



Scalable Stream Processing - Storm, MillWheel, and Google Dataflow

Amir H. Payberah
payberah@kth.se
01/10/2018

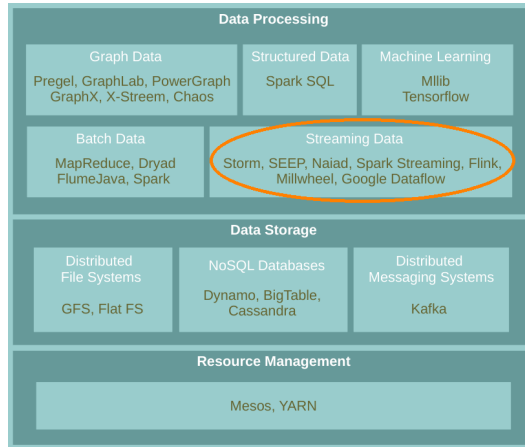




The Course Web Page

<https://id2221kth.github.io>

Where Are We?





Stream Processing Systems Design Issues

- ▶ Continuous vs. micro-batch processing
- ▶ Record-at-a-Time vs. declarative APIs



Outline

- ▶ Storm
- ▶ MillWheel
- ▶ Google Dataflow

Storm

Contribution

- ▶ **Storm** is a **real-time distributed stream** data processing engine.
- ▶ Design issues
 - **Continuous** vs. micro-batch processing
 - **Record-at-a-Time** vs. declarative APIs



► Stream

- Unbounded sequence of tuples.



► Tuple

- Core unit of data.
- Immutable set of key/value pairs.



Storm Spouts and Bolts

► Spouts

- Source of streams.
- Wraps a streaming data source and **emits** tuples.



► Bolts

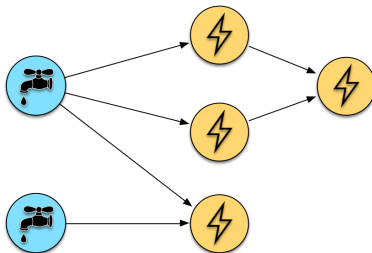
- Core **functions** of a streaming computation.
- Receive tuples and **do stuff**.
- Optionally emit **additional tuples**.



Storm Topology

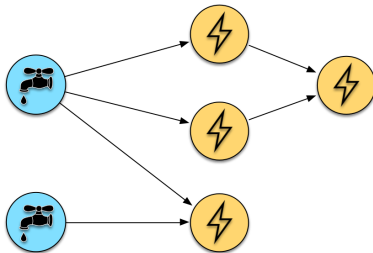
► Topology

- DAG of spouts and bolts.
- Analogous to a job in Spark/MapReduce job.
- One key difference is that a Spark/MapReduce job eventually finishes, whereas a topology runs forever.



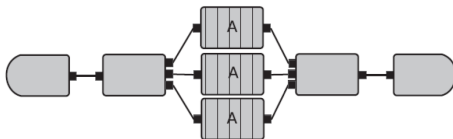
► Task

- Each **spout** or **bolt** executes as many **tasks** across the cluster.
- A **task** performs the **actual data processing**.



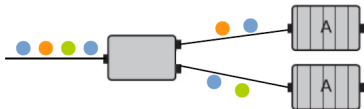
Stream Groupings (1/3)

- ▶ Stream groupings define how to send tuples from one set of tasks to another set of tasks.
- ▶ Data parallelism



Stream Groupings (2/3)

- **Shuffle** grouping: **randomly partitions** the tuples.



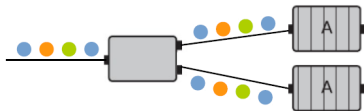
-

-

-
- A diagram showing a single input line on the left that splits into two parallel paths. Each path contains a gray rectangular component labeled 'A'. After the components, the two paths rejoin into a single output line on the right. Four colored circles (blue, orange, green, blue) are shown entering from the left, and four colored circles (blue, orange, green, blue) are shown exiting to the right, indicating a one-to-one correspondence between inputs and outputs.

Stream Groupings (3/3)

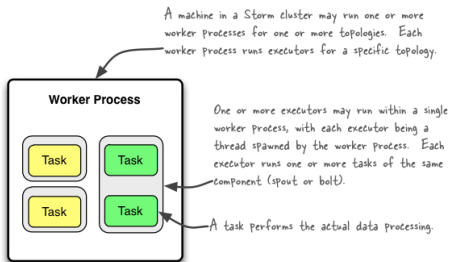
- **All** grouping: replicates the entire stream to all the consumer tasks.



- **Global** grouping: sends the entire stream to one of the bolt's tasks.

What Makes a Running Topology

- **Main entities** that are used to **run a topology** in a **Storm cluster**:
 - **Worker** processes: each worker executes a **subset of a topology**.
 - **Executors** (threads): a thread that is **spawned by a worker** process, and run **one or more tasks** for the **same component** (spout or bolt).
 - **Tasks**: performs the **actual data processing**.



[<http://storm.apache.org/releases/1.0.6/Understanding-the-parallelism-of-a-Storm-topology.html>]

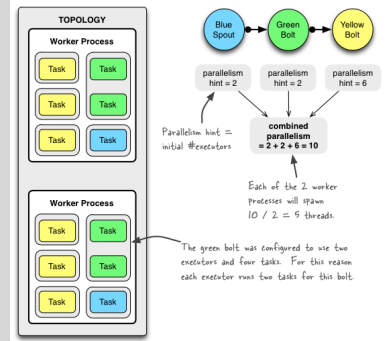
Example of A Running Topology

```
Config conf = new Config();
// use 2 worker processes
conf.setNumWorkers(2);

// set parallelism hint to 2
topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2);
topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
    .setNumTasks(4)
    .shuffleGrouping("blue-spout");

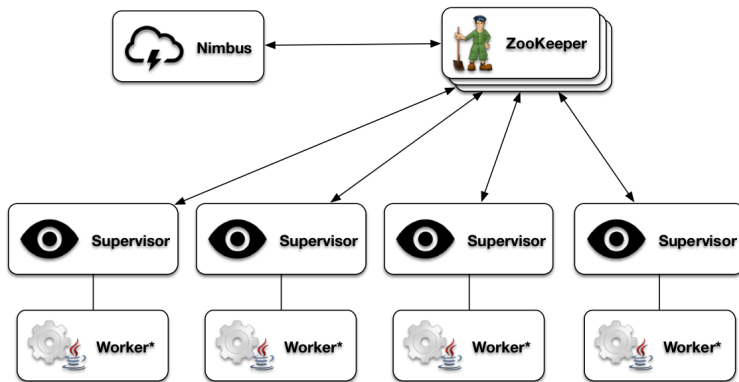
topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
    .shuffleGrouping("green-bolt");

StormSubmitter.submitTopology("mytopology", conf,
    topologyBuilder.createTopology());
```



