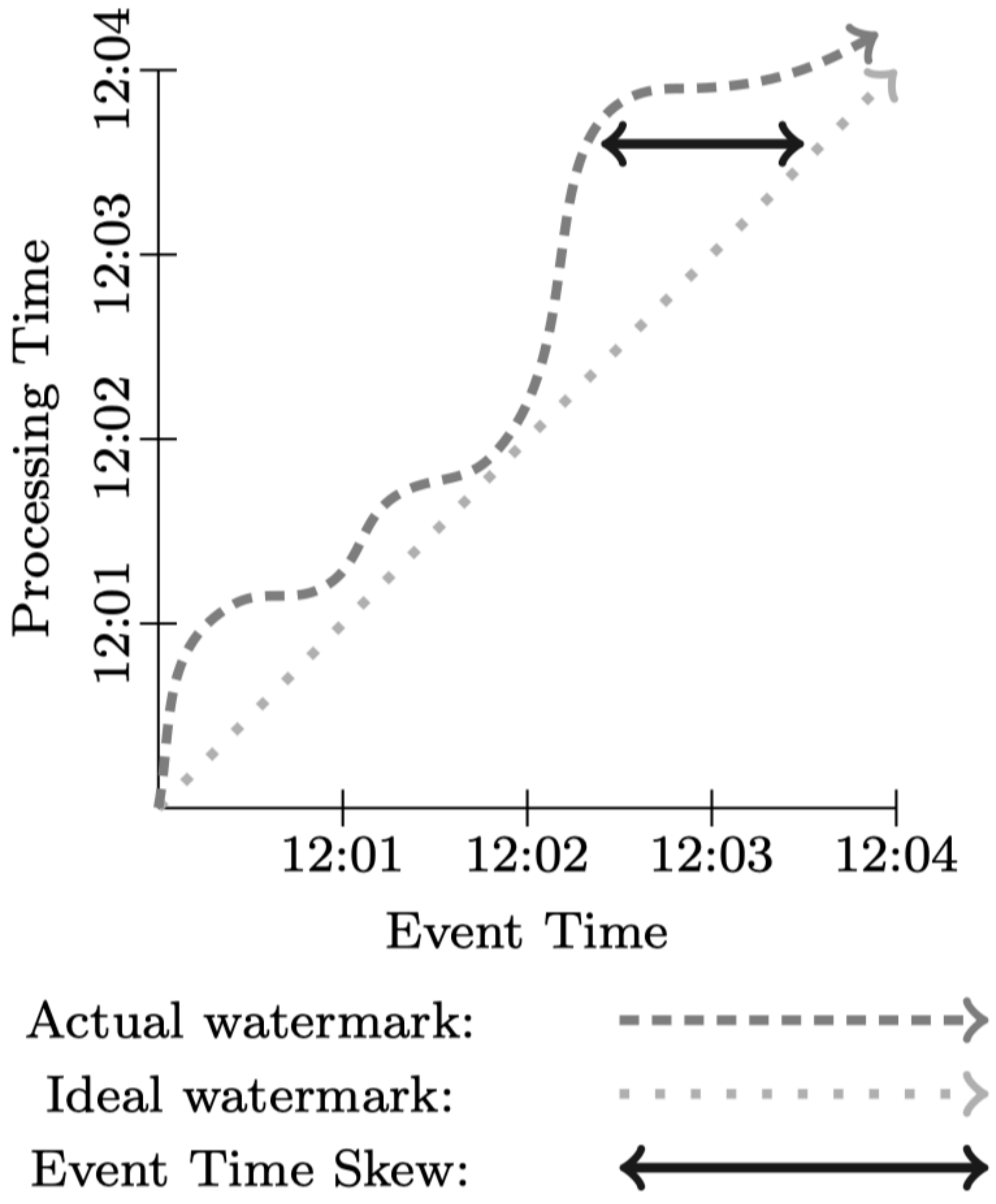
Assumptions

- Unbounded Data Sets
  - Data is a *stream*
  - A *batch* is a *window* into the stream
  - Window-ing is a **time-based** mechanism to focus your attention on the specified duration
- Time
  - Event-time, is the time where events actually occurred
  - Processing-time, is the time where events are observed in the system

