Distributed Real-Time Stream Processing: Why and How

Petr Zapletal
@petr_zapletal
Agenda

- Motivation
- Stream Processing
- Available Frameworks
- Systems Comparison
- Recommendations
The Data Deluge

- 8 Zettabytes (1 ZB = $10^{21}$ B = 1 billion TB) created in 2015
- Every minute we create
  - 200 million emails
  - 48 hours of YouTube video
  - 2 million google queries
  - 200,000 tweets
  - ...
- How can we make sense of all data
  - Most data is not interesting
  - New data supersedes old data
  - Challenge is not only storage but processing
New Sources And New Use Cases

- Many new sources of data become available
  - Sensors
  - Mobile devices
  - Web feeds
  - Social networking
  - Cameras
  - Databases
  - ...

- Even more use cases become viable
  - Web/Social feed mining
  - Real-time data analysis
  - Fraud detection
  - Smart order routing
  - Intelligence and surveillance
  - Pricing and analytics
  - Trends detection
  - Log processing
  - Real-time data aggregation
  - ...

Stream Processing to the Rescue

- Process data streams on-the-fly without permanent storage
- Stream data rates can be high
  - High resource requirements for processing (clusters, data centres)
- Processing stream data has real-time aspect
  - Latency of data processing matters
  - Must be able to react to events as they occur
Streaming Applications

ETL Operations

- Transformations, joining or filtering of incoming data

Windowing

- Trends in bounded interval, like tweets or sales
Streaming Applications

Machine Learning
- Clustering, Trend fitting, Classification

Pattern Recognition
- Fraud detection, Signal triggering, Anomaly detection
Processing Architecture Evolution

Batch Pipeline

- HDFS
- Serving DB
- Query

Lambda Architecture

- Batch Layer
- Serving Layer
- Stream layer
- Oozie
- Query

Standalone Stream Processing

- Stream Processing
- Query

Kappa Architecture

- Stream Processing
- Query
Distributed Stream Processing

- Continuous processing, aggregation and analysis of unbounded data
- General computational model as MapReduce
- Expected latency in milliseconds or seconds
- Systems often modelled as Directed Acyclic Graph (DAG)

- Describes topology of streaming job
- Data flows through chain of processors from sources to sinks
Points of Interest

- Runtime and programming model
- Primitives
- State management
- Message delivery guarantees
- Fault tolerance & Low overhead recovery
- Latency, Throughput & Scalability
- Maturity and Adoption Level
- Ease of development and operability
Runtime and Programming Model

- Most important trait of stream processing system
- Defines expressiveness, possible operations and its limitations
- Therefore defines systems capabilities and its use cases
Native Streaming

Native stream processing systems
continuous operator model

records processed one at a time
Micro-batching

Records processed in short batches
Native Streaming

- Records are processed as they arrive
- Native model with general processing ability

Pros
- Expressiveness
- Low-latency
- Stateful operations

Cons
- Throughput
- Fault-tolerance is expensive
- Load-balancing
Micro-batching

- Splits incoming stream into small batches
- Batch interval inevitably limits system expressiveness
- Can be built atop Native streaming easily

**Pros**

- High-throughput
- Easier fault tolerance
- Simpler load-balancing

**Cons**

- Lower latency, depends on batch interval
- Limited expressivity
- Harder stateful operations
## Programming Model

<table>
<thead>
<tr>
<th>Compositional</th>
<th>Declarative</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Provides basic building blocks as operators or sources</td>
<td>➔ High-level API</td>
</tr>
<tr>
<td>➔ Custom component definition</td>
<td>➔ Operators as higher order functions</td>
</tr>
<tr>
<td>➔ Manual Topology definition &amp; optimization</td>
<td>➔ Abstract data types</td>
</tr>
<tr>
<td>➔ Advanced functionality often missing</td>
<td>➔ Advance operations like state management or windowing supported out of the box</td>
</tr>
<tr>
<td></td>
<td>➔ Advanced optimizers</td>
</tr>
</tbody>
</table>
Apache Streaming Landscape

