Architectures for massive data management

Twitter Heron and Google Cloud DataFlow

Albert Bifet
albert.bifet@telecom-paristech.fr

October 20, 2015

Architectures

Lambda Architecture

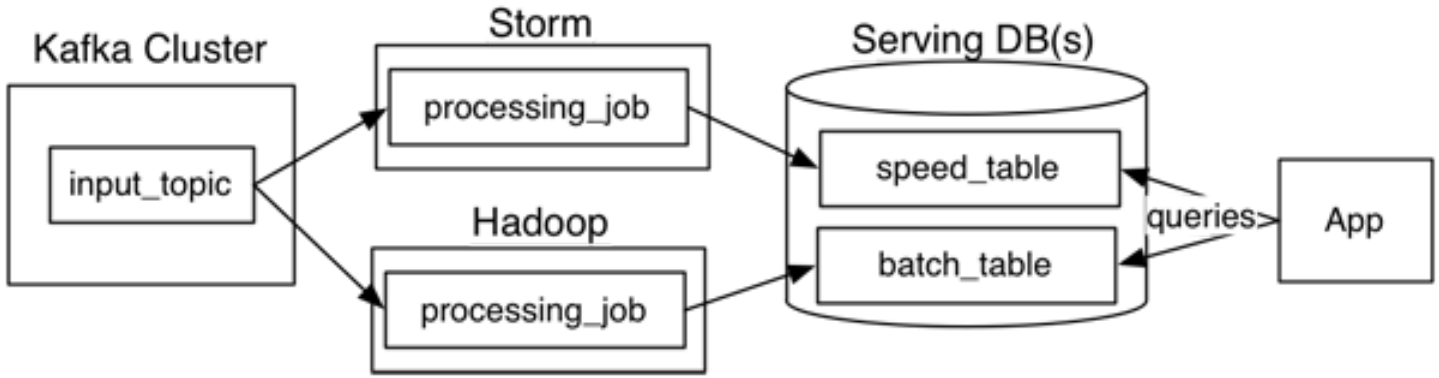


Figure: Nathan Marz

Kappa Architecture

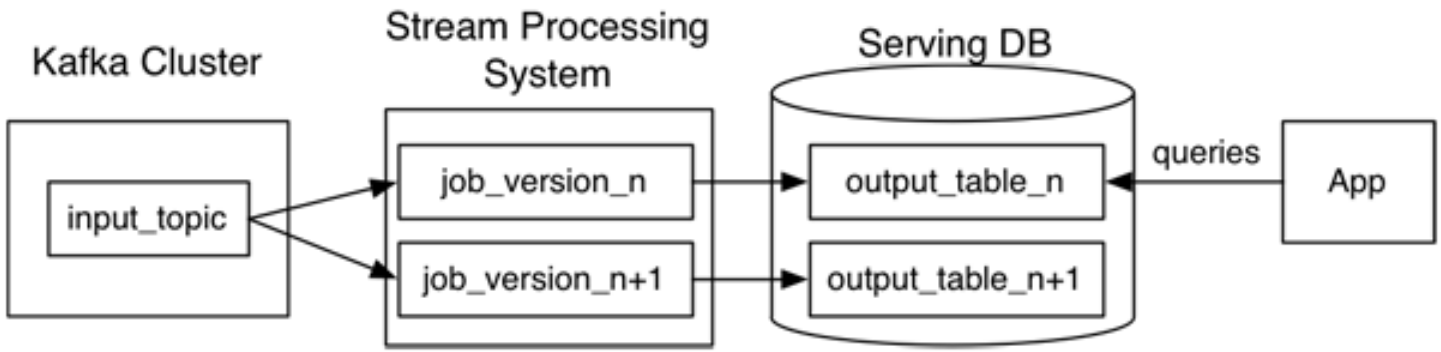


Figure: Questioning the Lambda Architecture by Jay Kreps

Twitter Heron

Twitter Heron

Heron includes these features:

1. Off the shelf scheduler
2. Handling spikes and congestion
3. Easy debugging
4. Compatibility with Storm
5. Scalability and latency

