My data is a day old...

Meh.
Case Study: Steve Jobs passes away
Initial Implementation

Algorithm: Co-occurrences within query sessions
Implementation: Pig scripts over query logs on HDFS

Problem: Query suggestions were several hours old!

Why?
- Log collection lag
- Hadoop scheduling lag
- Hadoop job latencies

We need real-time processing!
Solution?

Can we do better than one-off custom systems?

[Diagram of system architecture with HDFS, frontend cache, and backend engine components such as ranking algorithm, in-memory stores, and stats collector.]
Stream Processing Frameworks

Source: Wikipedia (River)
Background Review -- Stream Processing

<table>
<thead>
<tr>
<th>Batch processing</th>
<th>Stream processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>All the data</td>
<td>Continuously incoming data</td>
</tr>
<tr>
<td>Not real time</td>
<td>Latency critical (near real time)</td>
</tr>
</tbody>
</table>
Use Cases Across Industries

Credit
Identify fraudulent transactions as soon as they occur.

Transportation
Dynamic Re-routing Of traffic or Vehicle Fleet.

Retail
- Dynamic Inventory Management
- Real-time In-store Offers and recommendations

Consumer Internet & Mobile
Optimize user engagement based on user’s current behavior.

Healthcare
Continuously monitor patient vital stats and proactively identify at-risk patients.

Manufacturing
- Identify equipment failures and react instantly
- Perform Proactive maintenance.

Surveillance
Identify threats and intrusions In real-time

Digital Advertising & Marketing
Optimize and personalize content based on real-time information.
Canonical Stream Processing Architecture
What is a data stream?

Sequence of items:
- Structured (e.g., tuples)
- Ordered (implicitly or timestamped)
- Arriving continuously at high volumes
- Sometimes not possible to store entirely
- Sometimes not possible to even examine all items
What exactly do you do?

“Standard” relational operations:
  - Select
  - Project
  - Transform (i.e., apply custom UDF)
  - Group by
  - Join
  - Aggregations

What else do you need to make this “work”? 
Issues of Semantics

Group by... aggregate
When do you stop grouping and start aggregating?

Joining a stream and a static source
Simple lookup

Joining two streams
How long do you wait for the join key in the other stream?

Joining two streams, group by and aggregation
When do you stop joining?

What’s the solution?
Windows

Windows restrict processing scope:

- Windows based on ordering attributes (e.g., time)
- Windows based on item (record) counts
- Windows based on explicit markers (e.g., punctuations)
Windows on Ordering Attributes

Assumes the existence of an attribute that defines the order of stream elements (e.g., time)

Let $T$ be the window size in units of the ordering attribute

- Sliding window: $t_{i}' - t_i = T$
- Tumbling window: $t_{i+1} - t_i = T$
Windows on Counts

Window of size N elements (sliding, tumbling) over the stream
Windows from “Punctuations”

Application-inserted “end-of-processing”
Example: stream of actions... “end of user session”

Properties

Advantage: application-controlled semantics
Disadvantage: unpredictable window size (too large or too small)
Streams Processing Challenges

Inherent challenges
- Latency requirements
- Space bounds

System challenges
- Bursty behavior and load balancing
- Out-of-order message delivery and non-determinism
- Consistency semantics (at most once, exactly once, at least once)
How do consumers get data from producers?
Producer/Consumers

Producer

Consumer

Producer pushes e.g., callback
Producer/Consumers

Producer

Consumer

Consumer pulls e.g., poll, tail
Producer/Consumers

Queue, Pub/Sub
Producer/Consumers

Producer

Broker

Consumer

Consumer

Consumer

Consumer
Topologies

Storm topologies = “job”
Once started, runs continuously until killed

A topology is a computation graph
Graph contains vertices and edges
Vertices hold processing logic
Directed edges indicate communication between vertices

Processing semantics
At most once: without acknowledgments
At least once: with acknowledgments
Spouts and Bolts: Logical Plan

Components

Tuples: data that flow through the topology
Spouts: responsible for emitting tuples
Bolts: responsible for processing tuples
Spouts and Bolts: Physical Plan

Physical plan specifies execution details
Parallelism: how many instances of bolts and spouts to run
Placement of bolts/spouts on machines

...
Stream Groupings

Bolts are executed by multiple instances in parallel
User-specified as part of the topology

When a bolt emits a tuple, where should it go?
Answer: Grouping strategy
Shuffle grouping: randomly to different instances
Field grouping: based on a field in the tuple
Global grouping: to only a single instance
All grouping: to every instance
Spark Streaming: Discretized Streams

Run a streaming computation as a series of very small, deterministic batch jobs

Chop up the stream into batches of $X$ seconds
Process as RDDs!
Return results in batches

live data stream

batches of $X$ seconds

processed results

Spark Streaming

Spark

Typical batch window $\sim 1s$
Example: Get hashtags from Twitter

```scala
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data

Twitter Streaming API

tweets DStream

stored in memory as an RDD (immutable, distributed)
Example: Get hashtags from Twitter

```scala
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```

**Transformation:** modify data in one Dstream to create another DStream

**tweets DStream**

**hashTags Dstream**

[#{cat, dog, ...}]

**New RDDs created for every batch**
Example: Get hashtags from Twitter

```scala
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

output operation: to push data to external storage

every batch saved to HDFS
Fault Tolerance

Bottom line: they’re just RDDs!
Fault Tolerance

Bottom line: they’re just RDDs!

tweets RDD

hashTags RDD

flatMap

input data replicated in memory

lost partitions recomputed on other workers
Key Concepts

DStream – sequence of RDDs representing a stream of data
   Twitter, HDFS, Kafka, Flume, TCP sockets

Transformations – modify data from on DStream to another
   Standard RDD operations – map, countByValue, reduce, join, ...
   Stateful operations – window, countByValueAndWindow, ...

Output Operations – send data to external entity
   saveAsHadoopFiles – saves to HDFS
   foreach – do anything with each batch of results
Example: Count the hashtags

```scala
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.countByValue()
```
Example: Count the hashtags over last 10 mins

```scala
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

- **sliding window operation**
- **window length**
- **sliding interval**
Example: Count the hashtags over last 10 mins

```scala
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

![Diagram showing a sliding window over time intervals t-1, t, t+1, t+2, t+3, with arrows indicating the countByValue operation over all data in the window.](image-url)
val `tagCounts` = `hashtags`.countByValueAndWindow(Minutes(10), Seconds(1))

```
<table>
<thead>
<tr>
<th>t-1</th>
<th>t</th>
<th>t+1</th>
<th>t+2</th>
<th>t+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashTags</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>countByValue</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tagCounts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- **subtracts** the counts from batch before the window
- **adds** the counts from the new batch in the window
Smart window-based reduce

Incremental counting generalizes to many reduce operations

Need a function to “inverse reduce” (“subtract” for counting)

val tagCounts = hashtags.
   .countByValueAndWindow(Minutes(10), Seconds(1))

val tagCounts = hashtags.
   .reduceByKeyAndWindow(_ + _, _ - _, Minutes(10), Seconds(1))
Can process **6 GB/sec (60M records/sec)** of data on 100 nodes at **sub-second** latency Tested

- with 100 streams of data on 100 EC2 instances with 4 cores each
Comparison with Storm

Higher throughput than Storm

- Spark Streaming: 670k records/second/node
- Storm: 115k records/second/node

![Grep Comparison Chart]

![WordCount Comparison Chart]