- Storm
- Trident
- Spark Streaming
- Samza
- Flink
Storm

- Originally created by Nathan Marz and his team at BackType in 2010
- Being acquired and later open-sourced by Twitter, Apache project top-level since 2014
- Pioneer in large scale stream processing
- Low-level native streaming API
- Uses Thrift for topology definition
- Large number of API languages available
  - Storm Multi-Language Protocol
Trident

- Higher level micro-batching system build atop Storm
- Stream is partitioned into a small batches
- Simplifies building topologies
- Java, Clojure and Scala API
- Provides exactly once delivery
- Adds higher level operations
  - Aggregations
  - State operations
  - Joining, merging, grouping, windowing, etc.
Spark started in 2009 at UC Berkeley, Apache since since 2013
● General engine for large scale batch processing
● Spark Streaming introduced in 0.7, came out of alpha in 0.9 (Feb 2014)
● Unified batch and stream processing over a batch runtime
● Great integration with batch processing and its build-in libraries (SparkSQL, MLlib, GraphX)
● Scala, Java & Python API
Samza

- Developed in LinkedIn, open-sourced in 2013
- Builds heavily on Kafka’s log based philosophy
- Pluggable messaging system and executional backend
  - Uses Kafka & YARN usually
- JVM languages, usually Java or Scala
Flink

- Started as Stratosphere in 2008 at as Academic project
- Native streaming
- High level API
- Batch as special case of Streaming (bounded vs unbounded dataset)
- Provides ML (FlinkML) and graph processing (Gelly) out of the box
- Java, Scala & Python API
# System Comparison

<table>
<thead>
<tr>
<th>Streaming Model</th>
<th>Native</th>
<th>Micro-batching</th>
<th>Micro-batching</th>
<th>Native</th>
<th>Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Compositional</td>
<td>Declarative</td>
<td>Compositional</td>
<td>Declarative</td>
<td></td>
</tr>
<tr>
<td>Guarantees</td>
<td>At-least-once</td>
<td>Exactly-once</td>
<td>Exactly-once</td>
<td>At-least-once</td>
<td>Exactly-once</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Record ACKs</td>
<td>RDD based Checkpointing</td>
<td>Log-based</td>
<td>Checkpointing</td>
<td></td>
</tr>
<tr>
<td>State Management</td>
<td>Not build-in</td>
<td>Dedicated Operators</td>
<td>Dedicated DStream</td>
<td>Stateful Operators</td>
<td>Stateful Operators</td>
</tr>
<tr>
<td>Latency</td>
<td>Very Low</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Throughput</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Maturity</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td></td>
</tr>
</tbody>
</table>
Counting Words

NE Scala 2016 Apache
Apache Spark Storm
Apache Trident Flink
Streaming Samza Scala
2016 Streaming

(Apache, 3)
(Streaming, 2)
(Scala, 2)
(2016, 2)
(Spark, 1)
(Storm, 1)
(Trident, 1)
(Flink, 1)
(Samza, 1)
(NE, 1)
TopoogyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new Split(), 8).shuffleGrouping("spout");

Map<String, Integer> counts = new HashMap<String, Integer>();

public void execute(Tuple tuple, BasicOutputCollector collector) {
    String word = tuple.getString(0);
    Integer count = counts.containsKey(word) ? counts.get(word) + 1 : 1;
    counts.put(word, count);
    collector.emit(new Values(word, count));
}
Storm

```java
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new Split(), 8).shuffleGrouping("spout");

Map<String, Integer> counts = new HashMap<String, Integer>();

public void execute(Tuple tuple, BasicOutputCollector collector) {
    String word = tuple.getString(0);
    Integer count = counts.containsKey(word) ? counts.get(word) + 1 : 1;
    counts.put(word, count);
    collector.emit(new Values(word, count));
}
```
Storm

```java
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new Split(), 8).shuffleGrouping("spout");
...

Map<String, Integer> counts = new HashMap<String, Integer>();

public void execute(Tuple tuple, BasicOutputCollector collector) {
    String word = tuple.getString(0);
    Integer count = counts.containsKey(word) ? counts.get(word) + 1 : 1;
    counts.put(word, count);
    collector.emit(new Values(word, count));
}
```
public static StormTopology buildTopology(LocalDRPC drpc) {
    FixedBatchSpout spout = ...

