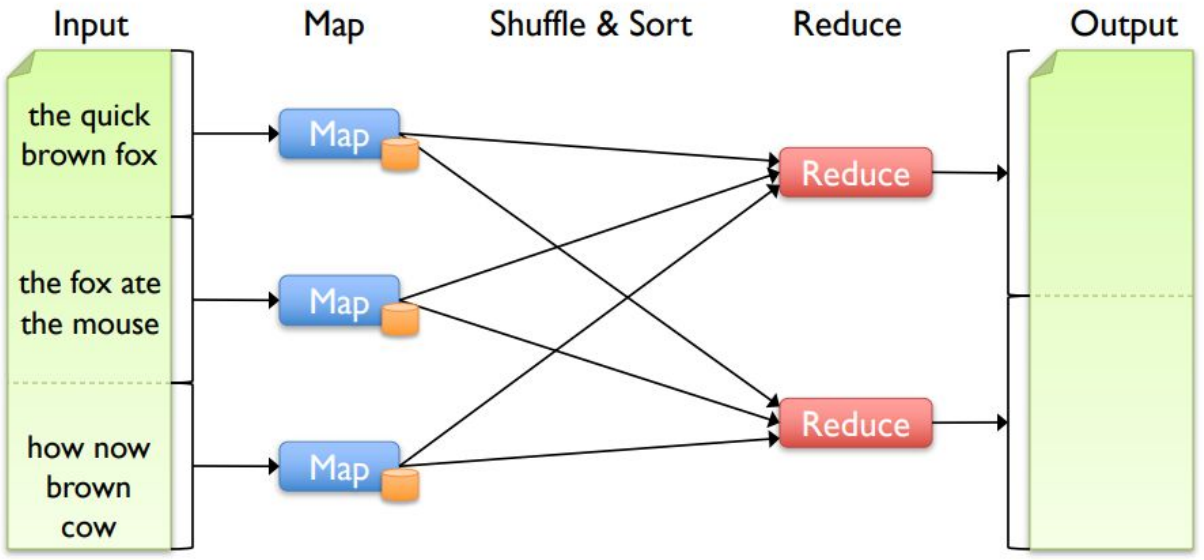
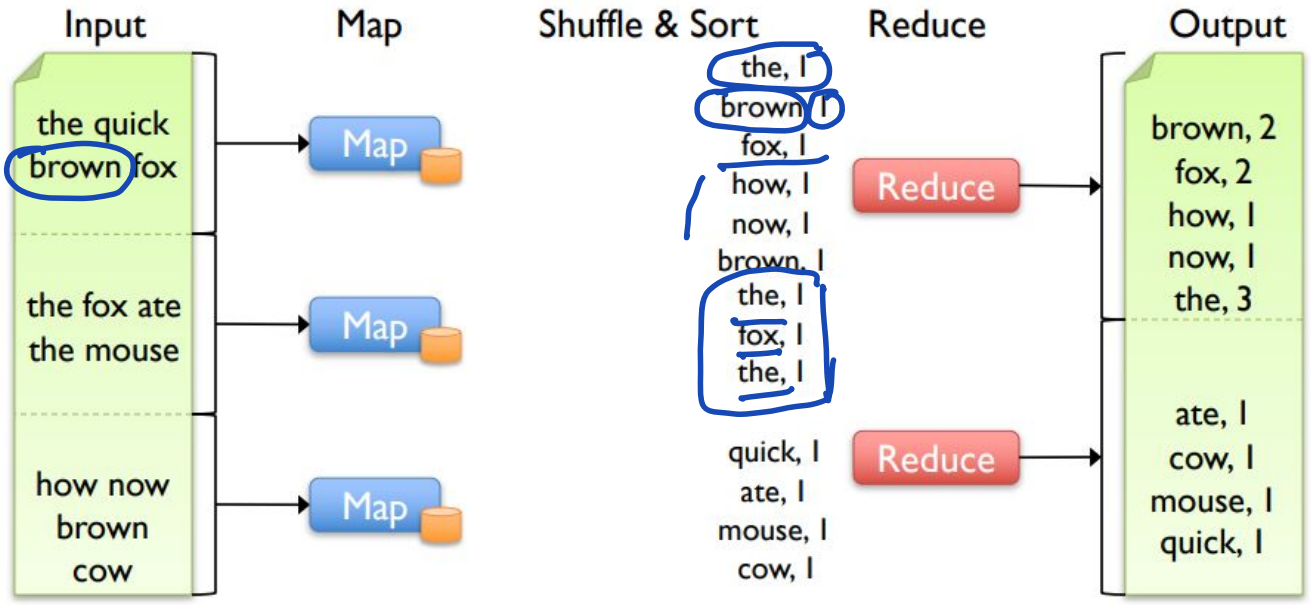
DataFlow Model and Stream Processing

