The Dataflow Model
Problem

• How can we process unbounded data?
• Example: track user activity on a website
Key ideas

- **Windowing**
  - Fixed windows
  - Sliding windows
  - Sessions

- **Time domains**
  - Event time
  - Processing time

- **Triggers**
Contribution

• Dataflow API
  • Easily build pipelines with your choice of windowing, time domain, and trigger
  • Independent of execution engine
  • Choose batch, micro-batch, or streaming depending on tradeoffs
Windowing
Types of windows

- Fixed windows
- Sliding windows
- Sessions
Fixed windows
Sliding windows

Example: compute running average over past 5 minutes of data
Session windows

Example: YouTube viewing sessions
Time domains

• For many applications, windows should be based on “event time” (when the events actually occur)
  
  • Example: billing YouTube advertisers

• Lag, partitions, etc, might cause an event to be processed later than its event time
  
  • Processing time
Challenge: time skew
Goal: Event-time windows

Fixed windows

Session windows
Challenge: completion

• With event times, how does the system know if it has received all of the data in a window?

• Example: phones might watch YouTube videos (and ads) offline
Watermarks

- Heuristics that tell the system when it is likely to have received most of the data in a window
- Based on global progress metrics
- Watermarks are insufficient:
  - Late data might arrive behind the watermark
  - Watermark might be too slow due to one late datum and increase latency for the whole system
Incremental processing

- Difficult to get the single best result from a window

- Instead, let windows produce multiple results (improving incrementally over time)
Triggers

- Triggers specify when to output window results
  - at watermark
  - at percentile watermark
  - every minute, etc

- Triggers specify how to output results
  - discard previous window
  - accumulate
  - accumulate and retract

- Triggers are composable
Examples
A simple implementation of retraction processing requires non-deterministic operations, since the multiple results generated by a single key are also reversible and can support retractions. Dataflow supports updates (e.g. a database or key/value store). In such cases, it is important that the operations downstream from session creation that depend on the data had been processed, then advanced to infinity. An implementation that is also reversible can support retractions. Dataflow will generate incorrect results for those keys unless it is informed via a retraction that the event-time semantics, a copy of the previous value will be emitted first, followed by the new value as a normal datum. Retractions are necessary in pipelines with multiple serial grouping stages.

Figure 5 diagrams the ideal watermark and an example actual watermark. The X axis plots the data in event time (i.e. when the events actually occurred), while the Y axis plots the data in processing time (i.e. when the pipeline observes the data). The actual watermark is exemplified by the meandering path the actual watermark takes in Figure 5, represented by the darker, dashed line. Note also that the heuristic nature of distributed systems presents the ideal watermark, i.e. if there were no event-time skew and all events were processed by the system as they occurred, skew is a common occurrence; this is exemplified by the meandering path the actual watermark takes in Figure 5, represented by the darker, dashed line. Note also that the heuristic nature of distributed systems.

Figure 5: Example Inputs
PCollection<KV<String, Integer>> output = input
   .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
          .accumulating())
   .apply(Sum.integersPerKey());

Figure 7: GlobalWindows, AtPeriod, Accumulating
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
      .discarding())
  .apply(Sum.integersPerKey());

**Figure 8: GlobalWindows, AtPeriod, Discarding**
Let's run this pipeline under the three execution engines: batch, micro-batch, streaming
Finally, it arrives, it causes the first window (for event-time would eventually be occupied by the 9. Hence, once the watermark to proceed past the point in event time that yet been injected, and thus, having observed the 5, allowed partition, etc.), the system did not realize that datum had not value 9 is actually late relative to the watermark. For what- engine, as in Figure 12. Most windows are emitted when the last round. This provides a very nice mix of latency and outputs for all windows whose contents had changed since watermark for every micro-batch round, and corresponding ist for that period). We would thus end up with a new to the end of time instantaneously, since no data would ex- of time (technically jumping from the end time of the batch would start at the beginning of time and advance to the end repeat. Each time, the watermark for the current batch would gather input data for one minute, process them, and have to be a bounded one, so as with the classic batch ex- Given our current implementation, the data source would execution of this pipeline would look like on a batch engine. Next, consider this pipeline executed on a streaming en- Now imagine executing a micro-batch engine over this Figure 11: FixedWindows, Micro-Batch