Twitter Heron

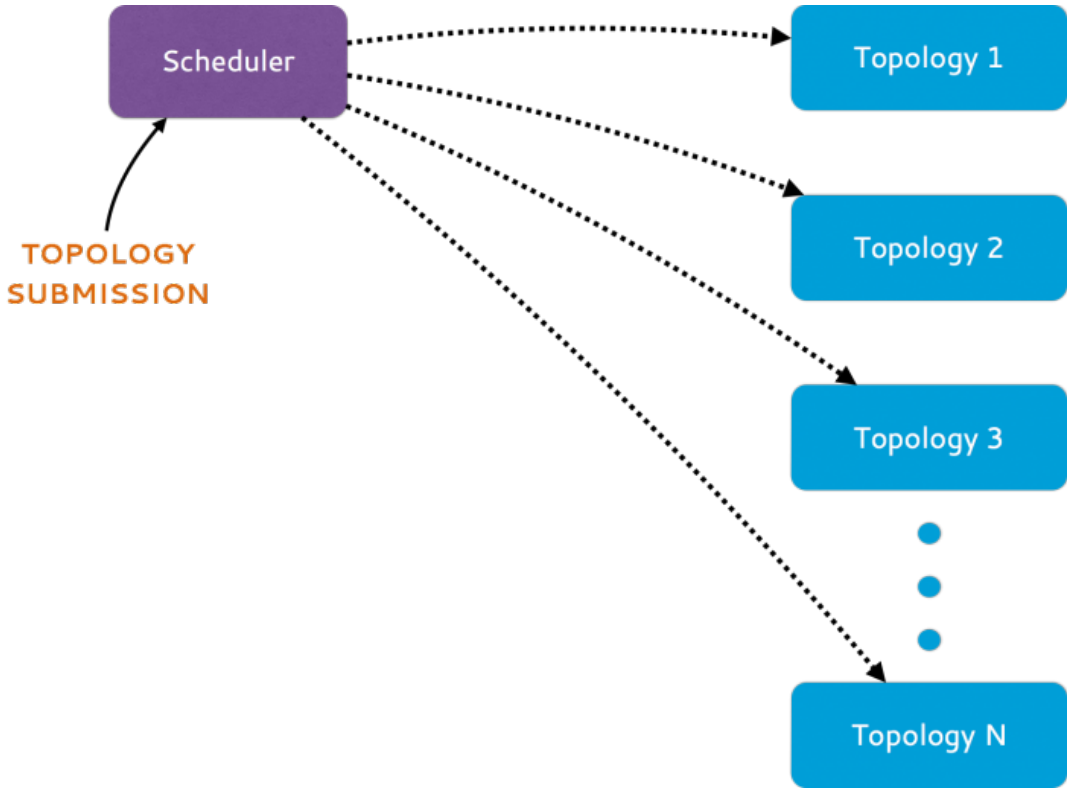


Figure: Heron Architecture

Twitter Heron

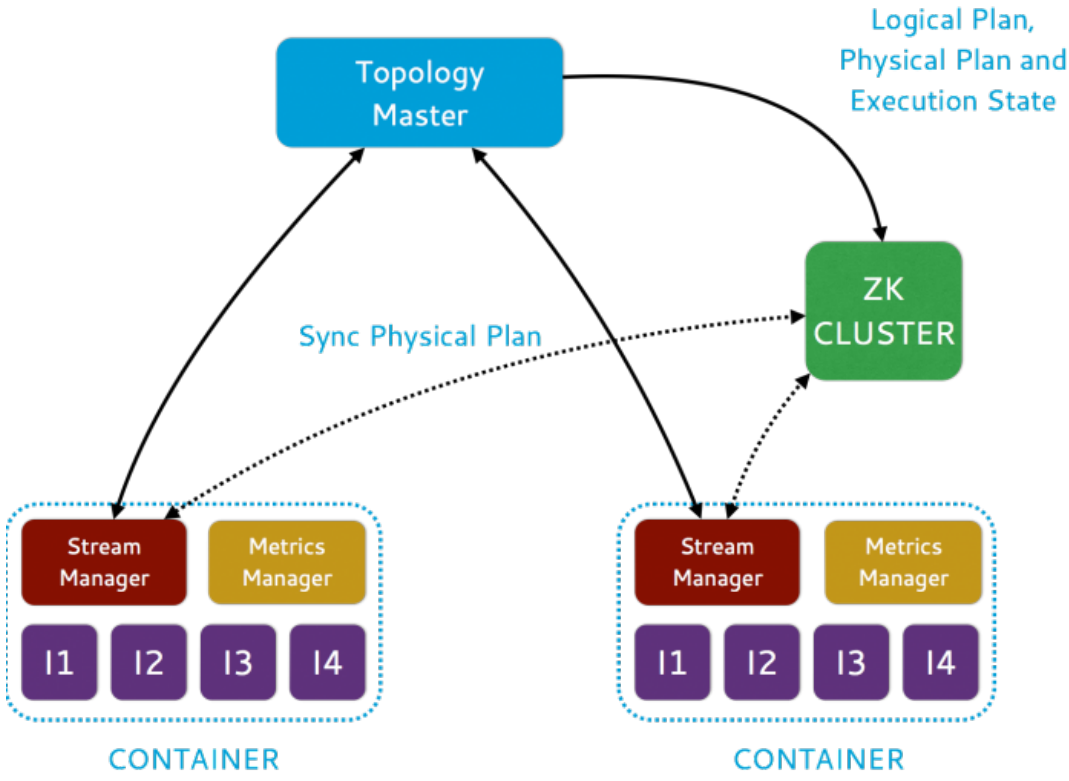


Figure: Topology Architecture

Twitter Heron

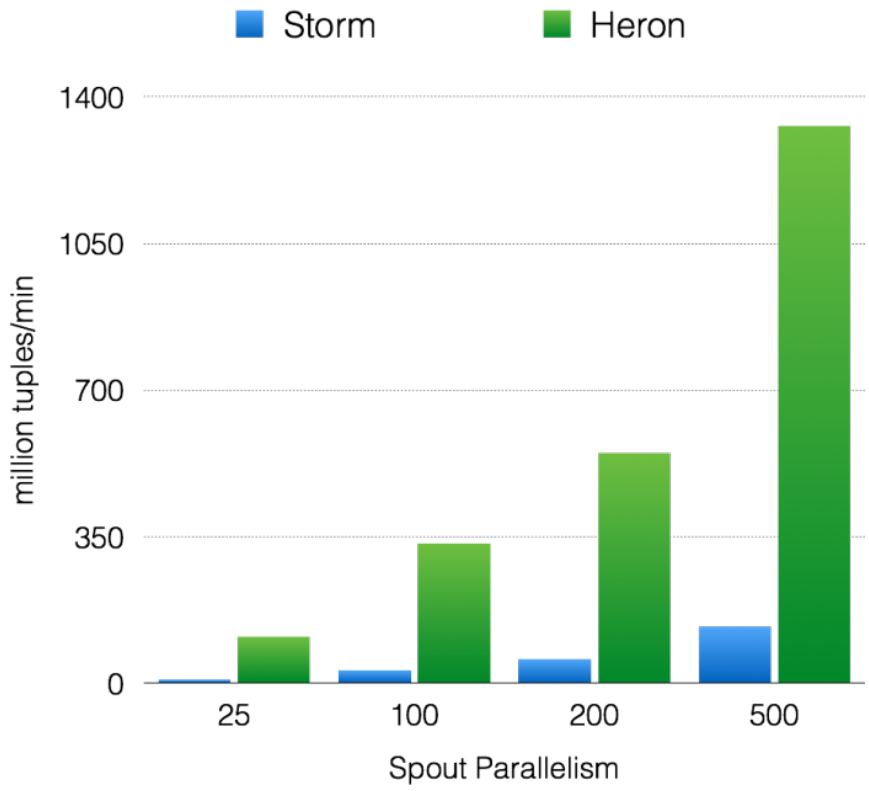


Figure: Throughput with acks enabled

Twitter Heron

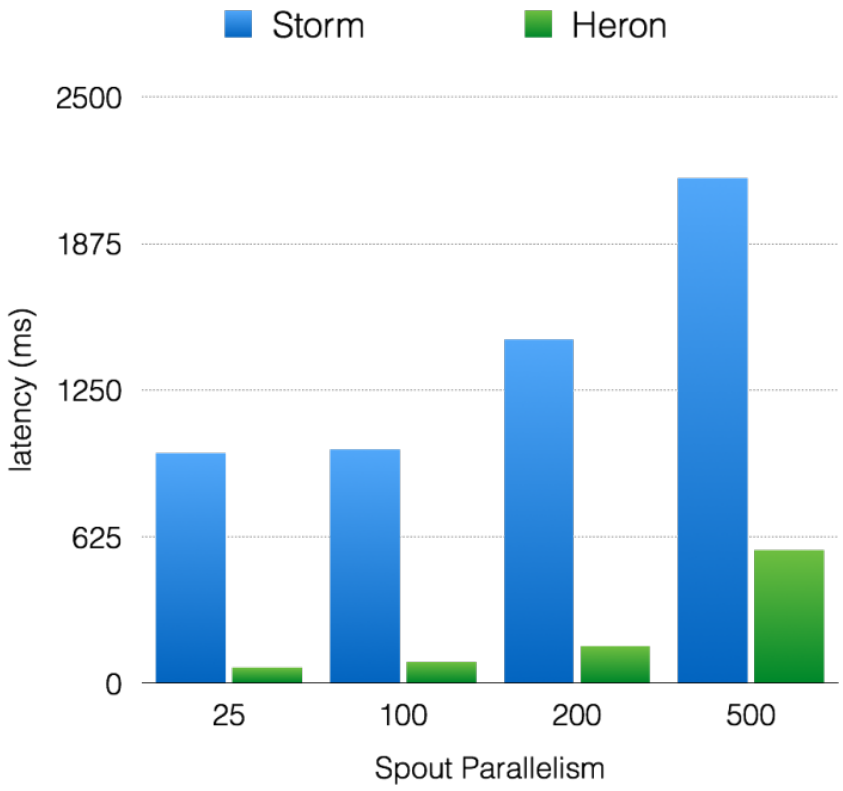


Figure: Latency with acks enabled

Twitter Heron

Twitter Heron Highlights:

1. Able to re-use the code written using Storm
2. Efficient in terms of resource usage
3. 3x reduction in hardware
4. Not open-source

Google Cloud DataFlow

Google 2004

There was need for an abstraction that hides many system-level details from the programmer.