Storm Architecture

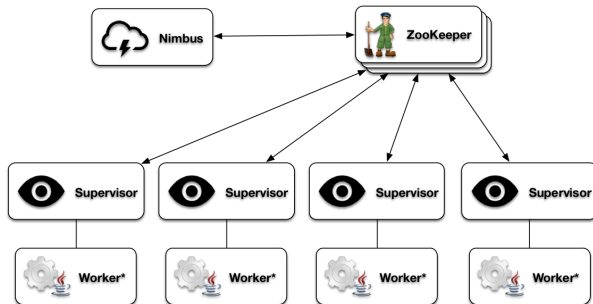
Storm Architecture (1/5)



Storm Architecture (2/5)

► Nimbus

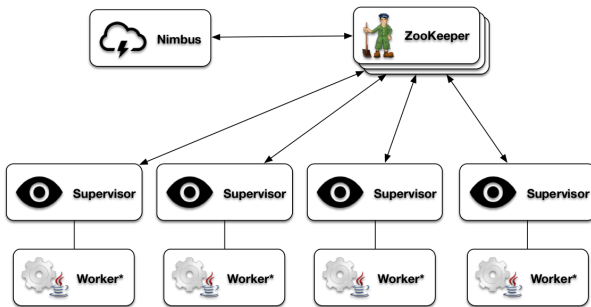
- The **master** node.
- **Clients** submit **topologies** to it.
- Responsible for **distributing and coordinating** the **execution of the topology**.



Storm Architecture (3/5)

► Worker nodes

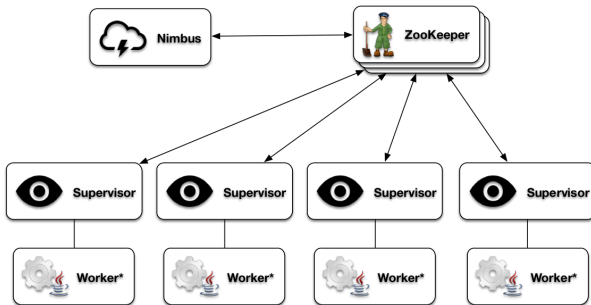
- Each **worker node** runs **one or more worker processes**.
- Each **worker process** runs a **JVM**, in which it runs **one or more executors**.
- **Executors** are made of **one or more tasks**, where the actual work for a **bolt** or a **spout** is done in the task.



Storm Architecture (4/5)

► Supervisor

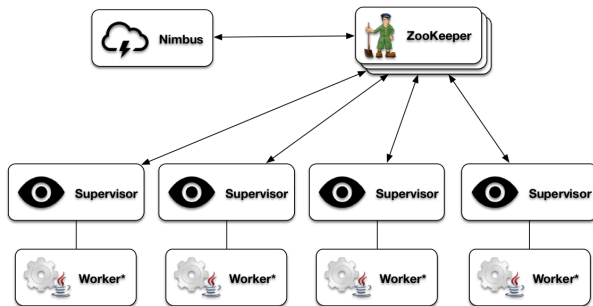
- Each **worker node** runs a **supervisor**.
- It receives **assignments** from **Nimbus** and **spawns workers** based on the assignment.
- Contact Nimbus with a **periodic heartbeat** protocol, advertising the **topologies** that they are currently running, and any **vacancies** that are available to run more topologies.



Storm Architecture (5/5)

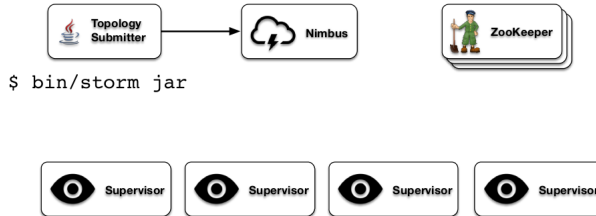
► Zookeeper

- Nimbus uses a combination of the **local disk(s)** and **Zookeeper** to store **state** about the topology.



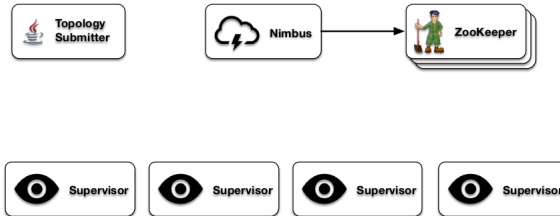
Storm Deployment (1/5)

- Topology submitter **uploads topology** to **Nimbus**.



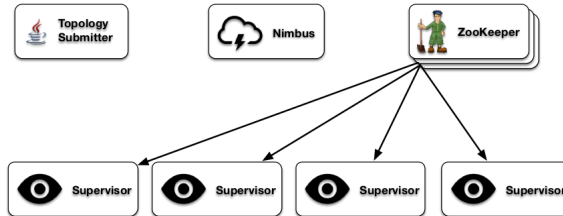
Storm Deployment (2/5)

- Nimbus calculates assignments and sends to Zookeeper.



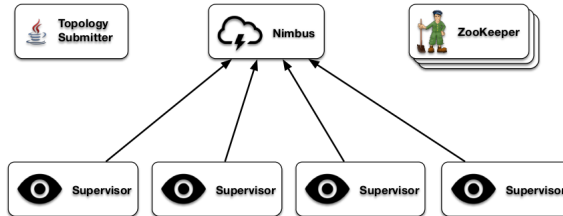
Storm Deployment (3/5)

- Supervisor nodes **receive assignment** information via **Zookeeper** watches.