Saurabh Agarwal

Map Reduce - Word Count



Map Reduce - Word Count



Spark - Motivation

- Most Real Applications require multiple MR Steps
  - Indexing Pipeline - 21 Steps
  - Analytics Query - 5-11 Steps

① Writing to disk, do shuffle operator, Read again

Spark - Motivation

- Most Real Applications require multiple MR Steps
  - Indexing Pipeline - 21 Steps
  - Analytics Query - 5-11 Steps

Spark - Motivation

- Most Real Applications require multiple MR Steps
  - Indexing Pipeline - 21 Steps
  - Analytics Query - 5-11 Steps

Write to file System, expensive to reuse data (pagerank, interactive data)

Resilient distributed data frames

Spark

- Resilient distributed datafrmaes
  - Immutable partitioned collections of objects
  - Transformation to build RDD's and actions to execute

→ Lineage graph ↓
↓ map (--->)
count ( ) ↓
!
execute ()

- Clean programmable API

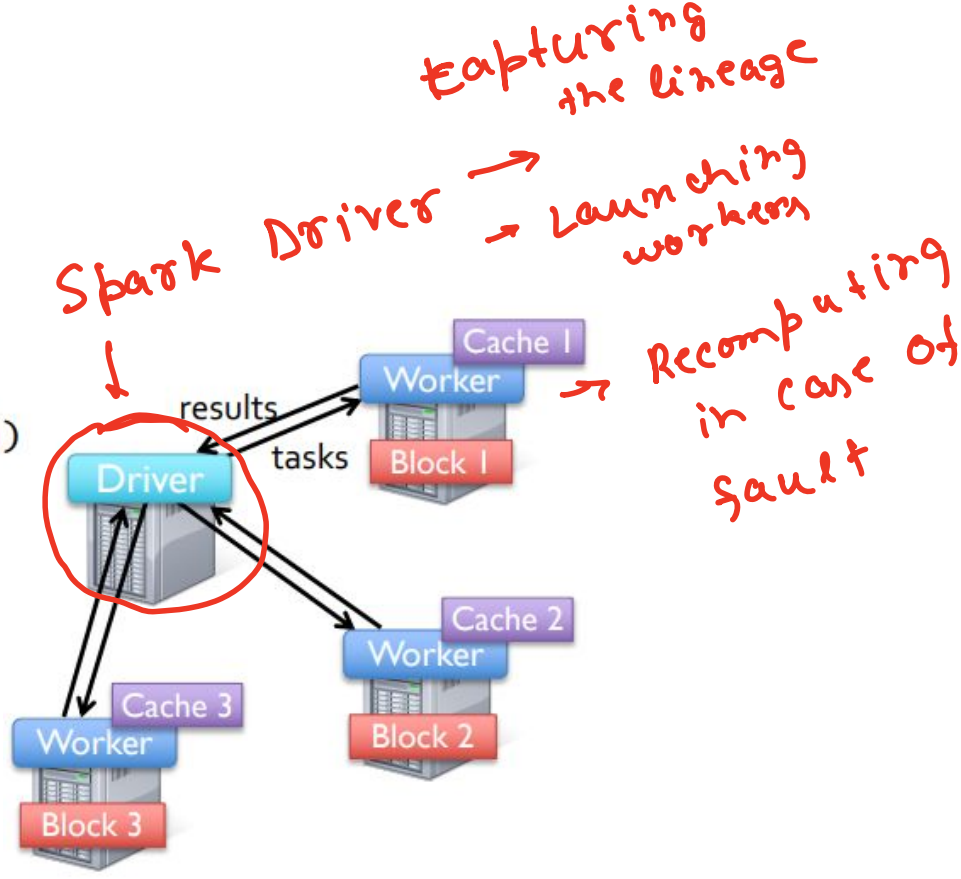
- Fault tolerance and in-memory processing