Google 2004

There was need for an abstraction that hides many system-level details from the programmer.

MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a scalable, robust, and efficient manner.

Google June 2014

What is using Google right now?

Google June 2014

What is using Google right now?

“We don’t really use MapReduce anymore,”
The company stopped using the system “years ago.”

Google June 2014

What is using Google right now?

“We don’t really use MapReduce anymore,” The company stopped using the system “years ago.”

“Cloud Dataflow is the result of over a decade of experience in analytics,” “It will run faster and scale better than pretty much any other system out there.”

Google Cloud Data Flow

The processing model of Google Cloud Dataflow is based upon technology from

- **FlumeJava**(2010): Java library that makes it easy to develop, test, and run efficient data parallel pipelines.
- **MillWheel**(2013): framework for building low-latency data-processing applications

Google Cloud Data Flow

Cloud Dataflow consists of :

- A set of SDKs that you use to define data processing jobs:
  - **PCollection:** specialized collection class to represent pipeline data.
  - **PTransforms:** powerful data transforms, generic frameworks that apply functions across an entire data set
  - **I/O APIs:** pipeline read and write data to and from a variety of formats and storage technologies.
- A Google Cloud Platform managed service:
  - Google Compute Engine VMs, to provide job workers.
  - Google Cloud Storage, for reading and writing data.
  - Google BigQuery, for reading and writing data.

Google Cloud Data Flow

Service Features

- **Dynamic Optimization**: the Dataflow service constructs a directed graph of the job and optimizes the graph for the most efficient execution.
- **Resource Management**: it includes spinning up and tearing down Compute Engine resources, collecting logs, and communicating with Cloud Storage technologies.
- **Job Monitoring**: its interface shows the different stages of the data processing pipeline.
- **Native I/O Adapters for Cloud Storage Technologies**: to Cloud Platform storage systems such as Cloud Storage and BigQuery.

Google Cloud Data Flow Paper

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 of how to reconcile the tensions between these seemingly competing propositions, often resulting in disparate implementations and systems.

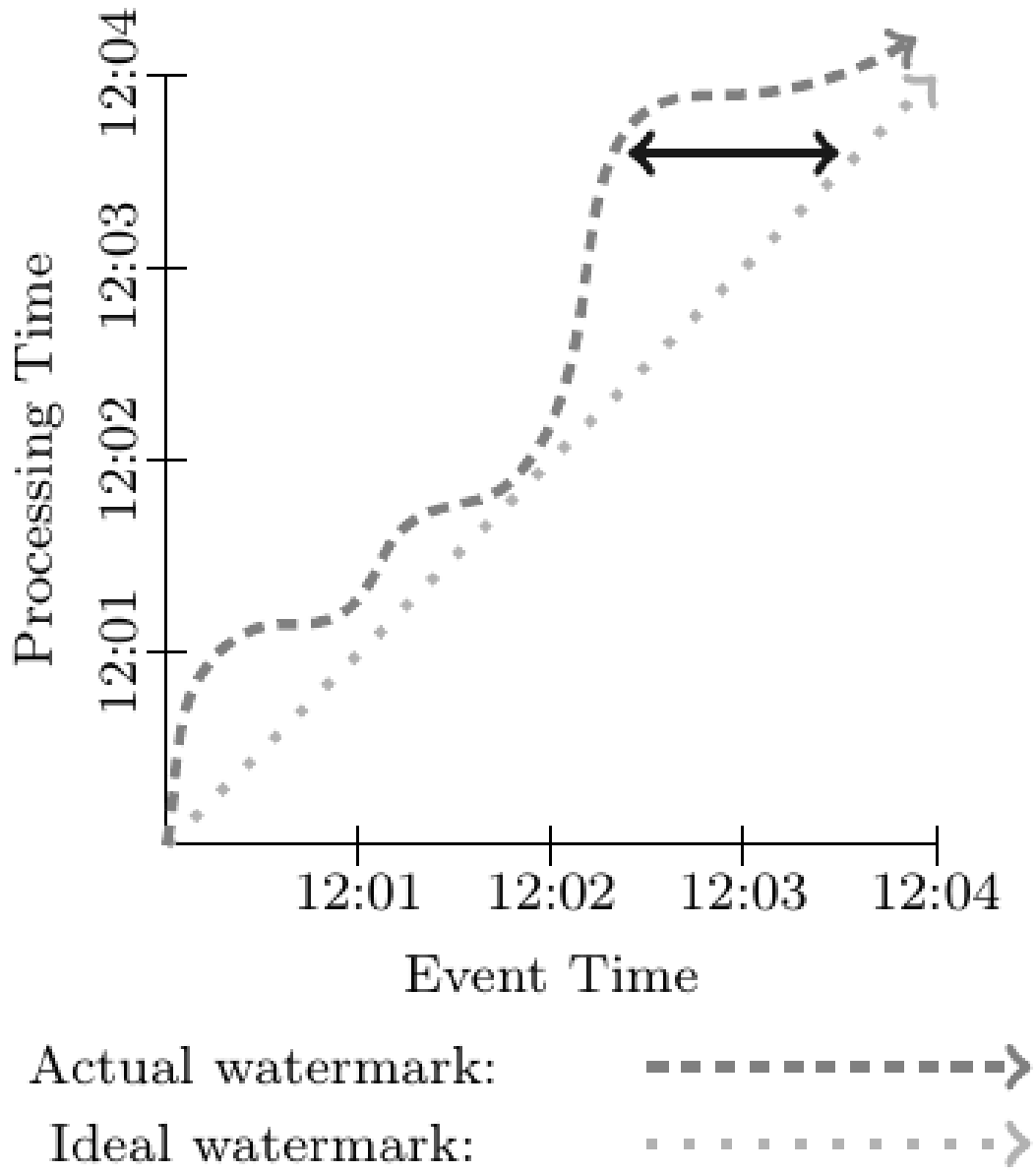
1. INTRODUCTION

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 greater value. Yet, existing models and systems still fall short in a number of common use cases.