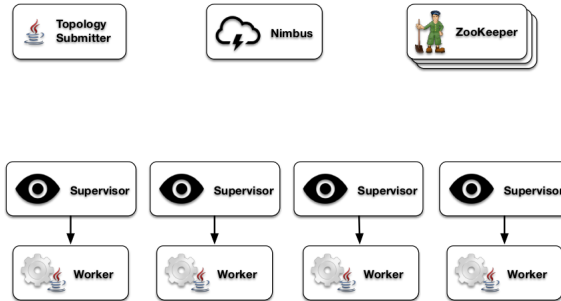
Storm Deployment (4/5)

- Supervisor nodes **download topology** from **Nimbus**.



Storm Deployment (5/5)

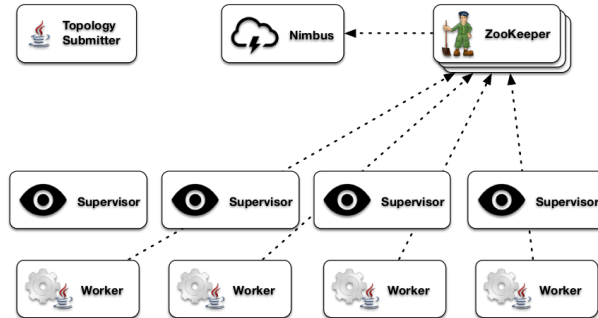
- Supervisors **spawn workers** (JVM processes) to **start the topology**.



Reliable Processing

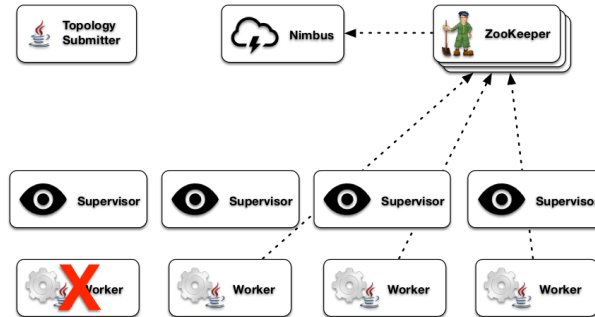
Fault Tolerance (1/4)

- Workers **heartbeat** back to Supervisors and Nimbus via **ZooKeeper**, as well as locally.

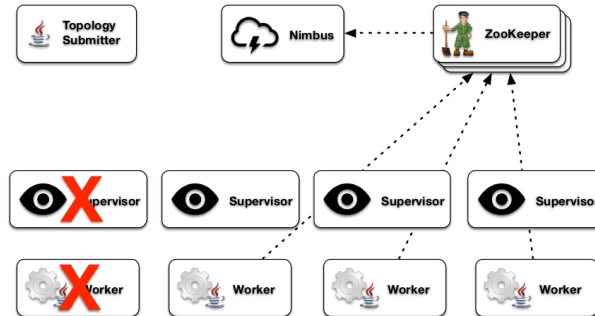


Fault Tolerance (2/4)

- ▶ If a **worker dies** (fails to heartbeat), the Supervisor will **restart** it.
- ▶ If a **worker dies repeatedly**, Nimbus will **reassign** the work to other nodes in the cluster.

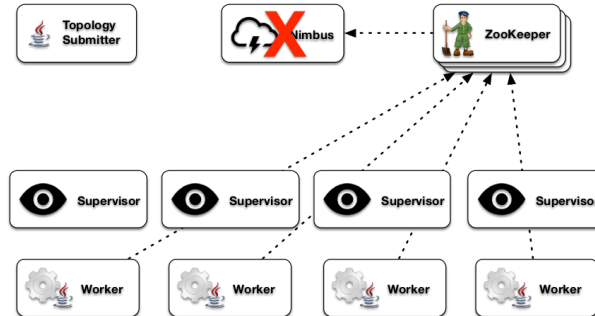


- If a **supervisor node dies**, Nimbus will **reassign** the work to other nodes.



Fault Tolerance (4/4)

- If **Nimbus** dies, topologies will **continue to function normally**, but won't be able to perform reassignments.



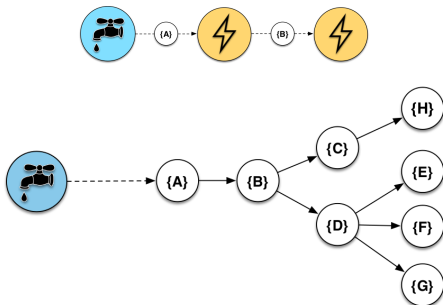


Reliable Processing (1/6)

- ▶ Storm provides **two types** of semantic guarantees:
 - **At most once**: each tuple is **either processed once, or dropped** in the case of a failure.
 - **At least once** (**reliable processing**): it guarantees that each tuple that is input to the topology will be **processed at least once**.

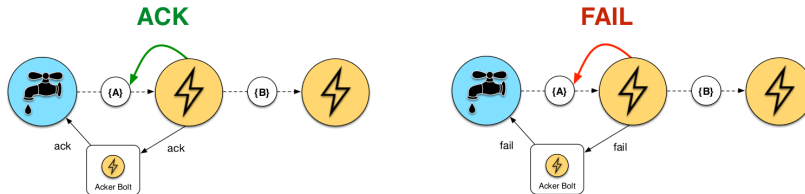
Reliable Processing (2/6)

- ▶ Bolts may emit tuples **anchored** to the ones they received.
 - Tuple **B** is a descendant of Tuple **A**.
- ▶ Multiple anchorings form a **Tuple tree**.



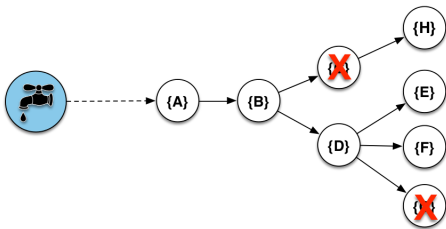
Reliable Processing (3/6)

- ▶ Bolts can **acknowledge** that a tuple has been processed **successfully**.
- ▶ Acks are delivered via a **system-level bolt**.
- ▶ Bolts can also **fail** a tuple to trigger a spout to **replay** the original.