Unified Model

Time Domain Skew

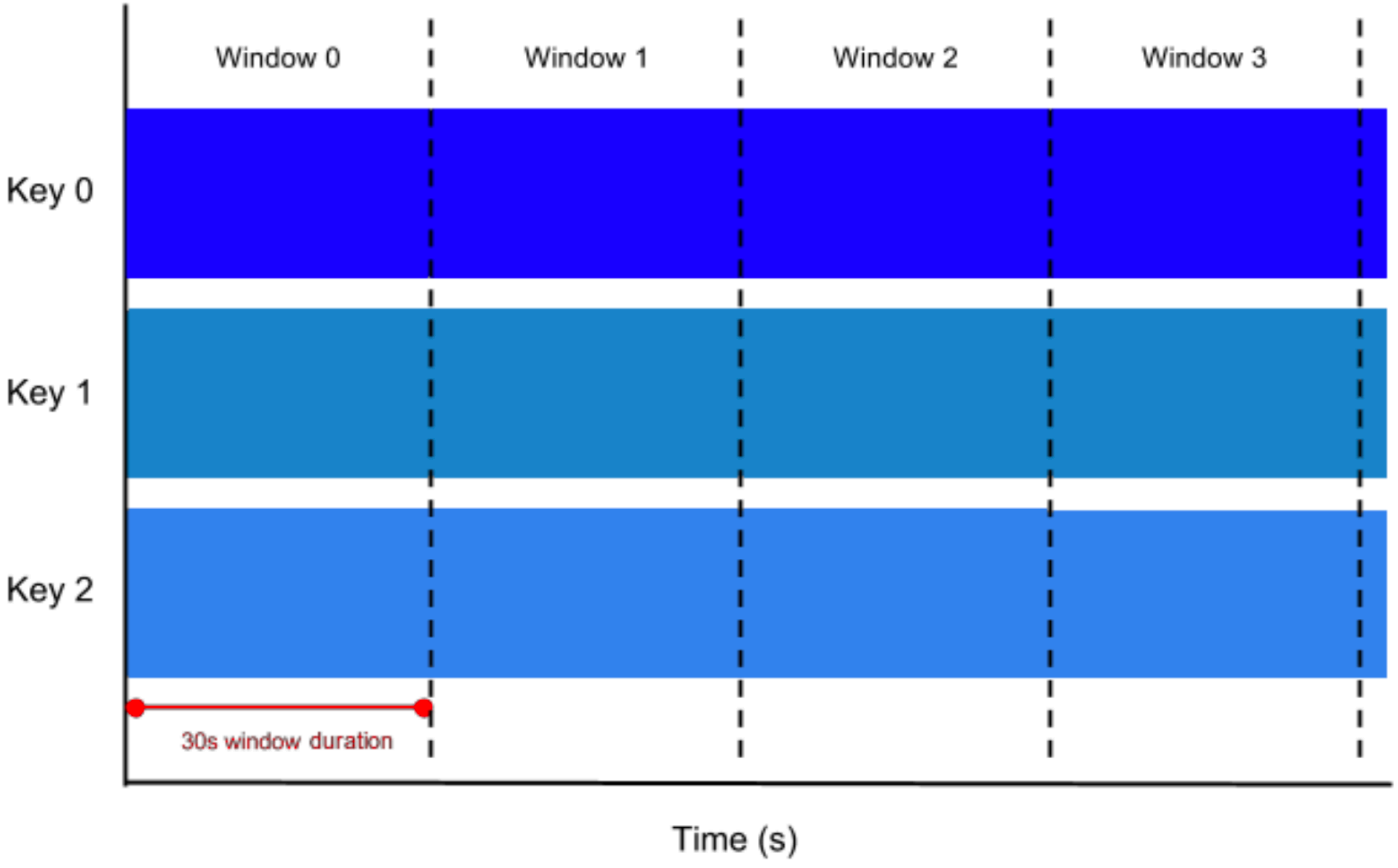
- In an ideal world, the time skew is **zero**
- However, the real world introduces *uncertainties* and the system must be able to handle it to provide correct and repeatable results
- A Time-domain mapping is the defacto tool for streaming analysis



Unified Model

Fixed Windows

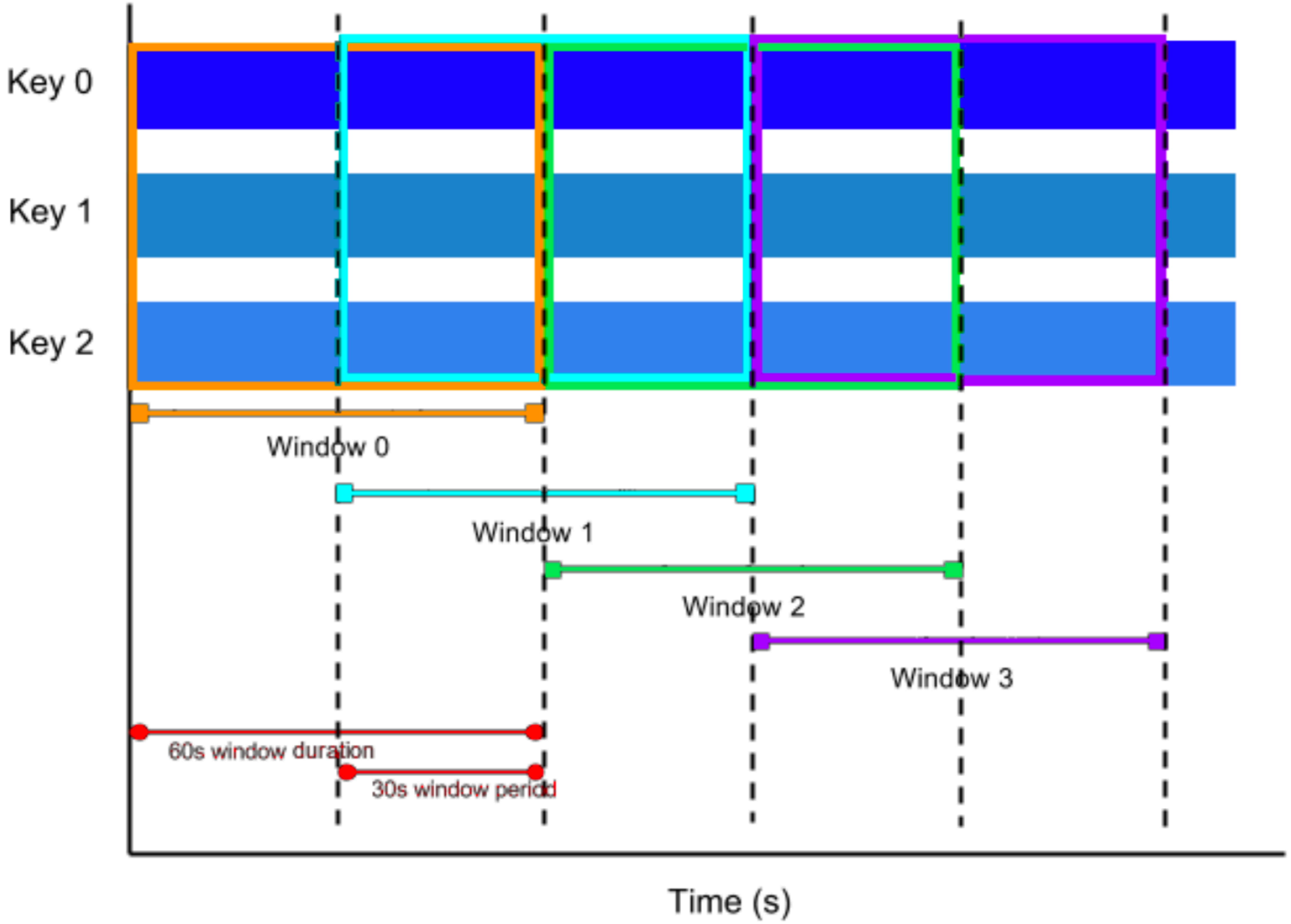
A **fixed window** represents a consistent duration, non overlapping time interval in the data stream