Consider an initial example: a streaming video provider

Figure: VLDB 2015

Watermark


Processing Time
12:01
12:02
12:03
12:04
12:01
12:02
12:03
12:04
Event Time
Actual watermark:
Ideal watermark:

API

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

Streaming:

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

Example

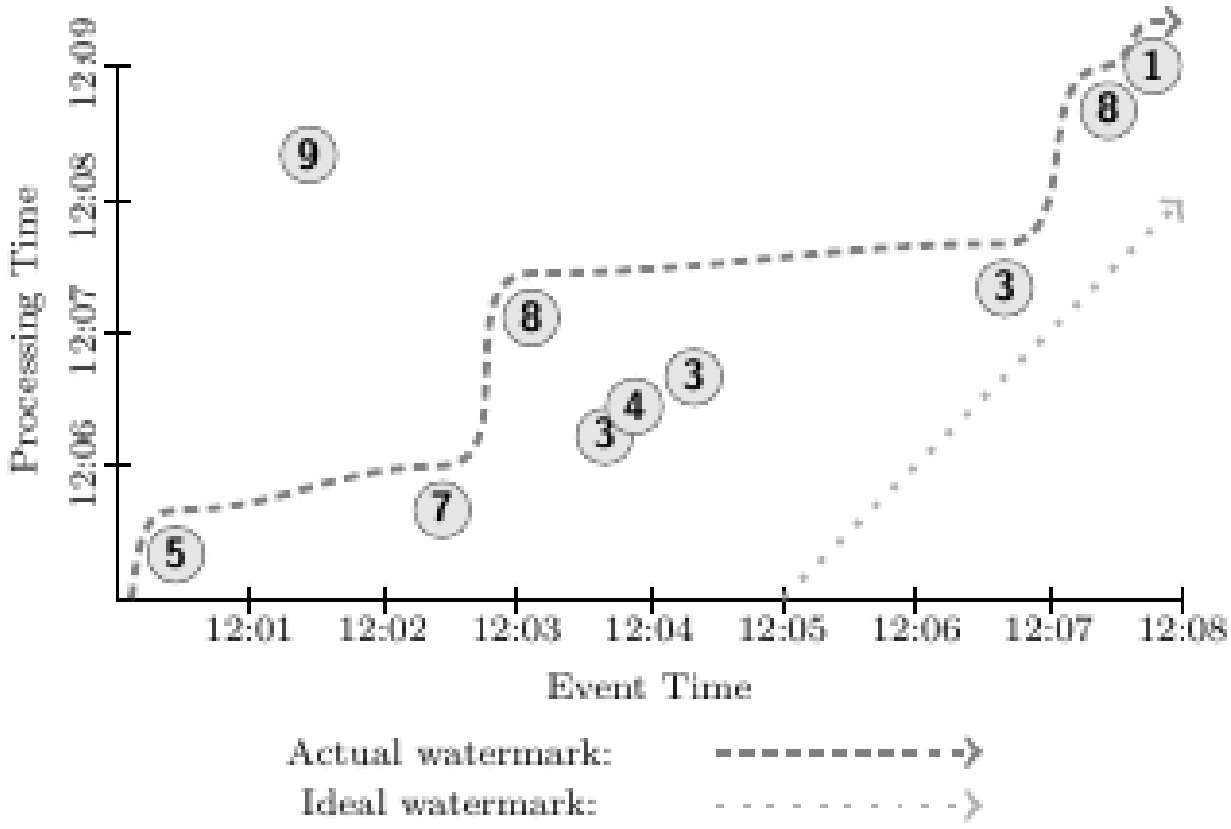


Figure: Example Inputs

Example

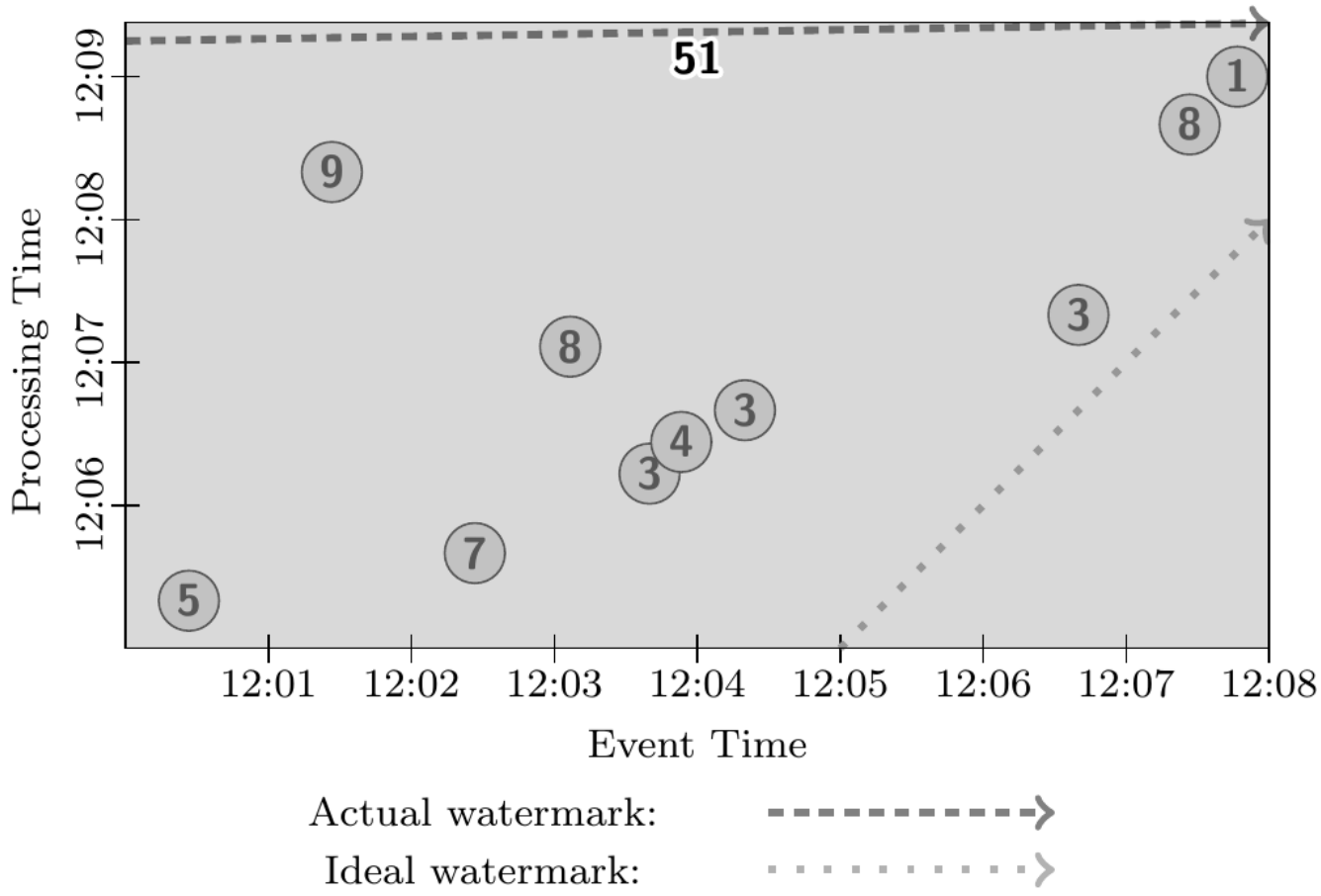


Figure: Standard Batch Execution

Example

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

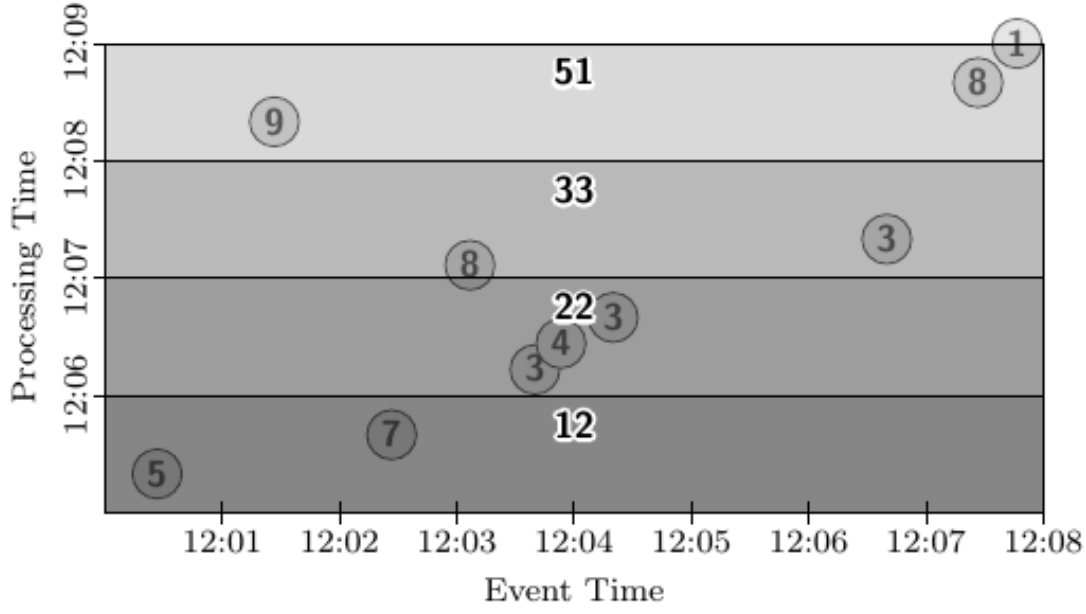


