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. Fernandez-Moctezuma,
Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt* , Sam Whittle

*Not the Eric Schmidt you think...

T. Brady

Problem

- Unbounded, unordered datasets
  - Web logs
  - Mobile usage statistics
  - Sensor networks
- Users have complex requirements:
  - Event-time ordering
  - Windowing by features of the data
  - Low latency
- One can never fully optimize along all dimensions of correctness, latency, and cost.
- How do we reconcile these conflicting requirements?

Previous Work: Need for Data Processing

- Mapreduce, Hadoop, Pig, Hive, Spark enabled **scale**
- SQL Systems enabled
  - **Query systems**
  - **Windowing**
  - **Data Streams**
  - **Time Domains**
  - **Semantic Models**
- Spark streaming, Millwheel, Storm enabled **low-latency processing**

But something is missing

Performance: Many good solutions but none have everything we want

- **High Latency** - batch systems
- **Not Fault Tolerant at Scale** - Aurora, TelegraphCQ, Niagara, Esper
- **Fail on Correctness** - Pulsar, Storm, Samza (No Exactly once semantics)
- **Lack Expressiveness** - MillWheel and Spark Streaming (Need for high-level models)
- **Too Complex** - Lambda Architecture Systems (Need to maintain batch and stream)

Paradigm:

- Focus on **input data as something which at some point will become complete**
- Nearly all **distinguish batch and streaming**

Key Aim of Paper: Shift In Approach

“Fully embrace the assumption that we **never know if or when we have seen all of our data**, only that data will arrive, **old data may be retracted**, and the only way to make the problem tractable is via **principled abstractions** that allow the practitioner the **choice of appropriate tradeoffs between correctness, latency and cost**.”

“**Execution engine [should not] dictate system semantics**; properly designed and built batch, micro-batch, and streaming systems can all provide equal levels of correctness”

Contribution: The Dataflow Model

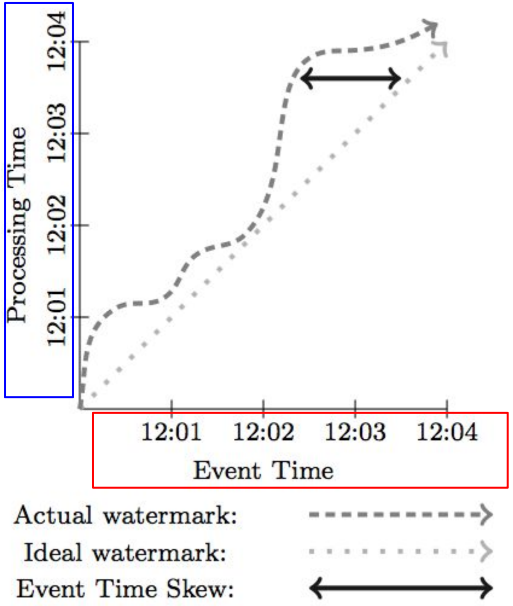
- A Unified Model allowing:
  - Event-time ordered results windowed by features of the data themselves
  - Unbounded, unordered data source
  - Correctness, Latency, and Cost tunable

- 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
  - **Where** in **processing time** they are materialized
  - **How** earlier **results relate** to later refinements

- Separates logic of data processing from the underlying physical implementation
  - choice of batch, micro-batch, or streaming engine → correctness, latency, and cost.

What time is it?

- **Event time** - time at which **event actually occurred**, never changes (e.g. when someone searched for “dog”)
- **Processing time** - time at which **event is observed** at a given point during processing
  - changes as moves event moves through pipeline
- No global clock



Primitives: **What** results are being computed

Two Core Transforms

- **ParDo** - generic parallel processing
  - Translates well to unbounded data

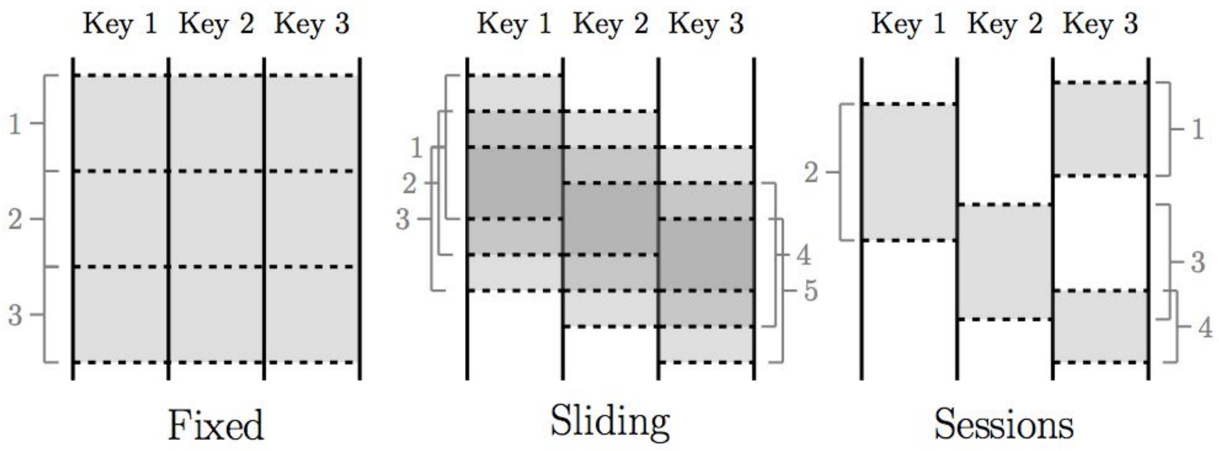
$$(fix, 1), (fit, 2)$$
$$\Big\downarrow\ ParDo(ExpandPrefixes)$$
$$(f, 1), (fi, 1), (fix, 1), (f, 2), (fi, 2), (fit, 2)$$