Unified Model

Sliding Windows

A **sliding time window** also represents time intervals in the data stream; however, sliding time windows can overlap. Because multiple windows overlap, most elements in a data set will belong to more than one window

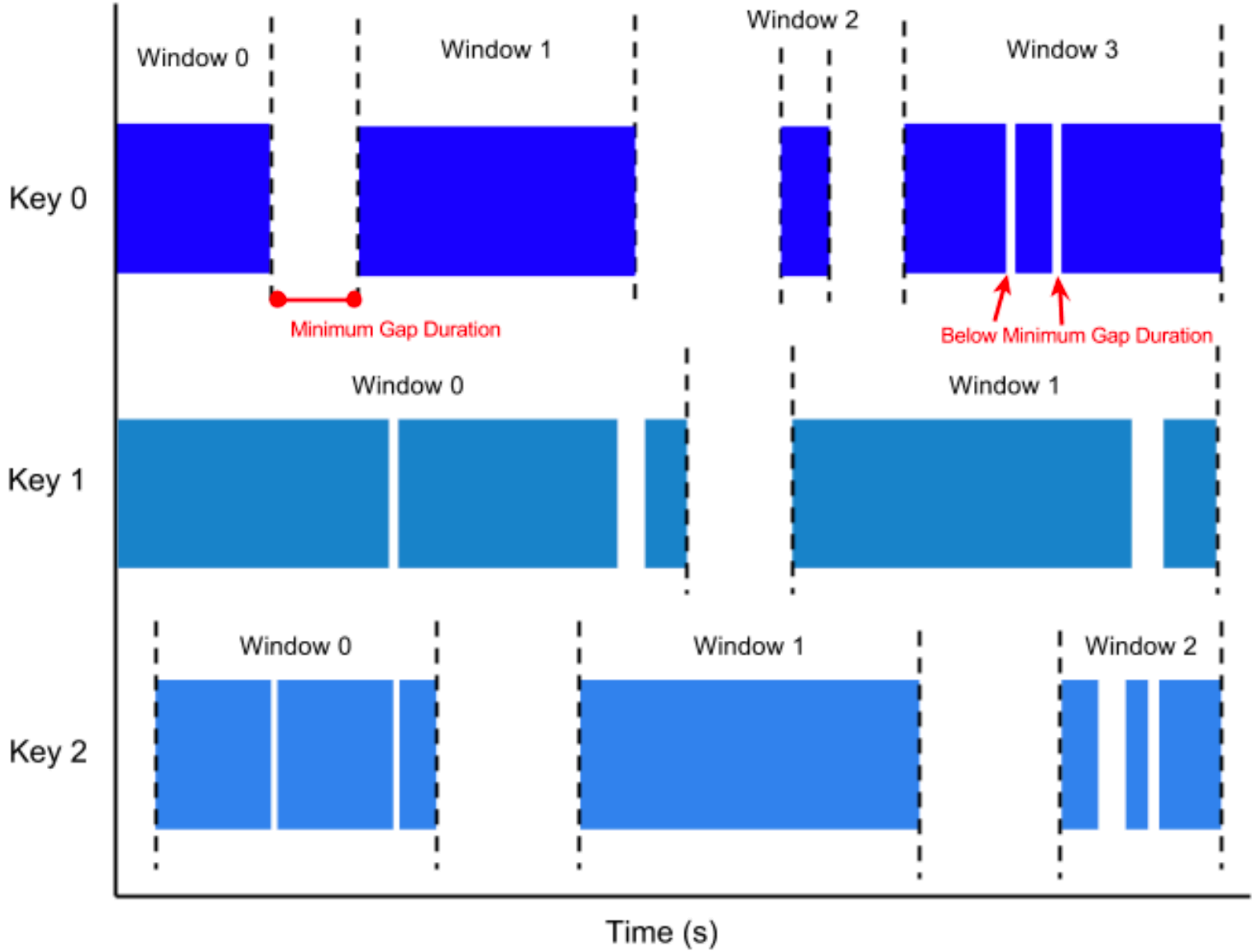


Unified Model

Session Windows

A **session window** function defines windows that contain elements that are within a certain gap duration of another element. Session windowing applies on a per-key basis and is useful for data that is irregularly distributed with respect to time.