![Figure 11: FixedWindows, Micro-Batch](image)

Figure 10: FixedWindows, Batch

![Figure 10: FixedWindows, Batch](image)
Finally arrives, it causes the first window (for event-time)
would eventually be occupied by the 9. Hence, once the
the watermark to proceed past the point in event time that
yet been injected, and thus, having observed the 5, allowed
tition, etc.), the system did not realize that datum had not
watermark passes them. Note however that the datum with
value 9 is actually late relative to the watermark. For what-
gine, as in Figure 12. Most windows are emitted when the
outputs for all windows whose contents had changed since
watermark for every micro-batch round, and corresponding
to the end of time instantaneously, since no data would ex-
would start at the beginning of time and advance to the end
repeat. Each time, the watermark for the current batch
would gather input data for one minute, process them, and
data source with one minute micro-batches. The system
next, consider this pipeline executed on a streaming en-
Now imagine executing a micro-batch engine over this

Figure 11: FixedWindows, Micro-Batch
Finally, if a late datum
finally arrives, it causes the first window (for event-time
would eventually be occupied by the 9. Hence, once the
the watermark to proceed past the point in event time that
yet been injected, and thus, having observed the 5, allowed
tition, etc.), the system did not realize that datum had not
time (technically jumping from the end time of the batch
would start at the beginning of time and advance to the end
Round. This provides a very nice mix of latency and
outputs for all windows whose contents had changed since
another stage. We would end up with a new
of time instantaneously, since no data would ex-
tary, network par-
removal correctness, as in Figure 11:
Given our current implementation, the data source would
 execution of this pipeline would look like on a batch engine.
Now imagine executing a micro-batch engine over this
Figure 11: FixedWindows, Micro-Batch
Figure 10: FixedWindows, Batch
Actual watermark:
Ideal watermark:
Event Time
Processing Time
Figure 12: FixedWindows, Streaming

Actual watermark: \\
Ideal watermark: \\

Figure 12: FixedWindows, Streaming
Finally arrives, it causes the first window (for event-time would eventually be occupied by the 9. Hence, once the watermark to proceed past the point in event time that yet been injected, and thus, having observed the 5, allowed partition, etc.), the system did not realize that datum had not ever reason (mobile input source being o...ime instantaneously, since no data would ex-

...of time (technically jumping from the end time of the batch would start at the beginning of time and advance to the end repeat. Each time, the watermark for the current batch would gather input data for one minute, process them, and data source with one minute micro-batches. The system

Now imagine executing a micro-batch engine over this...with windows being emitted as the simulated watermark ad-

...process the data in event-time order, have to be a bounded one, so as with the classic batch ex-

...would look like on a batch engine.

Next, consider this pipeline executed on a streaming en-

...goals we set out to achieve with this model.

...goals we set out to achieve with this model. (as well as the choice of micro-batch size) really becomes just a matter of latency versus cost, which is exactly one of the (micro-batch and streaming engines, the choice between them being processed in small batches. Given strongly-consistent data are accumulated in windows as they arrive instead of somewhat better latency than the micro-batch pipeline, since the watermark actually passes, as in Figure 13. This yields time-based triggers to provide us with regular updates until of our windows, we can add in some additional, processing-

...of watermarks being too slow from Section 2.3.

Having to wait for the watermark to advance; this is the case noticeably worse than the micro-batch system, on account of the late datum. But the overall latency of results is no-

...output per window, with a single refinement in the case of...and streaming engines, the choice between them

...data source with one minute micro-batches. The system...gination as the aggregation operation, which we will maintain video sessions requirements (modulo the use of summa-

...the video sessions requirements (modulo the use of summa-

...apply(Sum.integersPerKey());

...Figure 13: FixedWindows, Streaming, Partial

...Figure 12: FixedWindows, Streaming

...Figure 11: FixedWindows, Micro-Batch

...Figure 10: FixedWindows, Batch

Figure 13: FixedWindows, Streaming, Partial
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());

Figure 14: Sessions, Retracting