- **GroupByKey** - grouping (key, value) pairs
  - Not so easy with unbounded data

$$(f, 1), (fi, 1), (fix, 1), (f, 2), (fi, 2), (fit, 2)$$
$$\Big\downarrow\ GroupByKey$$
$$(f, [1, 2]), (fi, [1, 2]), (fix, [1]), (fit, [2])$$

Windowing Model: **Where** in event time results are computed

- Window: Time-based slices of dataset for processing as a group
- **Aligned** - applied across all data
- **Unaligned** - applied across given subset (e.g. per key)



Windowing Model: **Where** in event time results are computed

- Two operations
  - Set<Window> AssignWindows(T datum)

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

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

$$\begin{array}{l}(k, v_1, 12{:}00, [11{:}59, 12{:}01)),\\ (k, v_1, 12{:}00, [12{:}00, 12{:}02)),\\ (k, v_2, 12{:}01, [12{:}00, 12{:}02)),\\ (k, v_2, 12{:}01, [12{:}01, 12{:}03))\end{array}$$

  - Set<Window> MergeWindows(Set<Window> windows)
    - Typically redefine GroupByKey to GroupByKeyAndWindow

- Instead of (key,value) pairs, system is now handling (key, value, event time, window)

Windowing Model: GroupByKeyAndWindow

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

$$\downarrow \; \begin{aligned}&AssignWindows(\\ &\quad Sessions(30m))\end{aligned}$$

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

$$\downarrow \; DropTimestamps$$

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

$$\downarrow \; GroupByKey$$

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

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

$$\downarrow \; \begin{aligned}&MergeWindows(\\ &\quad Sessions(30m))\end{aligned}$$

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

$$\downarrow \; GroupAlsoByWindow$$

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

$$\downarrow \; ExpandToElements$$

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

Windowing Model: In Practice

- E.g. Window data into 30 minute sessions

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

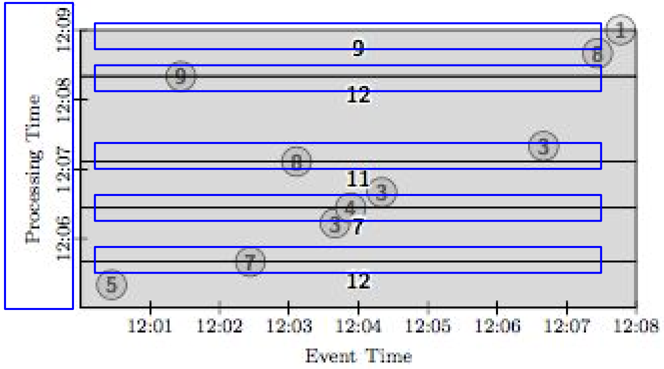
```
PCollection<KV<String, Integer>> input = IO.read(...);
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(Sessions.withGapDuration(
      Duration.standardMinutes(30))))
  .apply(Sum.integersPerKey());
```

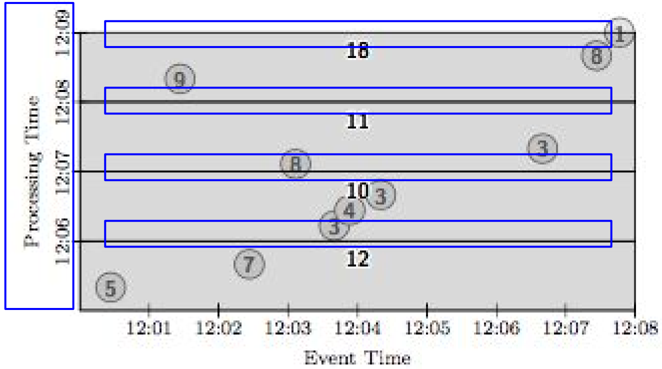
Triggering Model: **When** in processing time results are materialized