    TridentTopology topology = new TridentTopology();
    TridentState wordCounts = topology.newStream("spout1", spout)
        .each(new Fields("sentence"), new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
        .persistentAggregate(new MemoryMapState.Factory(),
                                new Count(), new Fields("count"));

    ...
}

public static StormTopology buildTopology(LocalDRPC drpc) {
    FixedBatchSpout spout = ... 

    TridentTopology topology = new TridentTopology(); 
    TridentState wordCounts = topology.newStream("spout1", spout) 
        .each(new Fields("sentence"), new Split(), new Fields("word")) 
        .groupBy(new Fields("word")) 
        .persistentAggregate(new MemoryMapState.Factory(), 
        new Count(), new Fields("count")); 

    ... 
}

Trident
public static StormTopology buildTopology(LocalDRPC drpc) {
    FixedBatchSpout spout = ...

    TridentTopology topology = new TridentTopology();
    TridentState wordCounts = topology.newStream("spout1", spout)
        .each(new Fields("sentence"), new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
        .persistentAggregate(new MemoryMapState.Factory(),
            new Count(), new Fields("count"));

    ...
}
Spark Streaming

```scala
val conf = new SparkConf().setAppName("wordcount")
val ssc = new StreamingContext(conf, Seconds(1))

val text = ...
val counts = text.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)

counts.print()

ssc.start()
ssc.awaitTermination()
```
val conf = new SparkConf().setAppName("wordcount")
val ssc = new StreamingContext(conf, Seconds(1))

val text = ...  
val counts = text.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)

counts.print()

ssc.start()
ssc.awaitTermination()
Spark Streaming

```scala
val conf = new SparkConf().setAppName("wordcount")
val ssc = new StreamingContext(conf, Seconds(1))

val text = ...
val counts = text.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)

counts.print()

ssc.start()
ssc.awaitTermination()
```
val conf = new SparkConf().setAppName("wordcount")
val ssc = new StreamingContext(conf, Seconds(1))

val text = ...
val counts = text.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)

counts.print()

ssc.start()
ssc.awaitTermination()
class WordCountTask extends StreamTask {

    override def process(envelope: IncomingMessageEnvelope,
                         collector: MessageCollector, coordinator: TaskCoordinator) {

        val text = envelope.getMessage.asInstanceOf[String]

        val counts = text.split(" ")
                      .foldLeft(Map.empty[String, Int]) {
                        (count, word) => count + (word -> (count.getOrElse(word, 0) + 1))
                      }

        collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wordcount"),
                                                    counts))

    }
}
class WordCountTask extends StreamTask {

  override def process(envelope: IncomingMessageEnvelope,
                       collector: MessageCollector, coordinator: TaskCoordinator) {

    val text = envelope.getMessage.asInstanceOf[String]

    val counts = text.split(" ").foldLeft(Map.empty[String, Int]) {
      (count, word) => count + (word -> (count.getOrElse(word, 0) + 1))
    }

    collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wordcount"),
                                               counts))
  }
}
class WordCountTask extends StreamTask {

    override def process(envelope: IncomingMessageEnvelope,
                         collector: MessageCollector, coordinator: TaskCoordinator) {

        val text = envelope.getMessage.asInstanceOf[String]

        val counts = text.split(" ").foldLeft(Map.empty[String, Int]) {
            (count, word) => count + (word -> (count.getOrElse(word, 0) + 1))
        }

        collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wordcount"),
                                                     counts))
    }
}
val env = ExecutionEnvironment.getExecutionEnvironment

val text = env.fromElements(...) 
val counts = text.flatMap(_.split(" "))
  .map((_, 1))
  .groupBy(0)
  .sum(1)

counts.print()

env.execute("wordcount")
val env = ExecutionEnvironment.getExecutionEnvironment

val text = env.fromElements(...)
val counts = text.flatMap ( _.split(" ")
    .map ( (_, 1) )
    .groupBy(0)
    .sum(1)

counts.print()

env.execute("wordcount")
Fault Tolerance

- Fault tolerance in streaming systems is inherently harder than in batch
  - Can’t restart computation easily
  - State is a problem
  - Jobs can run 24/7
  - Fast recovery is critical