Figure: GlobalWindows, AtPeriod, Accumulating

Example

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

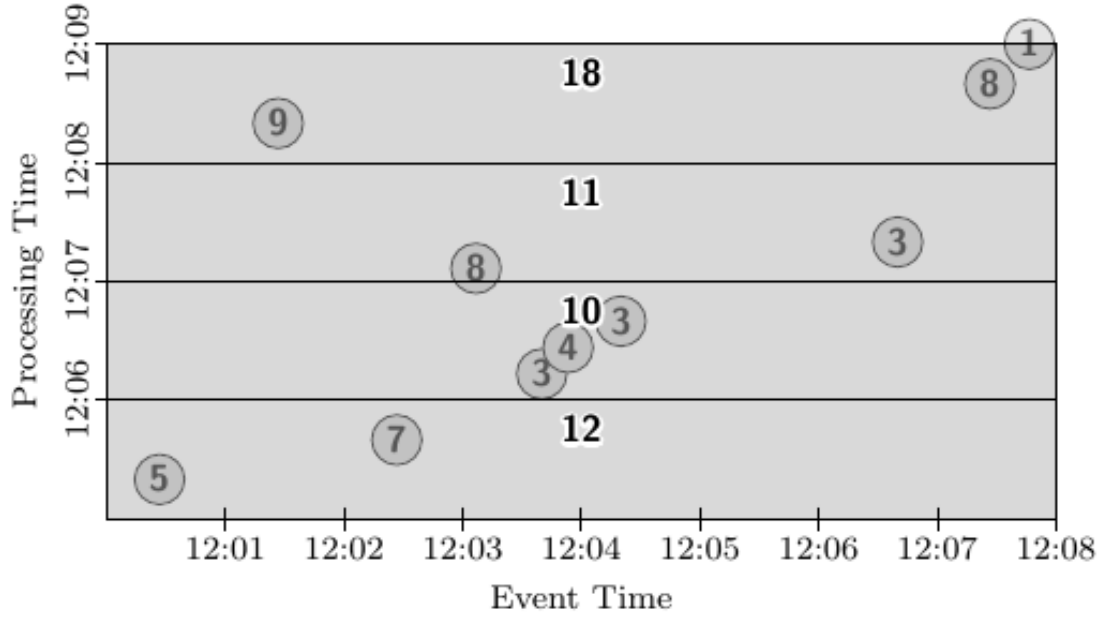


Figure: GlobalWindows, AtPeriod, Discarding

Example

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

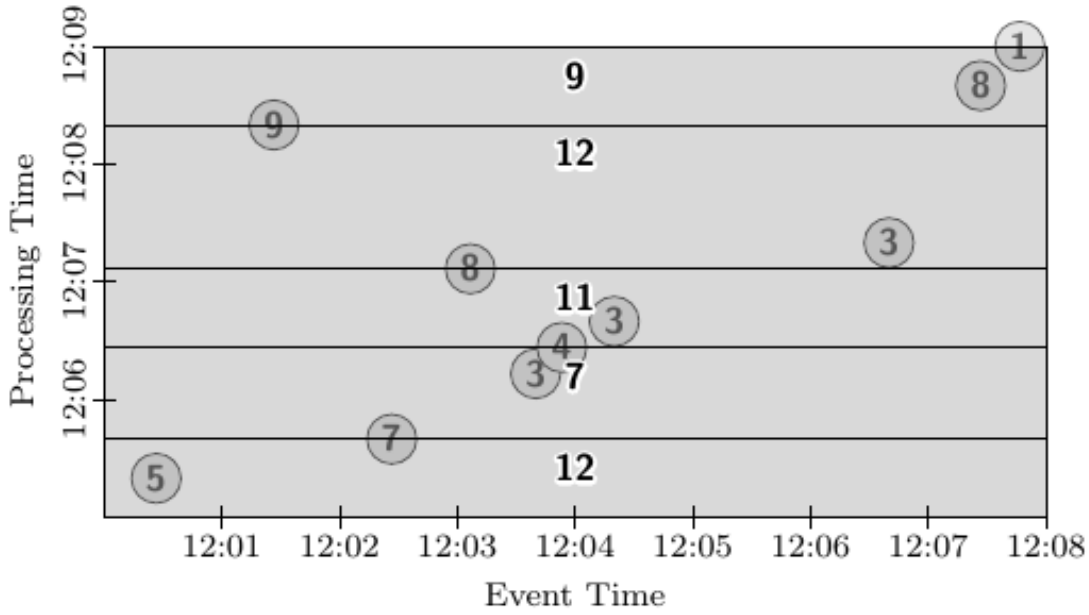