If data arrives after the minimum specified gap duration time, this initiates the start of a new window.



Dataflow Model

Core Primitives

- Dataflow SDK provides **two** (2) transforms that operates on (**key**, **value**) pairs through the system
  - **ParDo**
  - **GroupByKey**

$$(fix, 1), (fit, 2)$$

$$\Big\downarrow \; \begin{array}{l} ParDo(\\ \quad ExpandPrefixes) \end{array}$$

$$(f, 1), (fi, 1), (fix, 1), (f, 2), (fi, 2), (fit, 2)$$

$$(f, 1), (fi, 1), (fix, 1), (f, 2), (fi, 2), (fit, 2)$$

$$\Big\downarrow \; GroupByKey$$

$$(f, [1, 2]), (fi, [1, 2]), (fix, [1]), (fit, [2])$$

Dataflow Model

Core Primitives

- Windowing
  - Assigning Windows
  - Merging Windows

Note: You can leverage the Dataflow SDK to accomplish this.

$$(k, v_1, 12{:}00, [0, \infty)), (k, v_2, 12{:}01, [0, \infty))$$

$$\Big\downarrow \; AssignWindows(Sliding(2m, 1m))$$

$$(k, v_1, 12{:}00, [11{:}59, 12{:}01)),$$
$$(k, v_1, 12{:}00, [12{:}00, 12{:}02)),$$
$$(k, v_2, 12{:}01, [12{:}00, 12{:}02)),$$
$$(k, v_2, 12{:}01, [12{:}01, 12{:}03))$$

Dataflow Model

Core Primitives

- Windowing
  - Assigning Windows
  - Merging Windows

Note: You can leverage the Dataflow SDK to accomplish this.

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

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

$$
\begin{array}{c}
(k_1, v_1, 13{:}02, [0, \infty)),\\
(k_2, v_2, 13{:}14, [0, \infty)),\\
(k_1, v_3, 13{:}57, [0, \infty)),\\
(k_1, v_4, 13{:}20, [0, \infty))
\end{array}
$$

$$\Big\downarrow\ AssignWindows(Sessions(30m))$$

$$
\begin{array}{c}
(k_1, v_1, 13{:}02, [13{:}02, 13{:}32)),\\
(k_2, v_2, 13{:}14, [13{:}14, 13{:}44)),\\
(k_1, v_3, 13{:}57, [13{:}57, 14{:}27)),\\
(k_1, v_4, 13{:}20, [13{:}20, 13{:}50))
\end{array}
$$

$$\Big\downarrow\ DropTimestamps$$

$$
\begin{array}{c}
(k_1, v_1, [13{:}02, 13{:}32)),\\
(k_2, v_2, [13{:}14, 13{:}44)),\\
(k_1, v_3, [13{:}57, 14{:}27)),\\
(k_1, v_4, [13{:}20, 13{:}50))
\end{array}
$$

$$\Big\downarrow\ GroupByKey$$

$$
\begin{array}{c}
(k_1, [(v_1, [13{:}02, 13{:}32)),\\
(v_3, [13{:}57, 14{:}27)),\\
(v_4, [13{:}20, 13{:}50))]),\\
(k_2, [(v_2, [13{:}14, 13{:}44))])
\end{array}
$$

$$\Big\downarrow\ MergeWindows(Sessions(30m))$$

$$
\begin{array}{c}
(k_1, [(v_1, [\mathbf{13{:}02}, \mathbf{13{:}50})),\\
(v_3, [13{:}57, 14{:}27)),\\
(v_4, [\mathbf{13{:}02}, \mathbf{13{:}50}))]),\\
(k_2, [(v_2, [13{:}14, 13{:}44))])
\end{array}
$$

$$\Big\downarrow\ GroupAlsoByWindow$$

$$
\begin{array}{c}
(k_1, [([\mathbf{v_1}, \mathbf{v_4}], [13{:}02, 13{:}50)),\\
([\mathbf{v_3}], [13{:}57, 14{:}27))]),\\
(k_2, [([\mathbf{v_2}], [13{:}14, 13{:}44))])
\end{array}
$$

$$\Big\downarrow\ ExpandToElements$$

$$
\begin{array}{c}
(k_1, [v_1, v_4], \mathbf{13{:}50}, [13{:}02, 13{:}50)),\\
(k_1, [v_3], \mathbf{14{:}27}, [13{:}57, 14{:}27)),\\
(k_2, [v_2], \mathbf{13{:}44}, [13{:}14, 13{:}44))
\end{array}
$$

Dataflow Model

Core Primitives

- Windowing
  - Assigning Windows
  - Merging Windows

Note: You can leverage the Dataflow SDK to accomplish this.