- A lot of challenges must be addressed
  - No single point of failure
  - Ensure processing of all incoming messages
  - State consistency
  - Fast recovery
  - Exactly once semantics is even harder
- Record acknowledgments
- Tracks the lineage of tuples in DAG as they are processed
- Special “ACK” bold track each lineage
- Able to replay from the root of failed tuples
- Performance difficulties

**Reliable Processing**

_Acks are delivered via a system-level bolt_
Spark Streaming

- Failed DStreams can be recomputed using their lineage
- Checkpointing to persistent data storage
  - Reduce lineage length
  - Recover metadata
- Computation redistributed from failed node
- Similar to restarting Spark’s failed batch job

faster recovery by using multiple nodes for recomputations
Samza

- Takes an advantage of durable, partitioned, offset based messaging system
- Task monitors its offset, which moves forward as messages are processed
- Offset is checkpointed and can be restored on failure
Flink

- Based on distributed snapshots
- Sends checkpoint barriers through the stream
- Snapshots can be stored in durable storage

[Diagram showing data stream with checkpoint barriers and parts of different checkpoints]
Managing State

- Most of the non-trivial streaming applications have a state
- Stateful operation looks like this:

  \[ f: (input, state) \rightarrow (output, state') \]

- Delivery guarantees plays crucial role
  - At least once
    - Ensure all operators see all events ~ replay stream in failure case
  - Exactly once
    - Ensure that operators do not perform duplicate updates to their state
Storm & Trident

- States available only in Trident API
- Dedicated operators for state updates and queries
- State access methods
- Difficult to implement transactional state
- Exactly once semantics*

<table>
<thead>
<tr>
<th>Spout</th>
<th>Non-transactional</th>
<th>Transactional</th>
<th>Opaque transactional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-transactional</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Transactional</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Opaque transactional</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Spark Streaming

- State is implemented as another stream
  - UpdateStateByKey()
  - TrackStateByKey()
  - Requires checkpointing
- Stateless runtime by design
- Exactly-once semantics
Samza

- Stateful operators, any task can hold state
- State is stored as another log
- Ships with 2 key-value stores
  - In-memory & RocksDB
  - Possible to send updates to kafka changelog to restore store if needed
- Great for large data sets because of localized storage
- Can use custom storage engines
- At least once semantics, exactly once planned
- Stateful dataflow operators (conceptually similar to Samza)
- State access patterns
  - Local (Task) state - current state of a specific operator instance, operators do not interact
  - Partitioned (Key) state - maintains state of partitions (~ keys)
- Direct API integration
  - `mapWithState()`, `flatMapWithState()`, ...
- Checkpointing
  - Pluggable backends for storing snapshots
- Exactly-once semantics
Counting Words Revisited

NE Scala 2016 Apache
Apache Spark Storm
Apache Trident Flink
Streaming Samza Scala
2016 Streaming

(Apache,3)
(Streaming, 2)
(Scala, 2)
(2016, 2)
(Spark, 1)
(Storm, 1)
(Trident, 1)
(Flink, 1)
(Samza, 1)
(NE, 1)
import storm.trident.operation.builtin.Count;

TridentTopology topology = new TridentTopology();
TridentState wordCounts =
    topology.newStream("spout1", spout)
    .each(new Fields("sentence"), new Split(), new Fields("word"))
    .groupBy(new Fields("word"))
    .persistentAggregate(new MemoryMapState.Factory(), new Count(),
                        new Fields("count"))
    .parallelismHint(6);
import storm.trident.operation.builtin.Count;

TridentTopology topology = new TridentTopology();
TridentState wordCounts =
    topology.newStream("spout1", spout)
    .each(new Fields("sentence"), new Split(), new Fields("word"))
    .groupBy(new Fields("word"))
    .persistentAggregate(new MemoryMapState.Factory(), new Count(),
    new Fields("count"))
    .parallelismHint(6);
Spark Streaming