Figure: GlobalWindows, AtCount, Discarding

Example

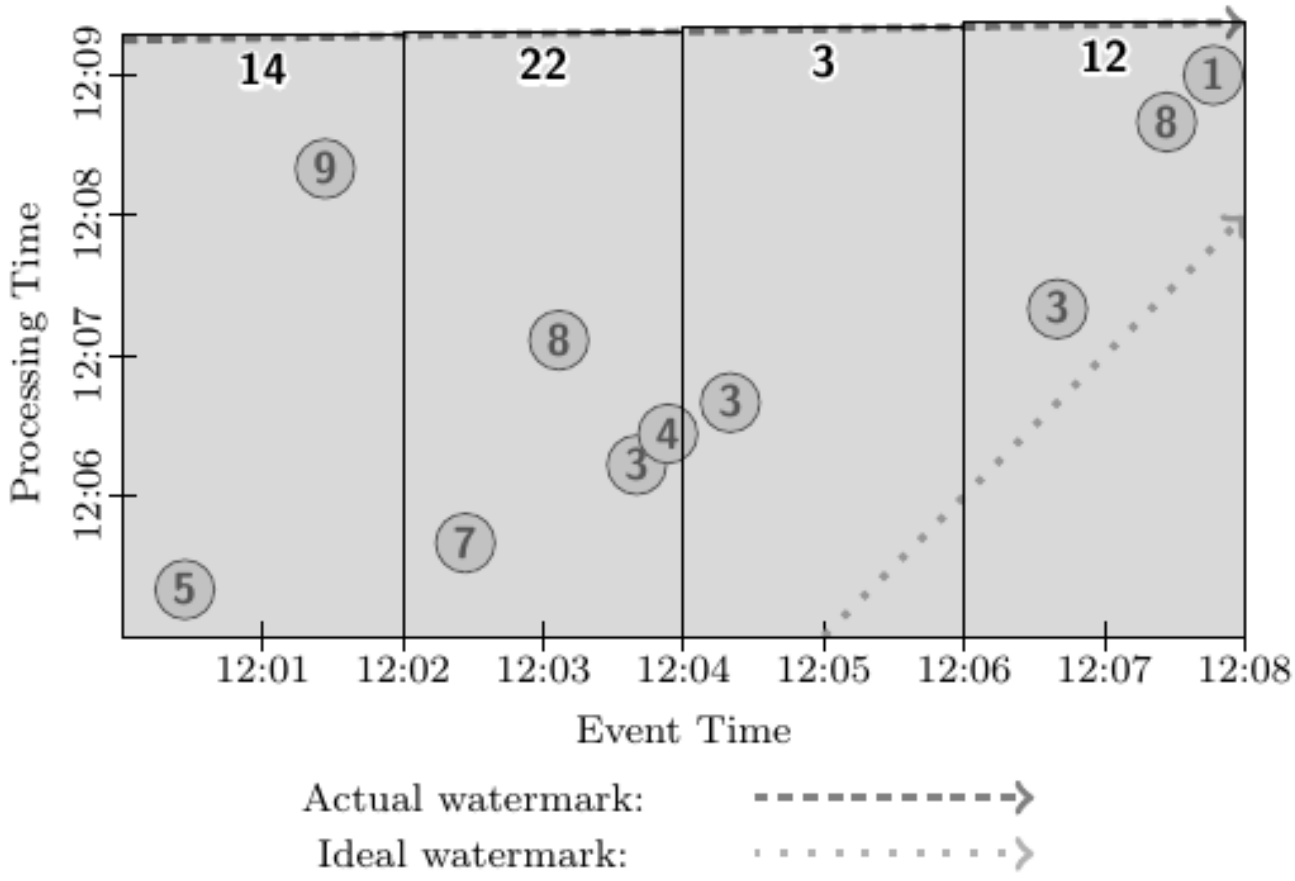


Figure: FixedWindows, Batch

Example

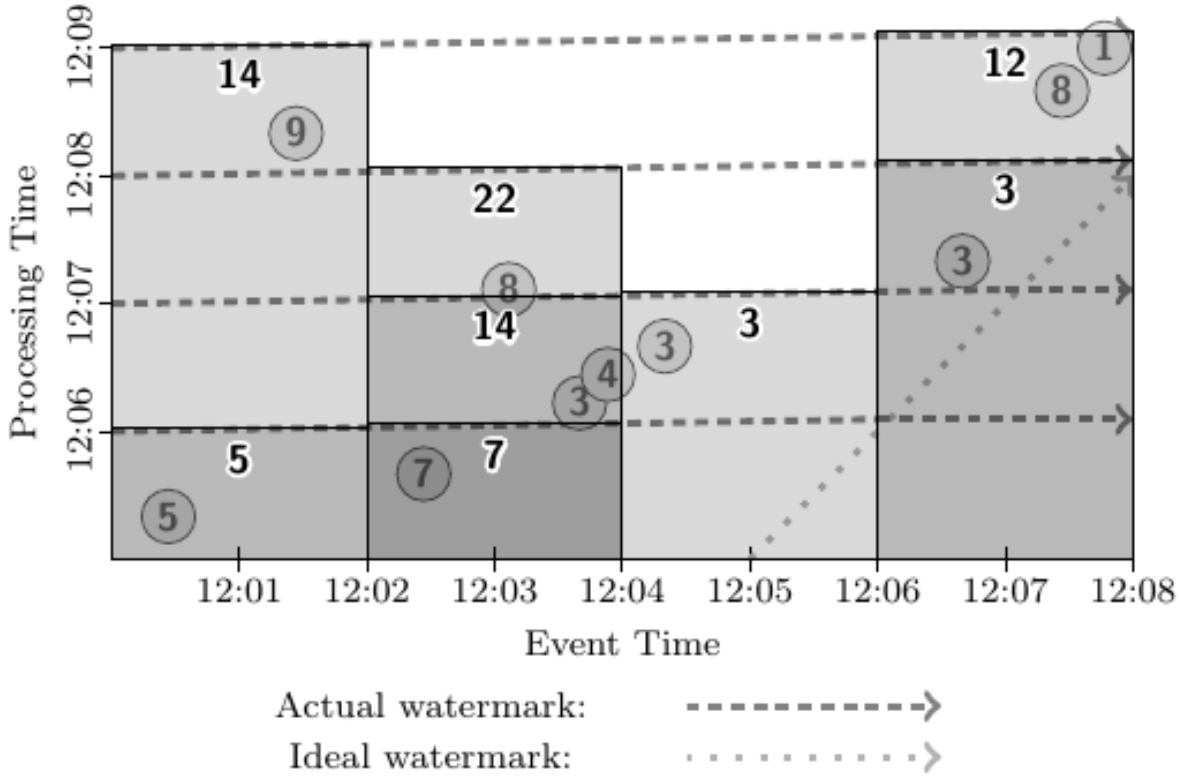


Figure: FixedWindows, Micro-Batch

Example

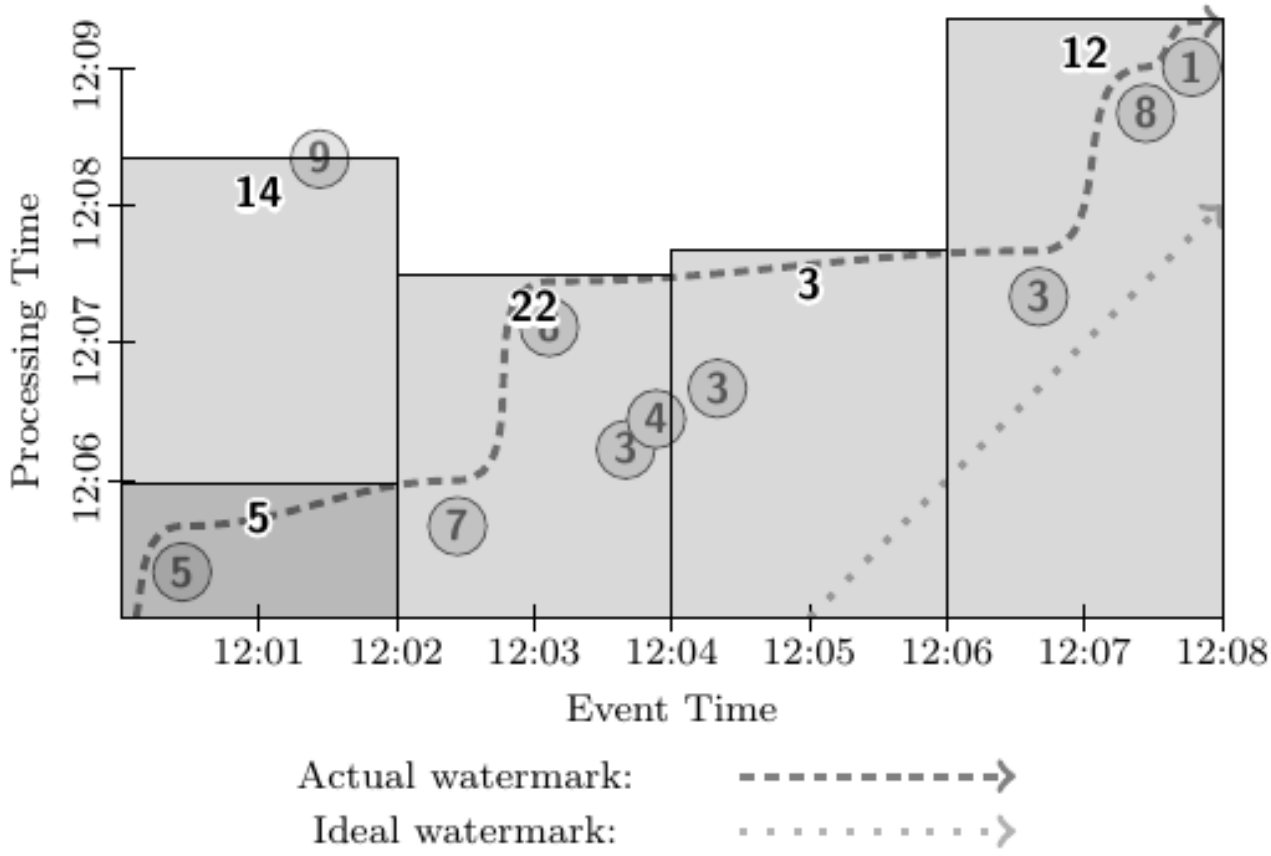
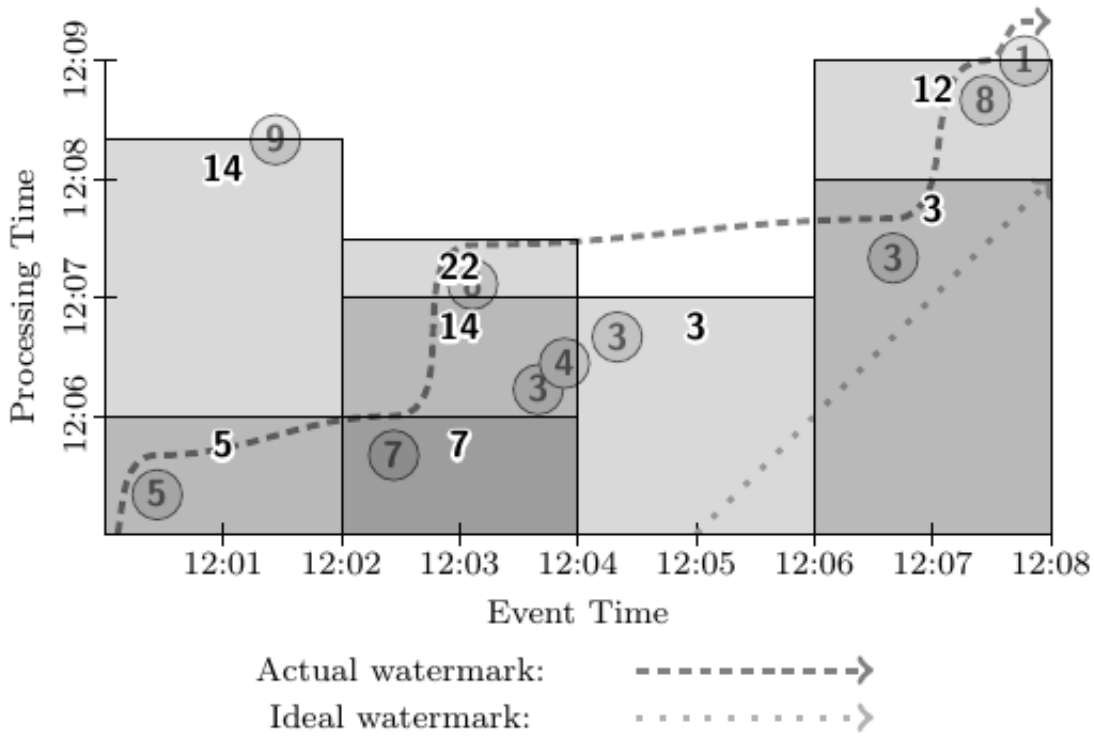