RDD's → compiler Optimizations
→ Fault tolerance

Spark - Example

- Read from a Log and filter errors

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.cache()
messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```



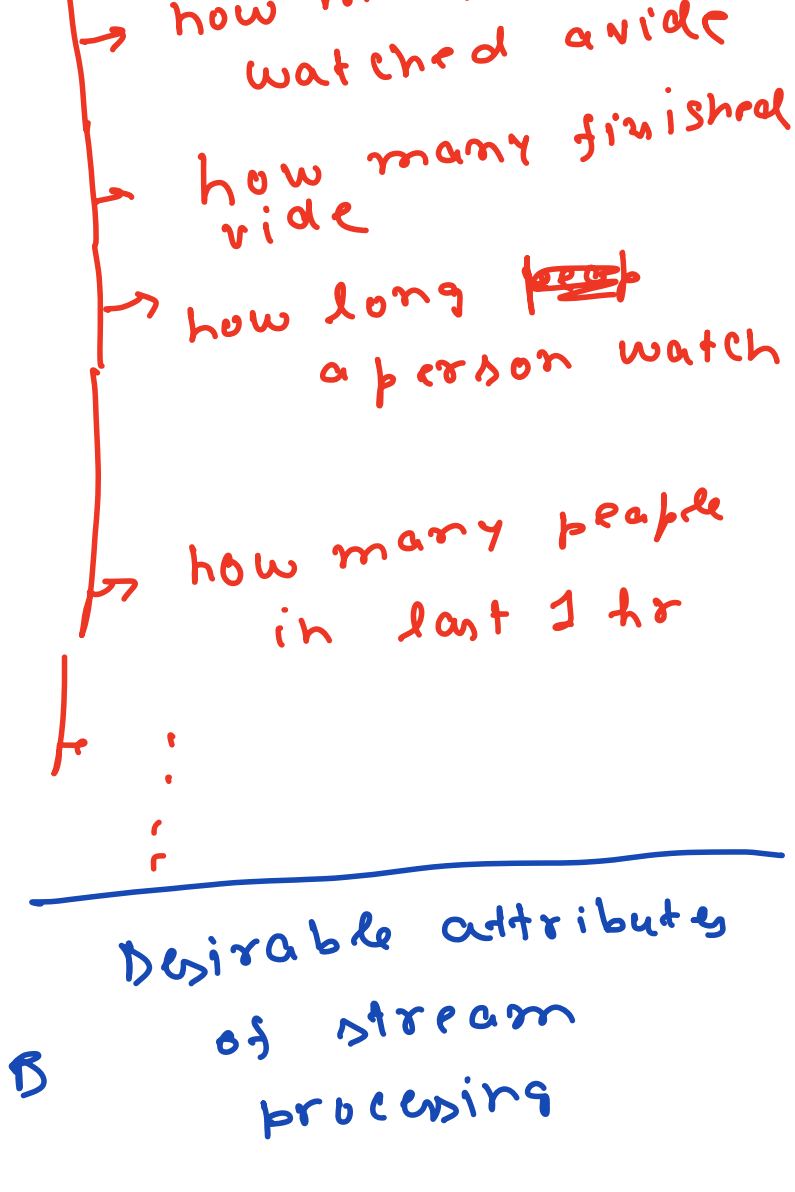
Stream Processing

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

Motivation

Consider an initial example: a streaming video provider wants to monetize their content by displaying video ads and billing advertisers for the amount of advertising watched. The platform supports online and offline views for content and ads. The video provider wants to know how much to bill each advertiser each day, as well as aggregate statistics about the videos and ads. In addition, they want to efficiently run offline experiments over large swaths of historical data.

Advertisers/content providers want to know how often and for how long their videos are being watched, with which content/ads, and by which demographic groups. They also want to know how much they are being charged/paid. They want all of this information as quickly as possible, so that they can adjust budgets and bids, change targeting, tweak campaigns, and plan future directions in as close to real time as possible. Since money is involved, correctness is paramount.