- Mechanism for stimulating the production of GroupByKeyAndWindow results in response to internal or external signals
- Allows you to control latency

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

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



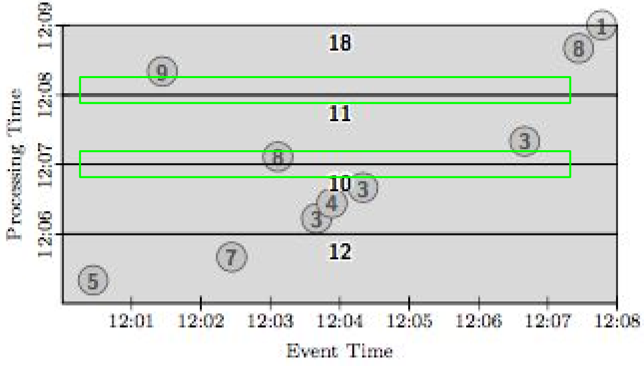


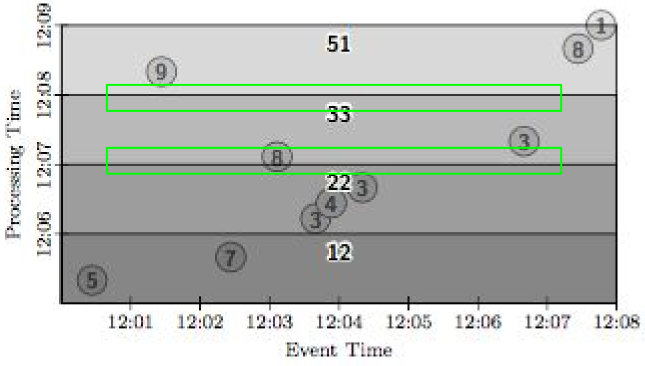
Incremental Model: **How** earlier results relate to later refinements

- Discarding
- Accumulating
- Accumulating and Retracting

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

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





Putting it all together

What results are being computed

Where in **event time** they are being computed

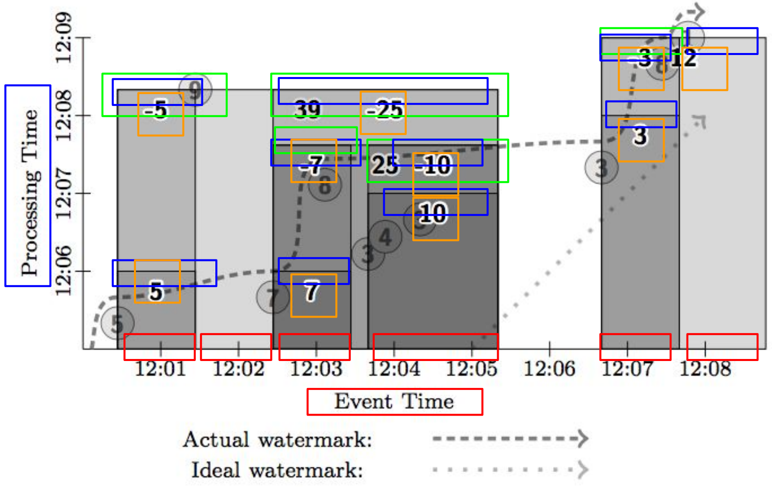
When in **processing time** they are materialized

How earlier results relate to **later refinements**

```
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());
```

“Session windowing with 1 minute timeout, enabling retractions”

- Sessions joined as more data received
- Results retracted as more data received



Contribution: The Dataflow Model

- A **Unified Model** allowing:
    - Event-time ordered results windowed by features of the data themselves
    - Unbounded, unordered data source
    - Correctness, Latency, and Cost tunable

- 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** logic of data processing from the underlying physical implementation
    - choice of batch, micro-batch, or streaming engine → correctness, latency, and cost.

- Scalable implementations on FlumeJava and Millwheel

How does it stack up?

- Low **latency**
  - via windowing and triggering
- **Scalable** and **Fault Tolerant**
  - Millwheel, FlumeJava
- **Correctness**
  - Incremental model with accumulations and retractions
- Greater **Expressiveness**
  - Windowing by features, complex triggering
- Reduced **Complexity**
  - Abstracted, Unified framework

But No Magic Bullet

- **That which was impractical** in existing systems **remains so**
  - Framework for parallel computation independent of underlying execution engine
  - Balance latency, correctness for a problem
- Aimed at ease of use, pragmatic, real world massive scale data processing
- Hard to **reason about the underlying performance.**
- What is the **Complexity** of these operations?
- What is the **Overhead**?
- Abstractions mean **less control**
  - Where is my computation happening?
  - But that’s the point of Dataflow Model...
  - Do I need to know?
- Paper doesn’t explore how this model is to be **implemented**
  - But open source is available

Thank You.
Questions?