Figure: FixedWindows, Streaming

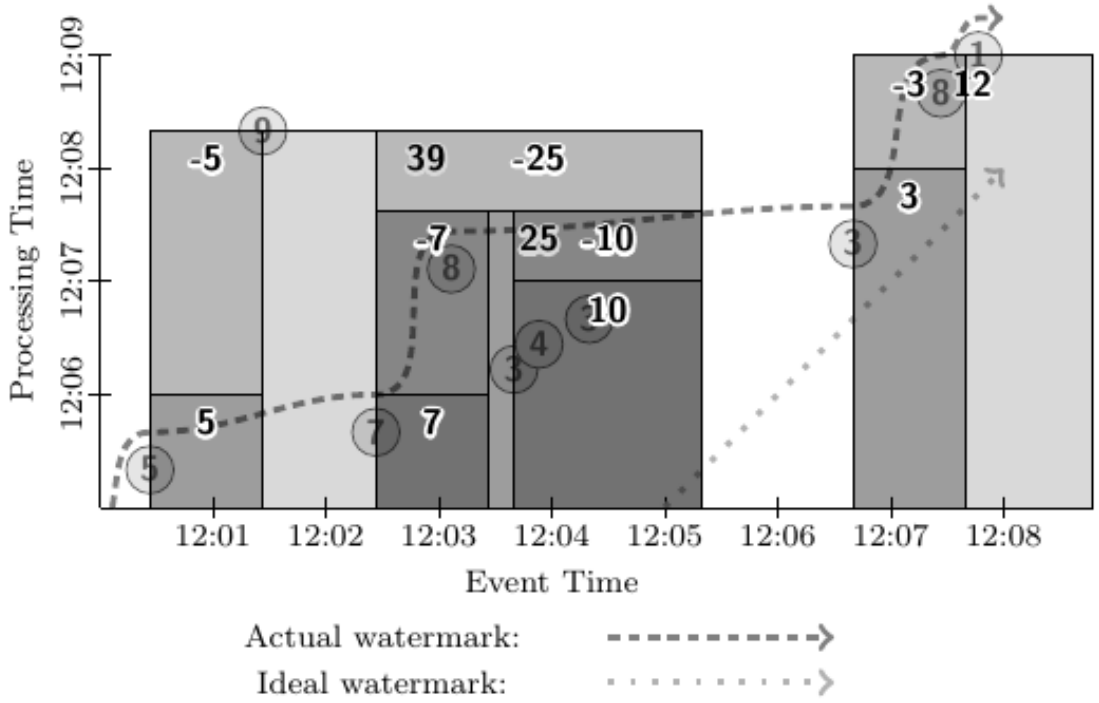
Example

```
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());
```



Example

```
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());
```



Google Cloud Data Flow

4. CONCLUSIONS

The future of data processing is unbounded data. Though bounded data will always have an important and useful place, it is semantically subsumed by its unbounded counterpart. Furthermore, the proliferation of unbounded data sets across modern business is staggering. At the same time, consumers of processed data grow savvier by the day, demanding powerful constructs like event-time ordering and unaligned windows. The models and systems that exist today serve as an excellent foundation on which to build the data processing tools of tomorrow, but we firmly believe that a shift in overall mindset is necessary to enable those tools to comprehensively address the needs of consumers of unbounded data.

Figure: Conclusions of the VLDB 2015 paper

Google Cloud Data Flow

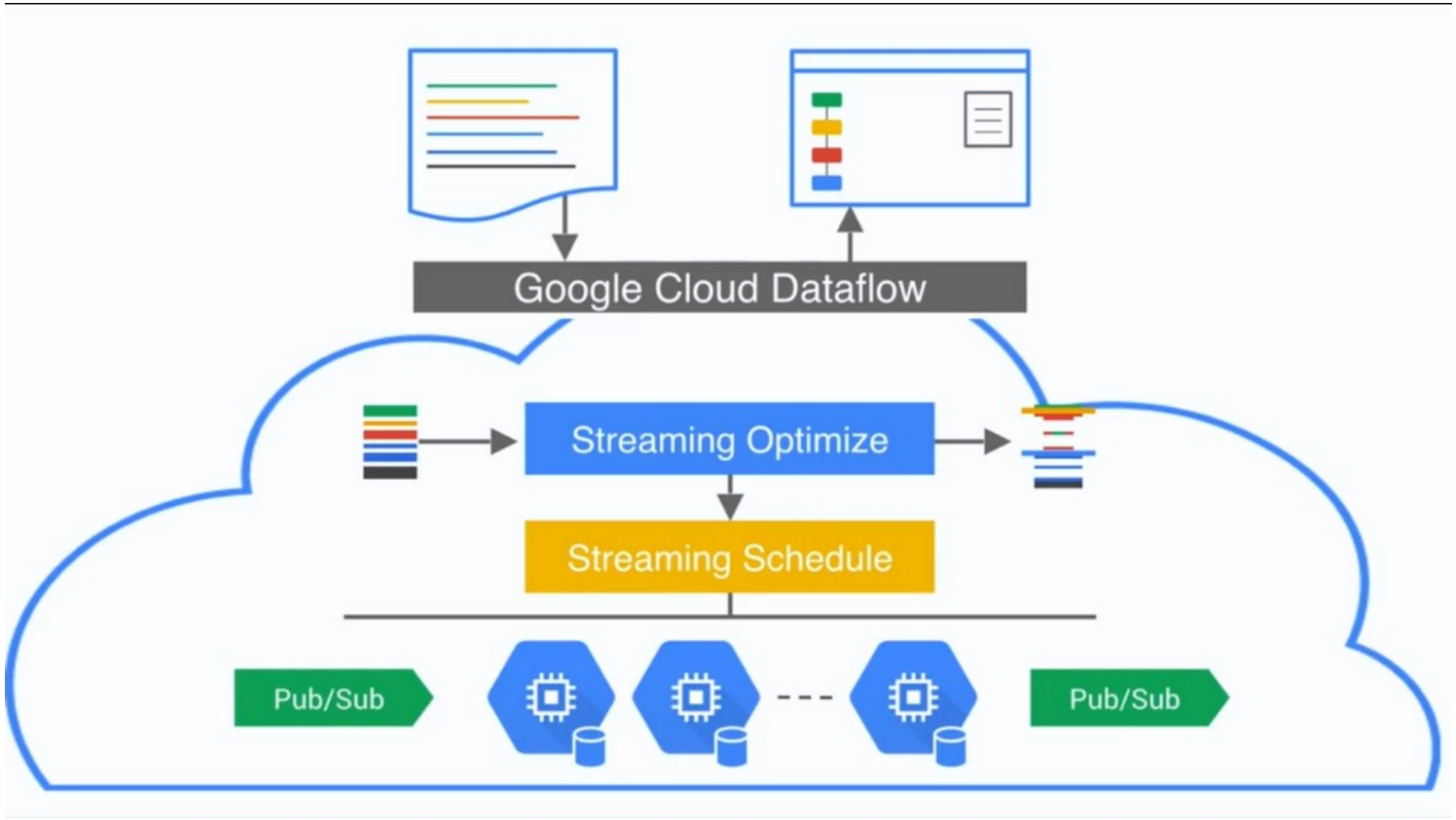


Figure: Architecture

Google Cloud Data Flow Summary

Cloud Dataflow code can run in:

- Cloud Dataflow runner for **Flink**
- Cloud Dataflow runner for **Spark**

Cloud Dataflow replaced **MapReduce**:

- It is based on FlumeJava and MillWheel, a stream engine as **Storm, Samza**
- It writes and reads to Google Pub/Sub, a service similar to **Kafka**