Reliable Processing (4/6)

- ▶ Any **failure** in the **tuple tree** will trigger a **replay** of the original tuple.
- ▶ How to track a **large-scale** tuple tree efficiently?



Reliable Processing (5/6)

- ▶ Tuples are assigned a 64-bit message id at spout.
- ▶ Emitted tuples are assigned new message ids.
- ▶ These message ids are XORed and sent to the acker bolt along with the original tuple message id.
- ▶ When the XOR checksum goes to zero, the acker bolt sends the final ack to the spout that admitted the tuple, and the spout knows that this tuple has been fully processed.
 - $a \oplus (a \oplus b) \oplus c \oplus (b \oplus c) == 0$



Reliable Processing (6/6)

- ▶ It is possible that due to **failure**, some of the **XOR checksum** will never go to zero.
- ▶ The spout initially assigns a **timeout parameter** to each tuple.
- ▶ The **acker bolt** keeps track of this timeout parameter, and if the XOR checksum does **not become zero** before the **timeout**, the tuple is considered to have failed.
 - The data source will **replay** it back in the subsequent iteration.

Word Count in Storm

Word Count - Random Sentence Spout

```
public static class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }

    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[]{ ... };
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sentence"));
    }
}
```

Word Count - Split Sentence Bolt

```
public static class SplitSentence extends BaseBasicBolt {  
    public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {  
        String sentence = tuple.getStringByField("sentence");  
        String words[] = sentence.split(" ");  
        for (String w : words) {  
            basicOutputCollector.emit(new Values(w));  
        }  
    }  
  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("word"));  
    }  
}
```



Word Count - Word Count Bolt

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null) count = 0;
        count++;
        counts.put(word, count);
        LOG.info("Count of word: " + word + " = " + count);
        collector.emit(new Values(word, count));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```



Word Count - Topology

```
public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("spout", new RandomSentenceSpout(), 5);

    builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
    builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));

    Config conf = new Config();
    conf.setMaxTaskParallelism(3);
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("word-count", conf, builder.createTopology());

    Thread.sleep(10000);
    cluster.shutdown();
}
```

Run the Word Count Topology

```
# Start ZooKeeper
cd zookeeper
cp conf/zoo_sample.cfg conf/zoo.cfg
./bin/zkServer.sh start

# Start Storm Cluster on Local machine
cd apache-storm
./bin/storm nimbus
./bin/storm supervisor
./bin/storm ui # http://localhost:8080/index.html

# Run the example WordCount
cd storm-wordcount
mvn clean install
cd ../apache-storm
./bin/storm jar ../storm-wordcount/target/storm-example-1.0-jar-with-dependencies.jar
    wordcount.WordCountTopology WordCount

# To kill the topology
./bin/storm kill WordCount
```

MillWheel

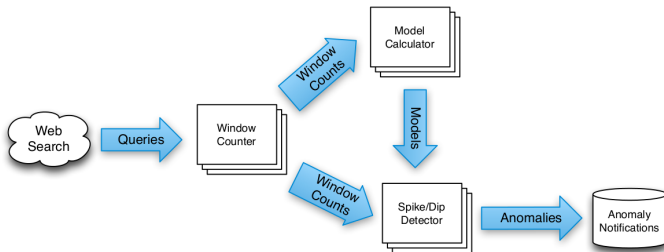


Motivation

- ▶ Google's **Zeitgeist** pipeline: **tracking trends** in web queries
- ▶ Builds a **historical model** of each query.
- ▶ Performs **anomaly detection**.

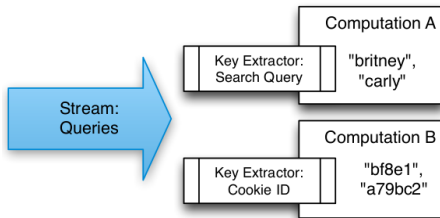
MillWheel Dataflow

- ▶ A graph of user-defined transformations (**computations**).
- ▶ **Stream**: unbounded data of (key, value, timestamp) records.
- ▶ **Timestamp**: wall clock time when the event occurred (**event-time**).



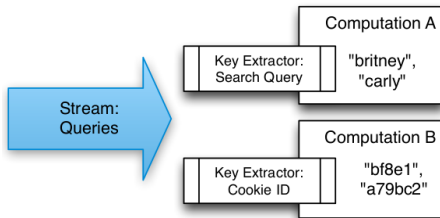
Key Extraction Function

- **Key extraction function**: specified by the stream consumer to **assign keys** to records.



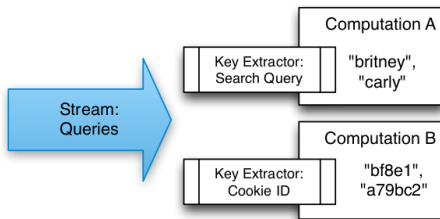
Key Extraction Function

- ▶ **Key extraction function**: specified by the stream consumer to **assign keys** to records.
- ▶ **Computation** can only access state for the **specific key**.
- ▶ **Multiple** computations can extract **different keys** from the **same stream**.



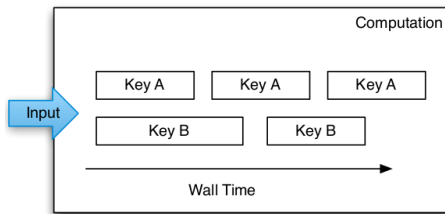
Key Extraction Function

- ▶ **Key extraction function**: specified by the stream consumer to **assign keys** to records.
- ▶ **Computation** can only access state for the **specific key**.
- ▶ **Multiple** computations can extract **different keys** from the **same stream**.
- ▶ A **computation** subscribes to **zero or more input** streams and publishes **one or more output** streams.



