Near Real-Time Stream Processing Architectures

Anand Iyer
Senior Product Manager, Cloudera
aiyer@cloudera.com
Its 2015: Your CTO wants Real-Time

Why now? Complex Event Processing (CEP) is not a new concept.
Use Cases Across Industries

**Credit Card & Monetary Transactions**
Identify fraudulent transactions as soon as they occur.

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

**Retail**
- Real-time in-store Offers and Recommendations.
- Email and marketing campaigns based on real-time social trends

**Digital Advertising & Marketing**
Optimize and personalize digital ads based on real-time information.

**Consumer Internet, Mobile & E-Commerce**
Optimize user engagement based on user’s current behavior. Deliver recommendations relevant “in the moment”

**Manufacturing**
- Identify equipment failures and react instantly
- Perform proactive maintenance.
- Identify product quality defects immediately to prevent resource wastage.

**Security & Surveillance**
Identify threats and intrusions, both digital and physical, in real-time.

**Transportation & Logistics**
- Real-time traffic conditions
- Tracking fleet and cargo locations and dynamic re-routing to meet SLAs
Operations on Sliding Windows

Easily define operations over a sliding window of data

Specify:
- Window length as multiple of micro-batch size
- Sliding step size

NOTE: Provide adequate memory to hold sliding window worth of data.
Maintain and update arbitrary state

`updateStateByKey(...)`
- Define initial state
- Provide state update function
- Continuously update with new information

Examples:
- Running count of words seen in text stream
- Per user session state from activity stream

Note:
Requires periodic check-pointing to fault-tolerant storage.
# OSS options for Stream Processing

<table>
<thead>
<tr>
<th></th>
<th>Spark Streaming</th>
<th>Storm</th>
<th>Trident (built on Storm)</th>
<th>Samza</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
<td>micro-batch</td>
<td>one-at-a-time</td>
<td>micro-batch</td>
<td>one-at-a-time</td>
</tr>
<tr>
<td><strong>Language Support</strong></td>
<td>Scala, Java, Python</td>
<td>Java, Scala, Python, Ruby, Clojure...</td>
<td>Java, Clojure, Scala</td>
<td>Scala, Java</td>
</tr>
<tr>
<td><strong>Resource Managers</strong></td>
<td>YARN, Mesos, Standalone</td>
<td>YARN, Mesos</td>
<td>YARN, Mesos</td>
<td>YARN</td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td>~0.5 seconds</td>
<td>~100ms</td>
<td>~0.5 seconds</td>
<td>~100ms</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>****</td>
<td>**</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td><strong>Age</strong></td>
<td>2+</td>
<td>3.5+</td>
<td>1.5+</td>
<td>1+</td>
</tr>
<tr>
<td><strong>Known Production Instances</strong></td>
<td>50+ Multi-Vendor Support</td>
<td>50+ Multi-Vendor Support</td>
<td>??? Multi-Vendor Support</td>
<td>Outside LinkedIn only a handful. No vendors.</td>
</tr>
</tbody>
</table>
# OSS options for Stream Processing

<table>
<thead>
<tr>
<th></th>
<th>Spark Streaming</th>
<th>Storm</th>
<th>Trident (built on Storm)</th>
<th>Samza</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exactly Once Processing</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Functions on Sliding Windows</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Higher Order Functions</strong></td>
<td>Yes. From Spark.</td>
<td>No.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>(Aggregations, Joins, etc)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Spark Streaming Advantage

- Automatically inherit developments in Spark
  - DataFrames
  - Mllib
  - Dynamic Resource Allocation
  - Vast ecosystem of “packages”

- Same framework for batch and streaming
  - Operational ease
  - Lambda Architectures are easy to implement
Exactly Once Processing

Should you care?

- In a cluster, machine failure is frequent
- “Double Counting” leads to False Positives: Alerting, predictive analytics, etc will have too many false positives when you double count data. You will end up “loosening” your thresholds

Thus, not a trivial consideration.
Exactly Once in Spark Streaming

**Receiving Data:**
- Use Kafka Direct receiver
- If offsets are fixed, can re-Create micro-batch RDD identically
- What if producer put dupes in Kafka? Generate UUID per event and dedupe.

**Process Data:**
- Deterministic DAG of operations

**Output Processed Data:**
- Failures can happen when only part of the output data is written
- Each micro-batch has a unique identifier: Batch-Time
- Use batch-time as key to perform “transactional writes”
Thank You!