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

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

Dataflow Model

Triggers and Incremental Processing

- A mechanism is **needed** to handles tuples- and processing-time-based windows (aka *watermarks*)
  - It's the notion of input completeness w.r.t event-time.
- A mechanism is also **needed** to serve results of a window (aka *triggers*)
  - In-built triggers for watermarks, processing time, user-defined
- When multiple windows are to emit results, the available options
  - `Discarding` (default in Apache Beam)
  - `Accumulating`
  - `Accumulating` & `Retracting`

Read Section 2.3 for more details

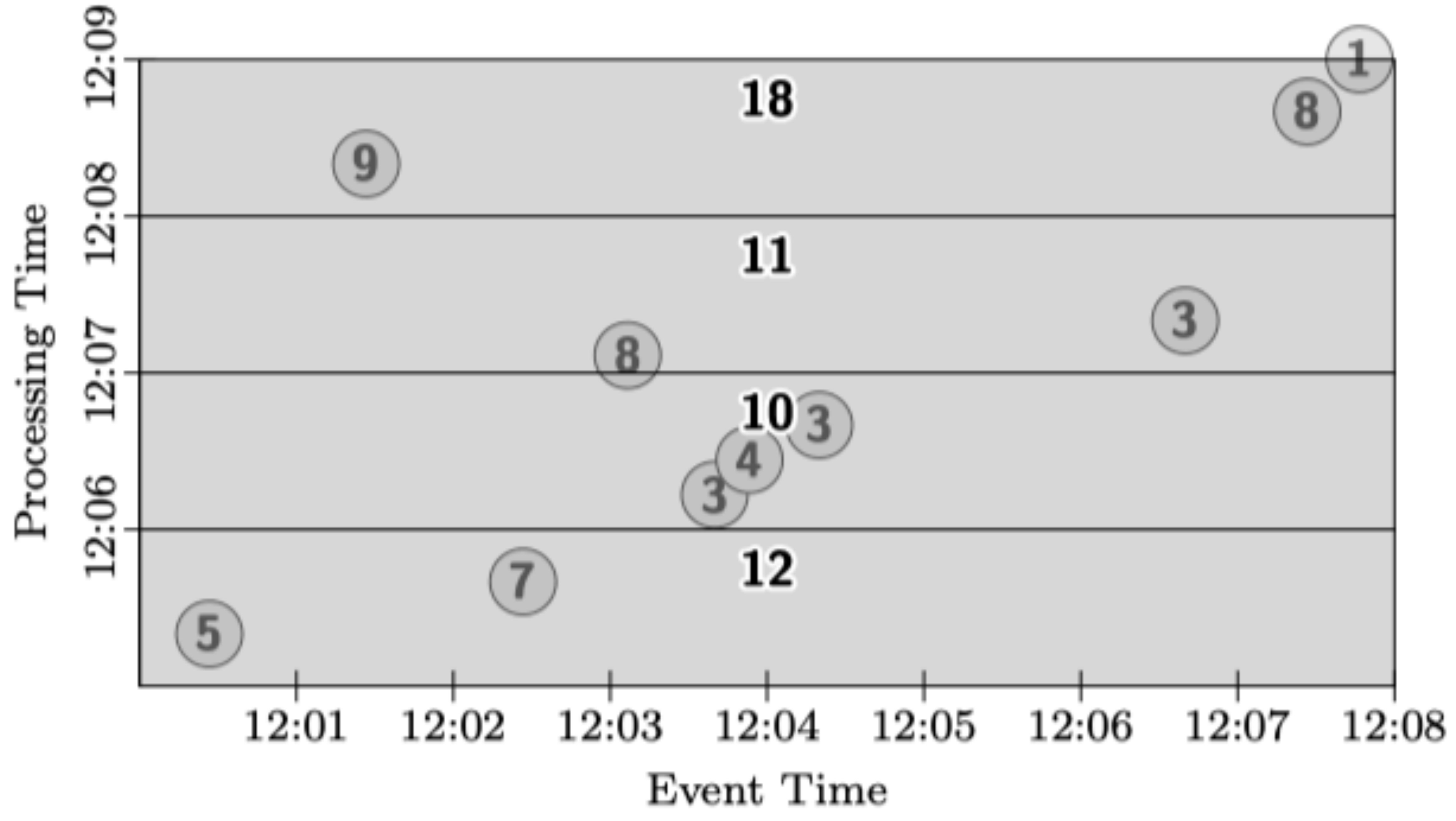
See the Apache Beam SDK on Triggers for more

Dataflow Model

Discarding

- Property of the *trigger* and needs to be declared before use.
- **`Assumption`**`: All data points are classified under some key. In the real-world, the same operations would operate in` **`parallel`** `for multiple keys.`