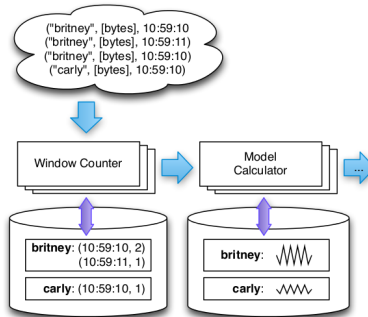
Computation

- ▶ Application logic lives in **computations**.
- ▶ Users can **add and remove** computations from a topology **dynamically**.
- ▶ Runs in the context of a **single key**.
- ▶ **Parallel** **per-key** processing.



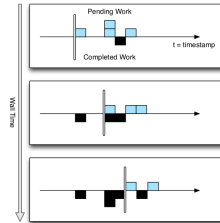
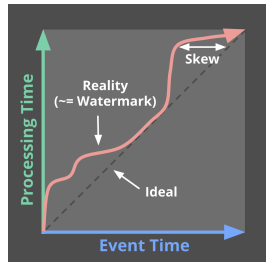
Persistent State

- ▶ Managed on **per-key** basis
- ▶ Stored in **Bigtable** or **Spanner**
- ▶ Common use: **aggregation**, **joins**, ...



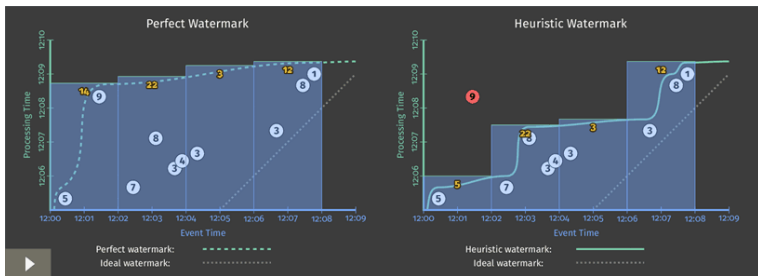
Low Watermarks (1/3)

- ▶ **Low watermark:** captures the **progress** of **event time completeness** as **processing time** progresses.
- ▶ Think of **low watermark** as a function $F(P) \rightarrow E$.
 - **P** and **E** are points in **processing time** and **event time**.
 - **E** is the point up to which the system believes **all inputs** with **event times** less than **E** have been observed.



Low Watermarks (2/3)

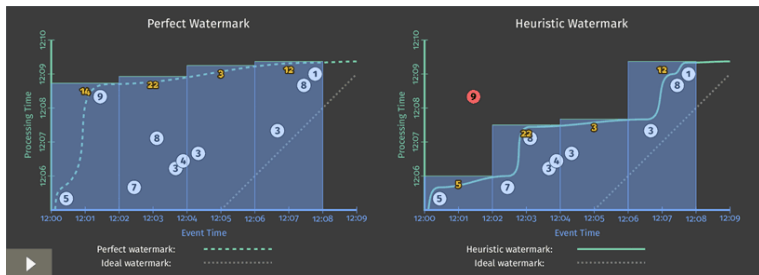
- ▶ Low watermark values are seeded by injectors that send data into MillWheel.
- ▶ Perfect vs. heuristic watermarks.



[<https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102>]

Low Watermarks (2/3)

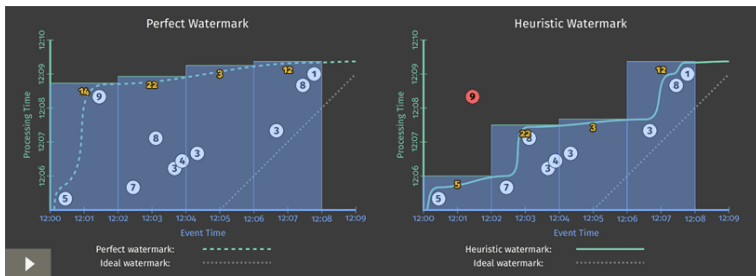
- ▶ Low watermark values are seeded by injectors that send data into MillWheel.
- ▶ Perfect vs. heuristic watermarks.
- ▶ Slow vs. fast watermarks.



[<https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102>]

Low Watermarks (3/3)

- ▶ **Late** records: records **sbehind** the low watermark.
- ▶ Process them according to the **application**, e.g., **discard** or **correct the result**.



<https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102>



Delivery Guarantees - Strong Productions

- ▶ Emitted records are **checkpointed** before **delivery**.
 - If an ACK is **not received**, the record can be **re-sent**.
 - **Exactly-one** delivery: **duplicates are discarded** by MillWheel at the recipient.



Delivery Guarantees - Strong Productions

- ▶ Emitted records are **checkpointed** before **delivery**.
 - If an ACK is **not received**, the record can be **re-sent**.
 - **Exactly-one** delivery: **duplicates are discarded** by MillWheel at the recipient.
- ▶ The **checkpoints** allow **fault-tolerance**.
 - If a processor crashes and is restarted somewhere else any intermediate computations can be recovered.

Delivery Guarantees - Strong Productions

- ▶ Emitted records are **checkpointed** before **delivery**.
 - If an ACK is **not received**, the record can be **re-sent**.
 - **Exactly-one** delivery: **duplicates are discarded** by MillWheel at the recipient.
- ▶ The **checkpoints** allow **fault-tolerance**.
 - If a processor crashes and is restarted somewhere else any intermediate computations can be recovered.
- ▶ When a delivery is ACKed the checkpoints can be **garbage collected**.

Delivery Guarantees - Strong Productions

- ▶ Emitted records are **checkpointed** before **delivery**.
 - If an ACK is **not received**, the record can be **re-sent**.
 - **Exactly-one** delivery: **duplicates are discarded** by MillWheel at the recipient.
- ▶ The **checkpoints** allow **fault-tolerance**.
 - If a processor crashes and is restarted somewhere else any intermediate computations can be recovered.
- ▶ When a delivery is ACKed the checkpoints can be **garbage collected**.
- ▶ The **Checkpoint**→**Delivery**→**ACK**→**GC** sequence is called a **strong production**.