Solution

- Separate user API from Execution

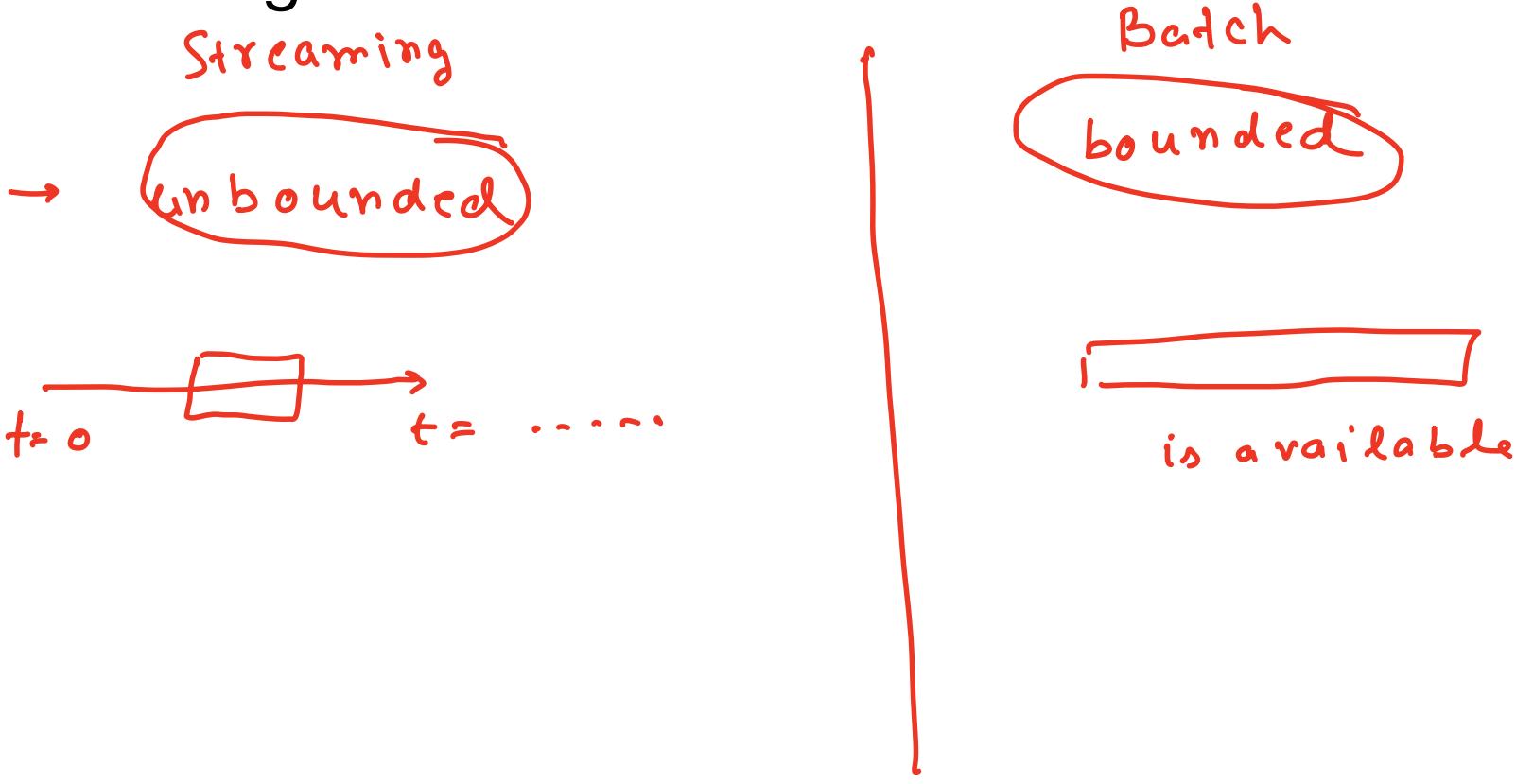
- Decompose Queries into
  - What to compute
  - Where in **Event Time** they are being computed
  - When in **Processing Time** they are being materialized
  - How earlier results are related