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



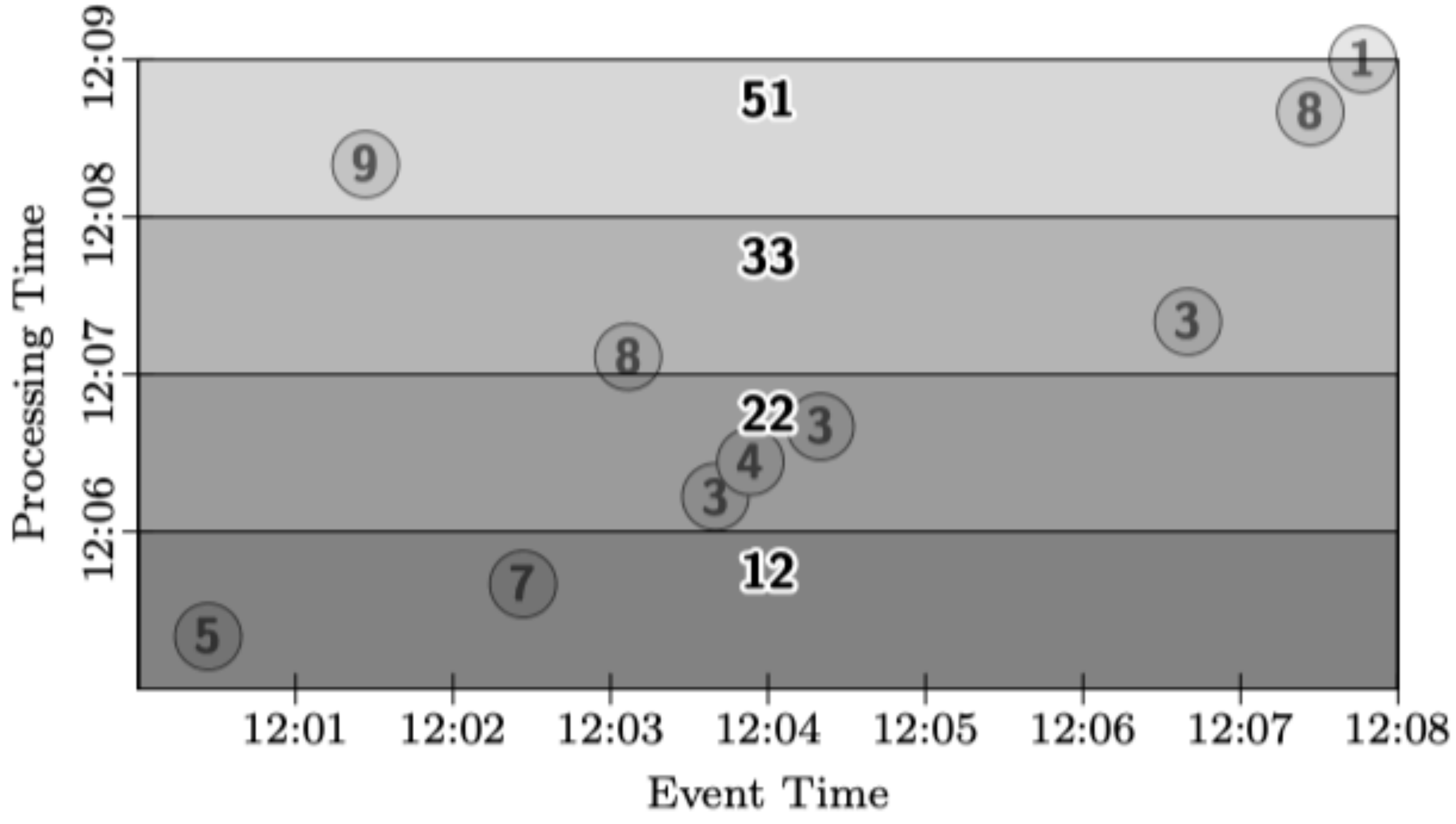
Dataflow Model

Accumulating

- Property of the *trigger* and needs to be declared before use.

- `**Assumption:**` `All data points are classified under some key. In the real-world, the same operations would operate in` `**parallel**` `for multiple keys.`