// Initial RDD input to updateStateByKey
val initialRDD = ssc.sparkContext.parallelize(List.empty[(String, Int)])

val lines = ...
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))

val trackStateFunc = (batchTime: Time, word: String, one: Option[Int], state: State[Int]) => {
  val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
  val output = (word, sum)
  state.update(sum)
  Some(output)
}

val stateDstream = wordDstream.trackStateByKey(
  StateSpec.function(trackStateFunc).initialState(initialRDD))
Spark Streaming

// Initial RDD input to updateStateByKey
val initialRDD = ssc.sparkContext.parallelize(List.empty[(String, Int)])

val lines = ...
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))

val trackStateFunc = (batchTime: Time, word: String, one: Option[Int], state: State[Int]) => {
  val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
  val output = (word, sum)
  state.update(sum)
  Some(output)
}

val stateDstream = wordDstream.trackStateByKey(
  StateSpec.function(trackStateFunc).initialState(initialRDD))
Spark Streaming

// Initial RDD input to updateStateByKey
val initialRDD = ssc.sparkContext.parallelize(List.empty[(String, Int)])

val lines = ...
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))

val trackStateFunc = (batchTime: Time, word: String, one: Option[Int], state: State[Int]) => {
  val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
  val output = (word, sum)
  state.update(sum)
  Some(output)
}

val stateDstream = wordDstream.trackStateByKey(
  StateSpec.function(trackStateFunc).initialState(initialRDD))
Spark Streaming

// Initial RDD input to `updateStateByKey`
val initialRDD = ssc.sparkContext.parallelize(List.empty[(String, Int)])

val lines = ...
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))

val trackStateFunc = (batchTime: Time, word: String, one: Option[Int], state: State[Int]) => {
val sum = one.getOrElse(0) + state.getOption().getOrElse(0)
val output = (word, sum)
state.update(sum)
Some(output)
}

val stateDstream = wordDstream.trackStateByKey(
    StateSpec.function(trackStateFunc).initialState(initialRDD))
Spark Streaming

// Initial RDD input to updateStateByKey
val initialRDD = ssc.sparkContext.parallelize(List.empty[(String, Int)])

val lines = ...
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))

val trackStateFunc = (batchTime: Time, word: String, one: Option[Int], state: State[Int]) => {
  val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
  val output = (word, sum)
  state.update(sum)
  Some(output)
}

val stateDstream = wordDstream.trackStateByKey(
  StateSpec.function(trackStateFunc).initialState(initialRDD))
class WordCountTask extends StreamTask with InitableTask {

  private var store: CountStore = _

  def init(config: Config, context: TaskContext) {
    this.store = context.getStore("wordcount-store").asInstanceOf[KeyValueStore[String, Integer]]
  }

  override def process(envelope: IncomingMessageEnvelope, collector: MessageCollector, coordinator: TaskCoordinator) {

    val words = envelope.getMessage.asInstanceOf[String].split(" ")

    words.foreach{ key =>
      val count: Integer = Option(store.get(key)).getOrElse(0)
      store.put(key, count + 1)
      collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wordcount"), (key, count)))
    }
  }
}
class WordCountTask extends StreamTask with InitableTask {

    private var store: CountStore = _

    def init(config: Config, context: TaskContext) {
        this.store = context.getStore("wordcount-store").asInstanceOf[KeyValueStore[String, Integer]]
    }

    override def process(envelope: IncomingMessageEnvelope,
                         collector: MessageCollector, coordinator: TaskCoordinator) {

        val words = envelope.getMessage.asInstanceOf[String].split(" ")

        words.foreach{ key =>
            val count: Integer = Option(store.get(key)).getOrElse(0)
            store.put(key, count + 1)
            collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wordcount"), (key, count)))
        }
    }
}

class WordCountTask extends StreamTask with InitableTask {

    private var store: CountStore = _

    def init(config: Config, context: TaskContext) {
        this.store = context.getStore("wordcount-store").asInstanceOf[KeyValueStore[String, Integer]]
    }

    override def process(envelope: IncomingMessageEnvelope, collector: MessageCollector, coordinator: TaskCoordinator) {

        val words = envelope.getMessage.asInstanceOf[String].split(" ")

        words.foreach{ key =>
            val count: Integer = Option(store.get(key)).getOrElse(0)
            store.put(key, count + 1)
            collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wordcount"), (key, count)))
        }
    }
}
val env = ExecutionEnvironment.getExecutionEnvironment

val text = env.fromElements(...)
val words = text.flatMap ( _.split(" ") )

words.keyBy(x => x).mapWithState{
  (word, count: Option[Int]) =>
  {
    val newCount = count.getOrElse(0) + 1
    val output = (word, newCount)
    (output, Some(newCount))
  }
}