Time stamp at which event occured

actually process the data

1. Speed
2. Scalability
3. Query modification
4. & Fault tolerant
5. Minimize framework overhead
6. Correctness

Streaming vs Batch



Windowing

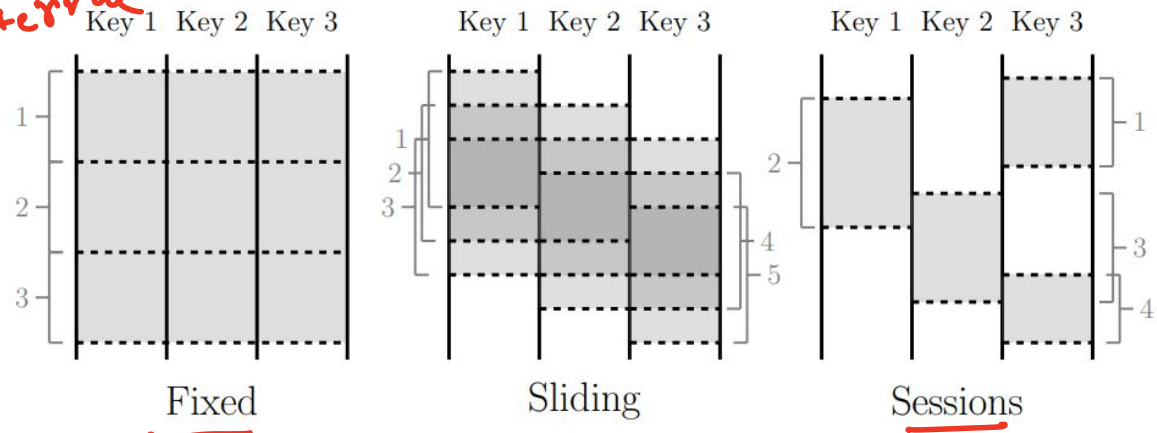
- Required for some operations, unnecessary for others

- Windowing in Batch data ??