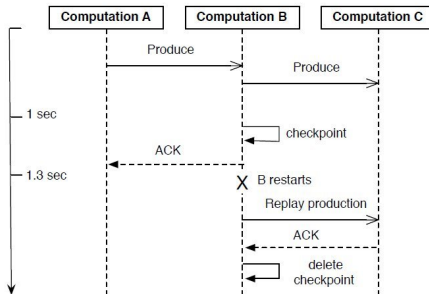


Delivery Guarantees - Weak Productions (1/2)

- ▶ Disable the exactly-once and/or strong production guarantee for applications that do not need it.
- ▶ Weak production is when Millwheel users can allow events to be sent before the checkpoint is committed to persistent storage.

Delivery Guarantees - Weak Productions (2/2)

- Weak production checkpointing prevents straggler productions from occupying undue resources in the sender (Computation A) by saving a checkpoint for receiver (Computation B).



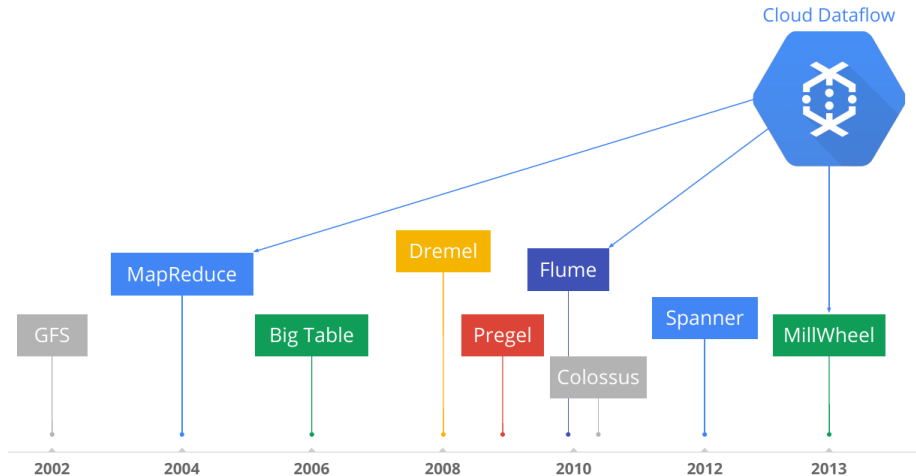
Google Cloud Dataflow



Google Cloud Dataflow (1/4)

- ▶ Google managed service for batch and stream data processing.
- ▶ Design issues
 - Continuous vs. micro-batch processing
 - Record-at-a-Time vs. declarative APIs

Google Cloud Dataflow (2/4)





Google Cloud Dataflow (3/4)

- ▶ **FlumeJava**: dataflow programming model
- ▶ **MapReduce**: batch processing
- ▶ **MillWheel**: stream processing



Google Cloud Dataflow (4/4)

- ▶ Open source **Cloud Dataflow SDK**
- ▶ Express your data processing **pipeline** using **FlumeJava**.



Google Cloud Dataflow (4/4)

- ▶ Open source **Cloud Dataflow SDK**
- ▶ Express your data processing **pipeline** using **FlumeJava**.
- ▶ If you run it in **batch** mode, it executed on the **MapReduce** framework.



Google Cloud Dataflow (4/4)

- ▶ Open source **Cloud Dataflow SDK**
- ▶ Express your data processing **pipeline** using **FlumeJava**.
- ▶ If you run it in **batch** mode, it executed on the **MapReduce** framework.
- ▶ If you run it in **streaming** mode, it is executed on the **MillWheel** framework.



-
- The diagram shows a data flow process with two parallel paths. On the left, a blue cylinder (input) feeds into a green rectangle, which then feeds into a yellow rectangle. On the right, a blue cylinder (input) feeds into a red rectangle, which then feeds into a purple rectangle. Both the yellow and purple rectangles feed into a central orange rectangle. From the orange rectangle, the flow splits: one path goes to a black rectangle and then to a blue cylinder (output), while the other path goes directly to a blue cylinder (output).

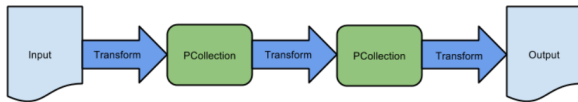


Dataflow Main Components

- ▶ Pipelines
- ▶ PCollections
- ▶ Transforms
- ▶ I/O sources and sinks

Pipelines (1/2)

- ▶ A **pipeline** represents a **data processing job**.
- ▶ **Directed graph** of operating on data.
- ▶ A pipeline consists of **two** parts:
 - **Data** (**PCollection**)
 - **Transforms** applied to that data

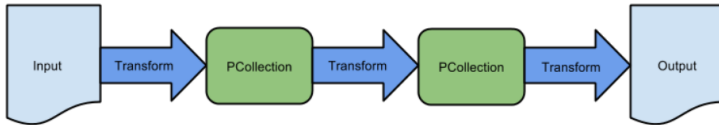


Pipelines (2/2)

```
public static void main(String[] args) {  
  
    // Create a pipeline  
    PipelineOptions options = PipelineOptionsFactory.create();  
    Pipeline p = Pipeline.create(options);  
  
    p.apply(TextIO.Read.from("gs://..."))    // Read input.  
      .apply(new CountWords())              // Do some processing.  
      .apply(TextIO.Write.to("gs://..."));  // Write output.  
  
    // Run the pipeline.  
    p.run();  
}
```

PCollections (1/2)

- ▶ A **parallel collection** of records
- ▶ **Immutable**
- ▶ Must specify **bounded** or **unbounded**





PCollections (2/2)

```
// Create a Java Collection, in this case a List of Strings.
static final List<String> LINES = Arrays.asList("line 1", "line 2", "line 3");

PipelineOptions options = PipelineOptionsFactory.create();
Pipeline p = Pipeline.create(options);

// Create the PCollection
p.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of())
```

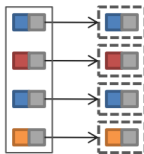


Transformations

- ▶ A **processing operation** that transforms data
- ▶ Each transform accepts **one (or multiple) PCollections** as input, performs an operation, and produces **one (or multiple) new PCollections** as output.
- ▶ Core transforms: **ParDo, GroupByKey, Combine, Flatten**

Transformations - ParDo

- Processes each element of a **PCollection** independently using a **user-provided DoFn**.