...

val env = ExecutionEnvironment.getExecutionEnvironment
val text = env.fromElements(...)
val words = text.flatMap( _.split(" ") )

words.keyBy(x => x).mapWithState{
  (word, count: Option[Int]) =>
  {
    val newCount = count.getOrElse(0) + 1
    val output = (word, newCount)
    (output, Some(newCount))
  }
}

...

Flink
Performance

- Hard to design not biased test, lots of variables
- Latency vs. Throughput
  - 500k/node/sec is ok, 1M/node/sec is nice and >1M/node/sec is great
  - Micro-batching latency usually in seconds, native in millis
- Guarantees
- Fault-tolerance cost
- State cost
- Network operations & Data locality
- Serialization
- Lots of tuning options
Project Maturity

- **Storm & Trident**
  - For a long time de-facto industrial standard
  - Widely used (Twitter, Yahoo!, Groupon, Spotify, Alibaba, Baidu and many more)
  - > 180 contributors

- **Spark Streaming**
  - Around 40% of Spark users use Streaming in Production or Prototyping
  - Significant uptake in adoption (Netflix, Cisco, DataStax, Pinterest, Intel, Pearson, ...)
  - > 720 contributors (whole Spark)

- **Samza**
  - Used by LinkedIn and tens of other companies
  - > 30 contributors

- **Flink**
  - Still emerging, first production deployments
  - > 130 contributors
### Summary

<table>
<thead>
<tr>
<th>Streaming Model</th>
<th>Native</th>
<th>Micro-batching</th>
<th>Micro-batching</th>
<th>Native</th>
<th>Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Compositional</td>
<td>Declarative</td>
<td>Compositional</td>
<td>Declarative</td>
<td></td>
</tr>
<tr>
<td>Guarantee</td>
<td>At-least-once</td>
<td>Exactly-once</td>
<td>Exactly-once</td>
<td>At-least-once</td>
<td>Exactly-once</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Record ACKs</td>
<td>RDD based Checkpointing</td>
<td>Log-based</td>
<td>Checkpointing</td>
<td></td>
</tr>
<tr>
<td>State Management</td>
<td>Not in-build</td>
<td>Dedicated Operators</td>
<td>Dedicated DStream</td>
<td>Stateful Operators</td>
<td>Stateful Operators</td>
</tr>
<tr>
<td>Latency</td>
<td>Very Low</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>Throughput</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>Maturity</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
<td></td>
</tr>
</tbody>
</table>
General Guidelines

- As always it depends
- Always evaluate particular application needs
- Fully understand internals improper use may have disastrous consequences
- Prefer High Level API
- Usually wants exactly once delivery
- Almost all non-trivial jobs have state
- Fast recovery is critical
  - Use Chaos Monkey or similar tool to be sure
Framework Recommendations

- **Storm & Trident**
  - Great fit for small and fast tasks
  - Very low (tens of milliseconds) latency
  - State & Fault tolerance degrades performance significantly
  - Potential update to Heron
    - Keeps API, according to Twitter better in every single way
    - Future open-sourcing is uncertain

- **Spark Streaming**
  - If Spark is already part of your infrastructure
  - Take advantage of various Spark libraries
  - Lambda architecture
  - Latency is not critical
  - Micro-batching limitations
Framework Recommendations

- **Samza**
  - Kafka is a cornerstone of your architecture
  - Application requires large states
  - At least once delivery is fine

- **Flink**
  - Conceptually great, fits very most use cases
  - Take advantage of batch processing capabilities
  - Need a functionality which is hard to implement in micro-batch
  - Enough courage to use emerging project
Dataflow and Apache Beam
Thank you

- Jobs at www.cakesolutions.net/careers
- Mail petrz@cakesolutions.net
- Twitter @petr_zapletal