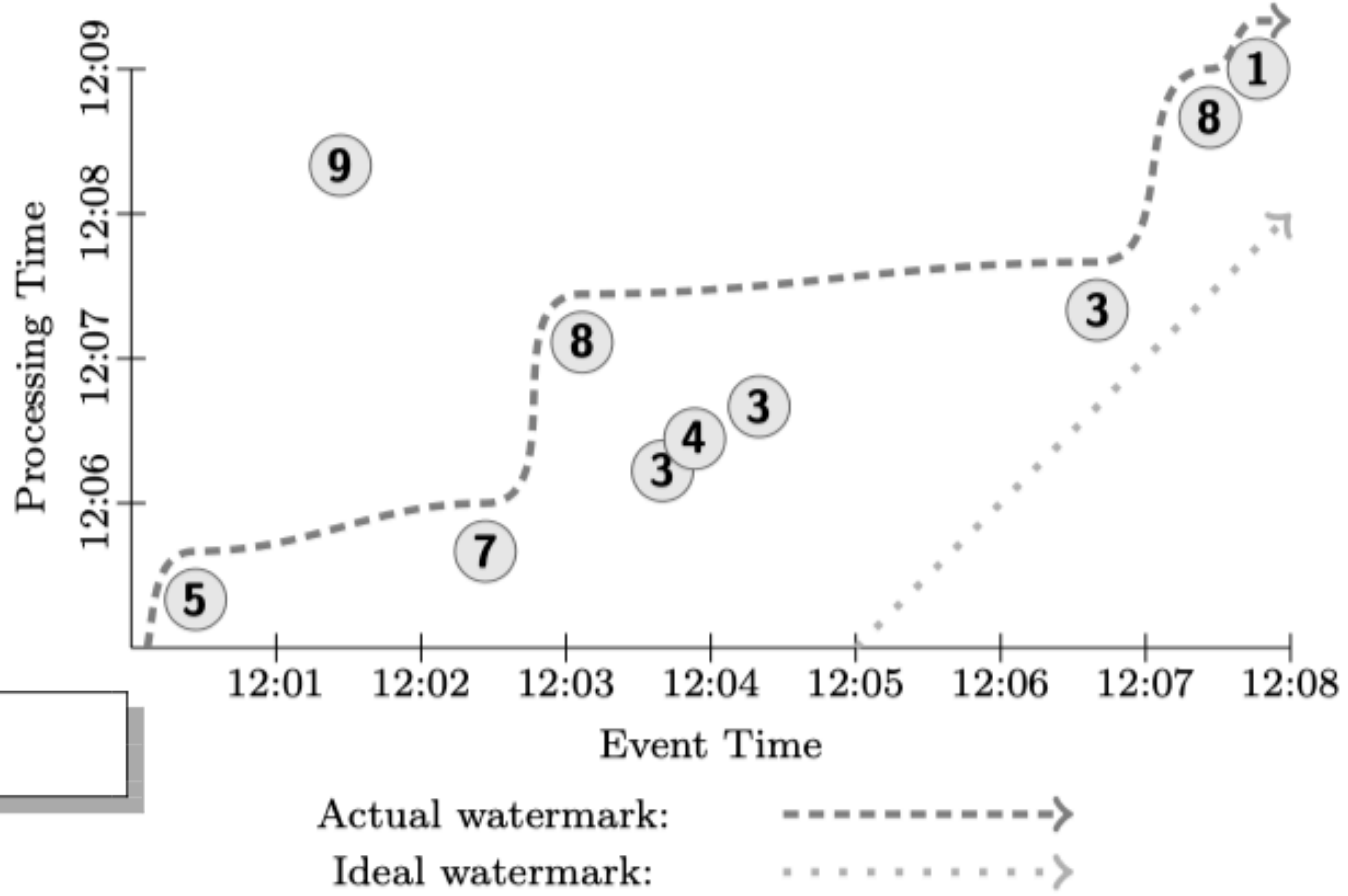
```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
               .accumulating())
  .apply(Sum.integersPerKey());
```



Dataflow Model

Examples - Bounded Data

- Assuming the numbers 1..10 were emitted, out-of-order, and travelled to this model. **What does it look like ?**




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

Dataflow Model

Examples - Bounded Data

- Batch data is event-time agnostic
- Batch is just another perspective of streaming data