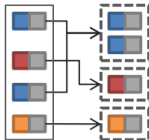
```
// The input PCollection of Strings.
PCollection<String> words = ...;

// The DoFn to perform on each element in the input PCollection.
static class ComputeWordLengthFn extends DoFn<String, Integer> { ... }

// Apply a ParDo to the PCollection "words" to compute lengths for each word.
PCollection<Integer> wordLengths = words.apply(ParDo.of(new ComputeWordLengthFn()));
```

Transformations - GroupByKey

- Takes a **PCollection** of key-value pairs and **gathers up all values with the same key**.



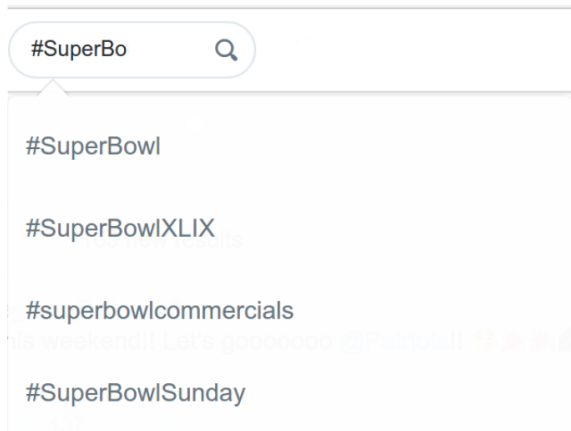
```
// A PCollection of key/value pairs: words and line numbers.  
PCollection<KV<String, Integer>> wordsAndLines = ...;  
  
// Apply a GroupByKey transform to the PCollection "wordsAndLines".  
PCollection<KV<String, Iterable<Integer>>> groupedWords = wordsAndLines.apply(  
    GroupByKey.<String, Integer>create());
```


Transformations - Join and CoGroupByKey

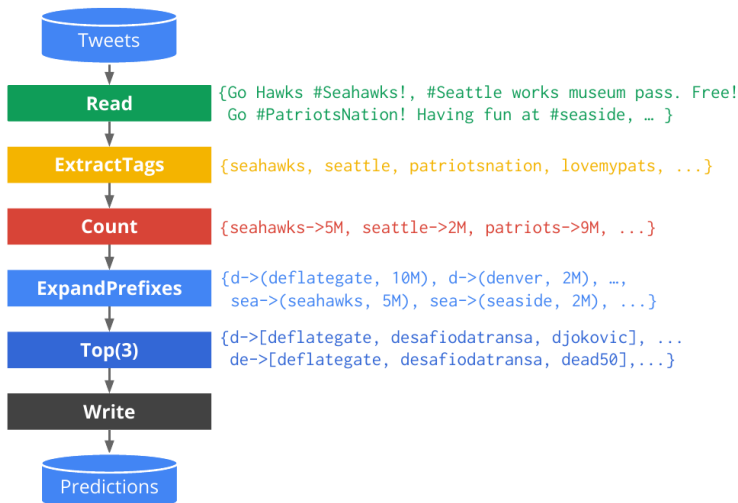
- Groups together the **values** from **multiple PCollections** of key-value pairs.

```
// Each data set is represented by key-value pairs in separate PCollections.  
// Both data sets share a common key type ("K").  
PCollection<KV<K, V1>> pc1 = ...;  
PCollection<KV<K, V2>> pc2 = ...;  
  
// Create tuple tags for the value types in each collection.  
final TupleTag<V1> tag1 = new TupleTag<V1>();  
final TupleTag<V2> tag2 = new TupleTag<V2>();  
  
// Merge collection values into a CoGbkResult collection.  
PCollection<KV<K, CoGbkResult>> coGbkResultCollection =  
    KeyedPCollectionTuple.of(tag1, pc1)  
        .and(tag2, pc2)  
        .apply(CoGroupByKey.<K>create());
```

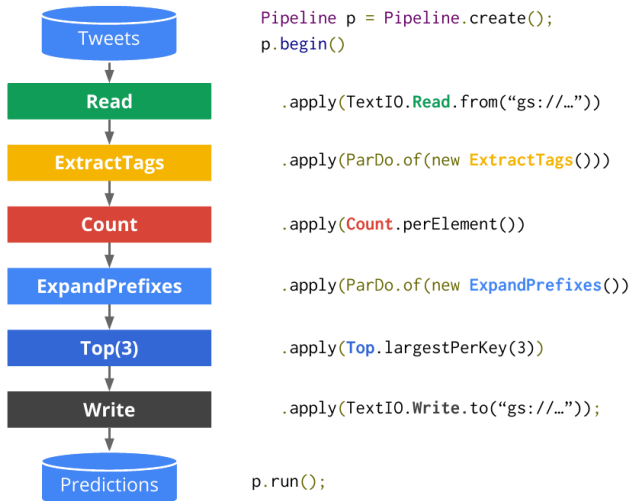
Example: HashTag Autocompletion (1/3)



Example: HashTag Autocompletion (2/3)



Example: HashTag Autocompletion (3/3)



Windowing and Triggering



Windowing and Triggering

- ▶ **Windowing** determines **where** in **event time**, data are grouped together for processing.
- ▶ **Triggering** determines **when** in **processing time** the results of groupings are emitted as panes.