Windowing

① Partition the data into pre-defined time interval

ex → partitioned data a day



t=0 → t → t=3

t=1 → t=4

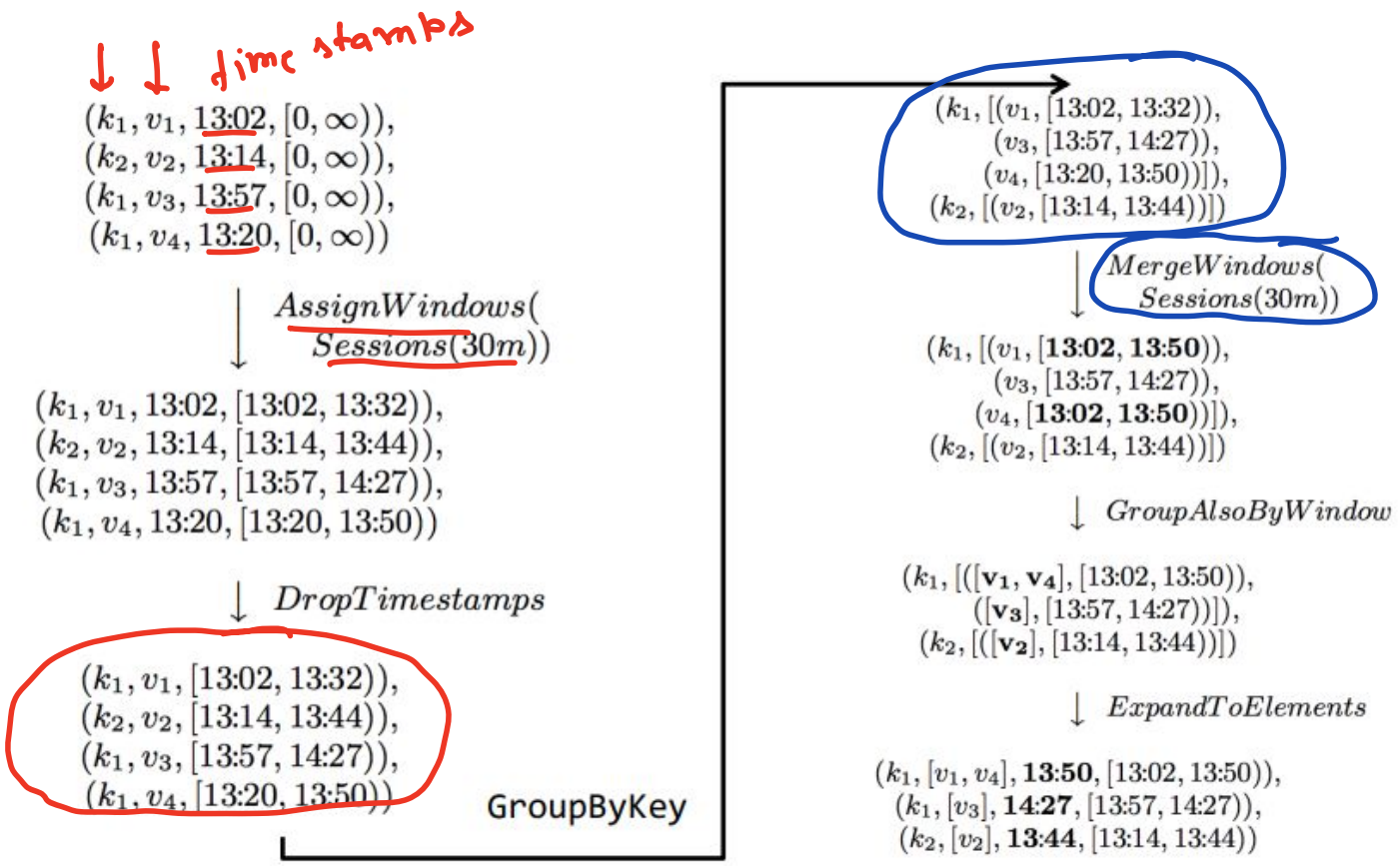
Data Flow Model API

- ParDo:

- GroupByKey:

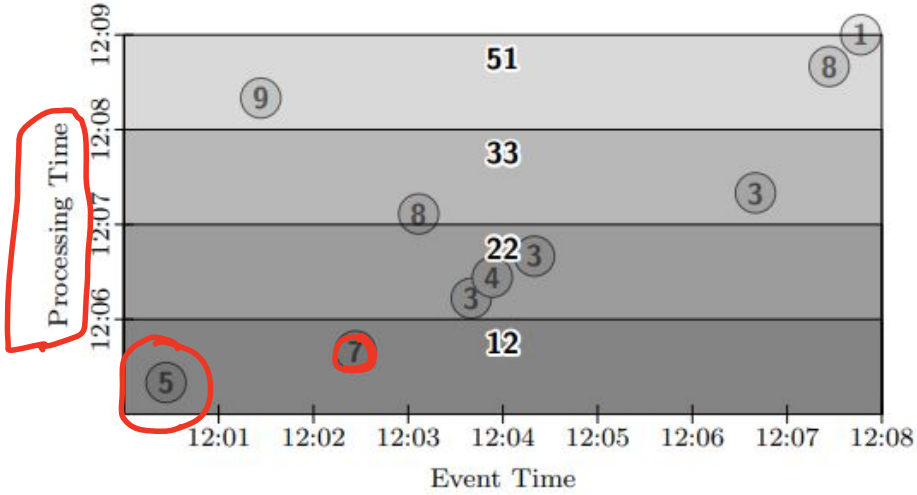
- Windowing
  - AssignWindow
  - MergeWindow

Data Flow Model API



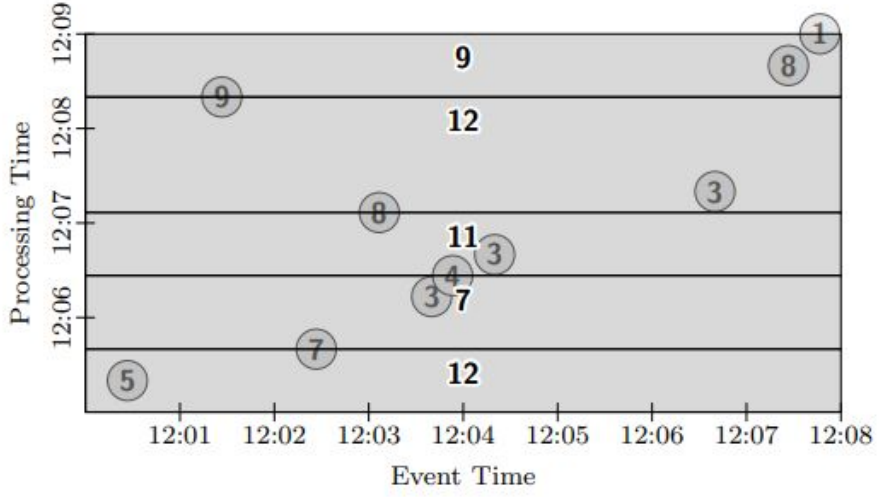
Fixed Interval

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



Fixed Data Count

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



Micro-Batch Processing

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