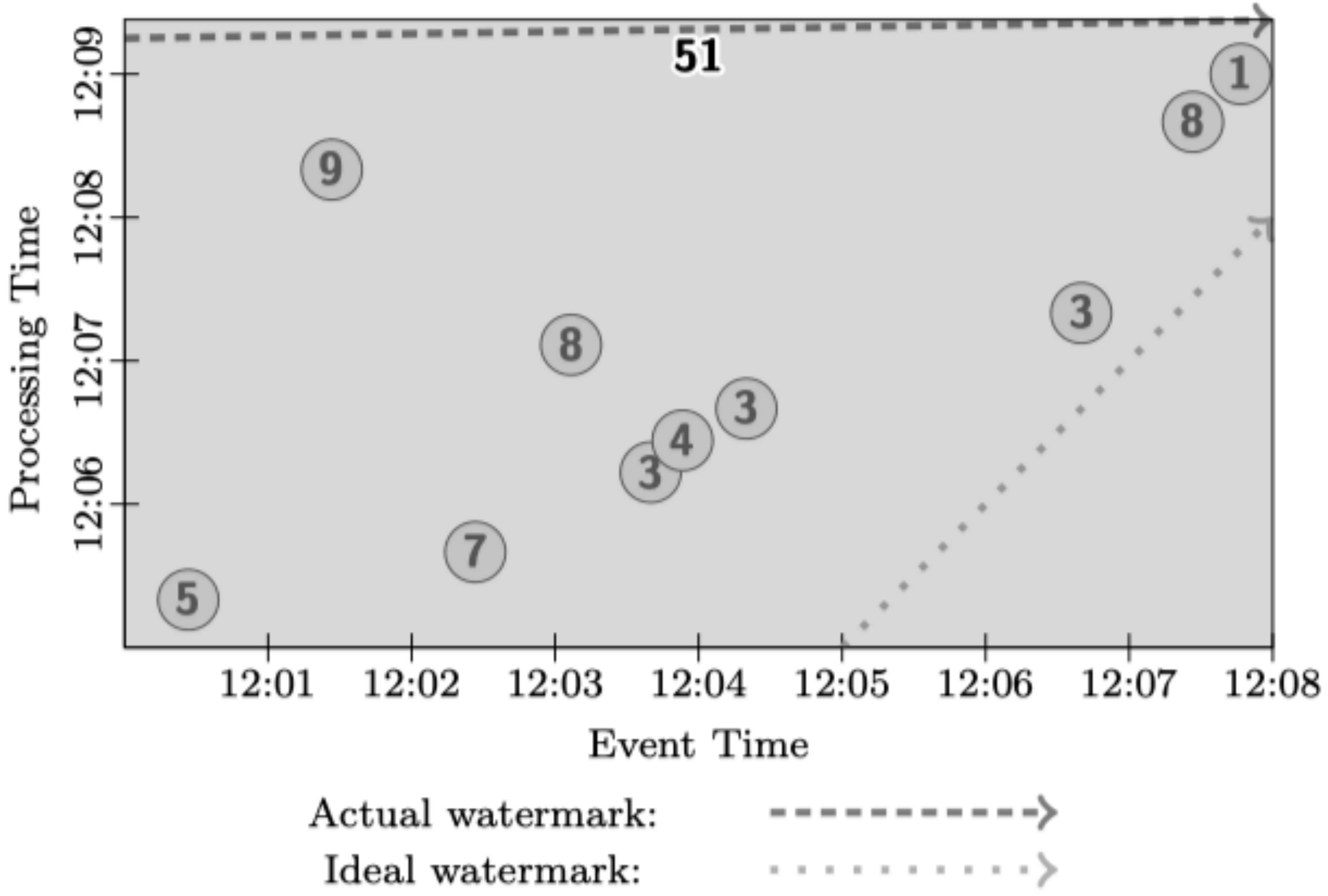


Figure 6: Classic Batch Execution

Dataflow Model

Concluding Remarks

"Based on our many years of experience with real-world, massive-scale, unbounded data processing within Google, we believe the model presented here is a good step in that direction."

References:

* Streaming 101
* Streaming 102
* Design of Approximation Algorithms

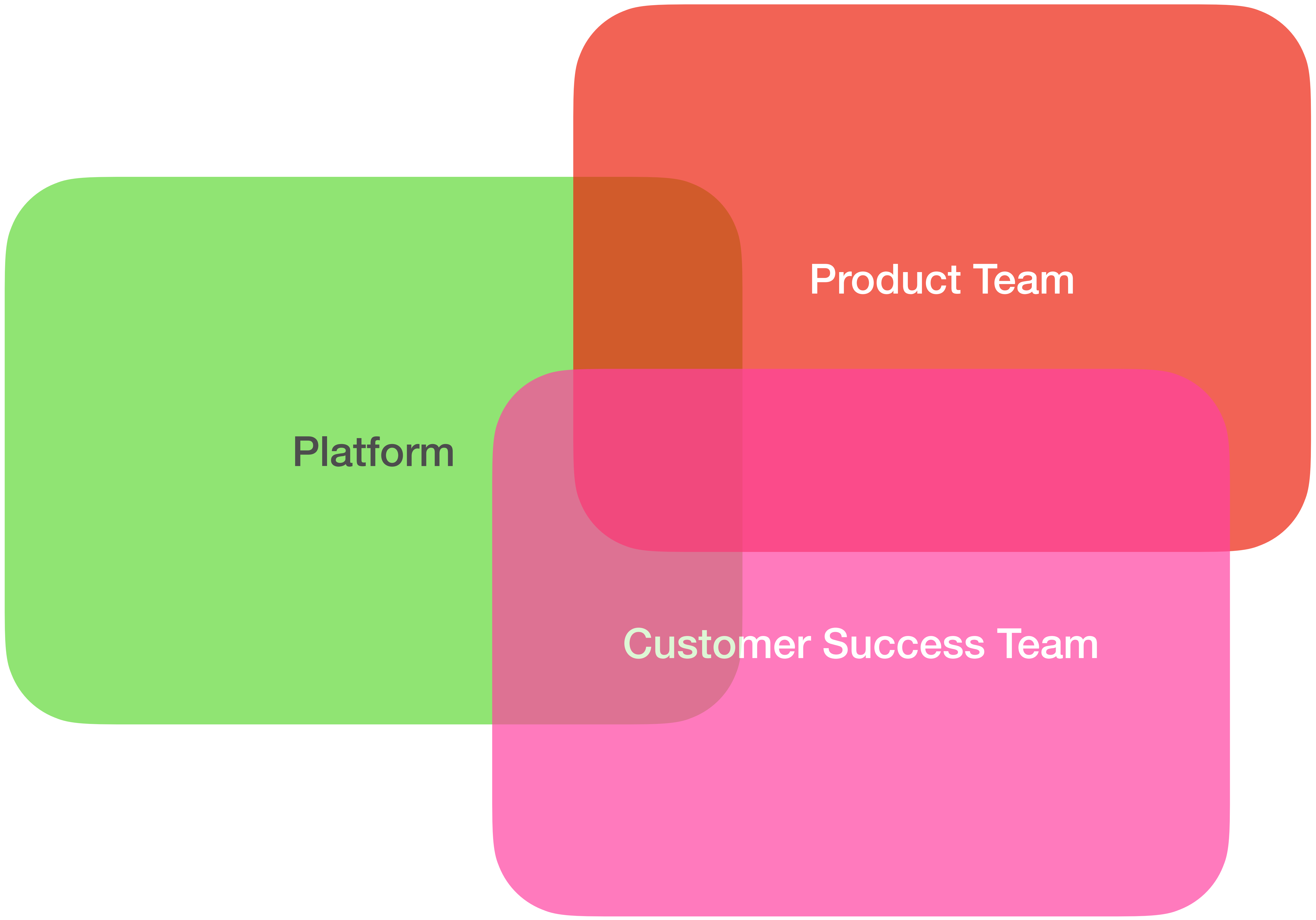
Engineering Re-0rg

Buy in

- Stake Holders
- Product Team
- Engineering Team
- Correct the posture of the Engineering function
- Lower Total Cost Ownership (OpEx to OpEx)

Engineering Organization Interaction

“Cloud Native” on AWS



Engineering Organization

Cloud Native on GCP