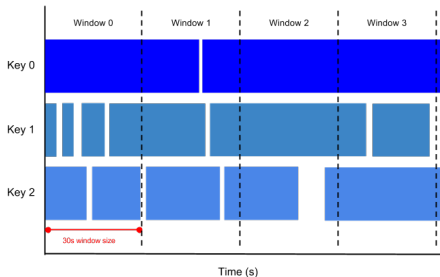


Windowing (1/4)

- ▶ Logically divide up or groups the elements of a `PCollection` into finite windows.
- ▶ Each element in a `PCollection` is assigned to one or more windows.
- ▶ Windowing functions:
 - Fixed time windows
 - Sliding time windows
 - Session windows

Windowing (2/4)

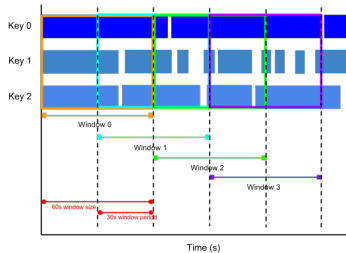
- ▶ Fixed time windows
- ▶ Represents the **time interval** in the data stream to define bundles of data, e.g., hourly



```
PCollection<String> items = ...;  
PCollection<String> fixed_windowed_items = items.apply(  
    Window.<String>into(FixedWindows.of(1, TimeUnit.MINUTES)));
```


Windowing (3/4)

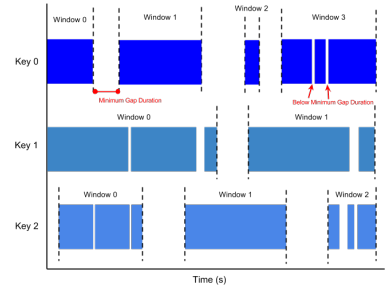
- ▶ Sliding time windows
- ▶ Uses **time intervals** in the data stream to define bundles of data, however the windows overlap.



```
PCollection<String> items = ...;
PCollection<String> sliding_windowed_items = items.apply(
    Window.<String>into(SlidingWindows
        .of(Duration.standardMinutes(30))
        .every(Duration.standardSeconds(5))));
```

Windowing (4/4)

- ▶ Session windows
- ▶ Defines windows around areas of concentration in the data.
- ▶ Useful for data that is irregularly distributed with respect to time, e.g., user mouse activity
- ▶ Applies on a **per-key** basis



```
PCollection<String> items = ...;
PCollection<String> session_windowed_items = items.apply(
    Window.<String>into(Sessions.withGapDuration(Duration.standardMinutes(10))));
```



Triggers

- ▶ Determine **when to emit** elements into an **aggregated window**.
- ▶ Provide **flexibility** for dealing with time skew and data lag.
 - Example: deal with **late-arriving data**.
 - Example: get **early results**, before all the data in a given window has arrived.

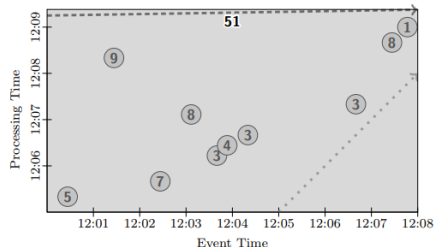
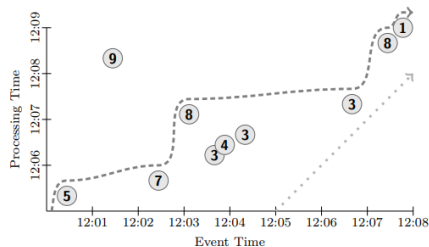


Triggers

- ▶ Determine **when to emit** elements into an **aggregated window**.
- ▶ Provide **flexibility** for dealing with time skew and data lag.
 - Example: deal with **late-arriving data**.
 - Example: get **early results**, before all the data in a given window has arrived.
- ▶ **Three** main types of triggers:
 - **Time-based** triggers
 - **Data-driven** triggers
 - **Composit** triggers

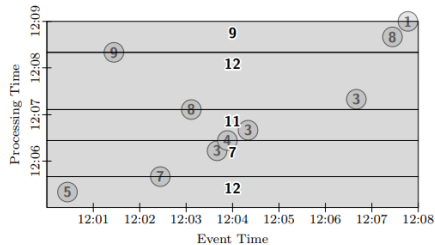
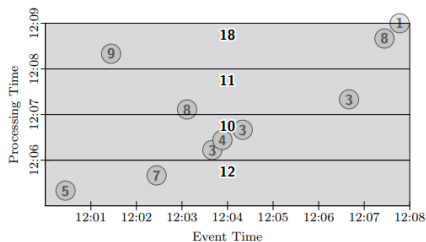
Example (1/3)

► Batch processing



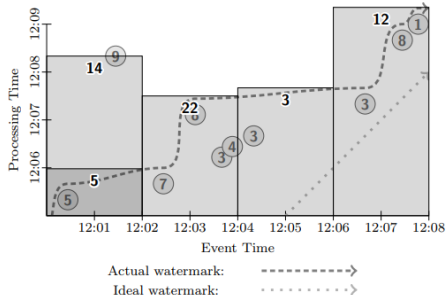
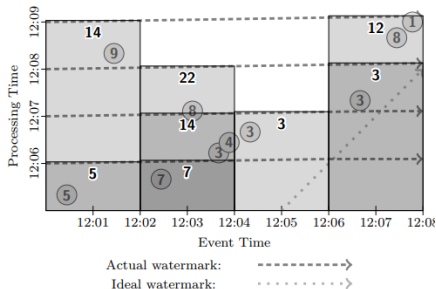
Example (2/3)

- ▶ Trigger at **period**
- ▶ Trigger at **count**



Example (3/3)

- ▶ Fixed window, trigger at **period** (micro-batch)
- ▶ Fixed window, trigger at **watermark** (streaming)



Summary



Summary

► Storm

- Tuple and stream
- Spout, bolt, and topology
- Nimbus, worker, supervisor, and zookeeper
- Reliable processing: xored ids



Summary

► MillWheel

- DAG of computations
- Persistent state: per-key
- Low watermark
- Exactly-one delivery



Summary

- ▶ Google cloud dataflow
 - Pipeline
 - PCollection: windows and triggers
 - Transforms

- ▶ A. Toshniwal et al., “Storm @ twitter”, ACM SIGMOD 2014.
- ▶ T. Akidau et al., “MillWheel: fault-tolerant stream processing at internet scale”, VLDB 2013.
- ▶ T. Akidau et al., “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing”, VLDB 2015.
- ▶ The world beyond batch: Streaming 102
<https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102>

Questions?

Some slides and pictures were derived from Derek G. Murray (Google) slides.