Distributed Real-Time Stream Processing: Why and How

Petr Zapletal
@petr_zapletal

CAKESOLUTIONS

NE Scala 2016

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
    - …

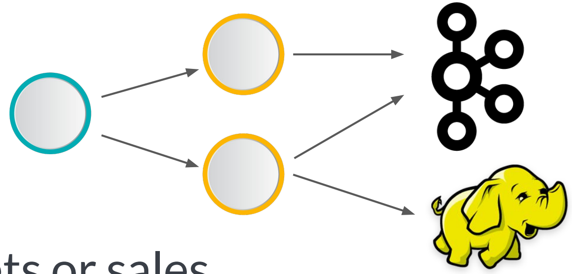
Stream Processing to the Rescue

- 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

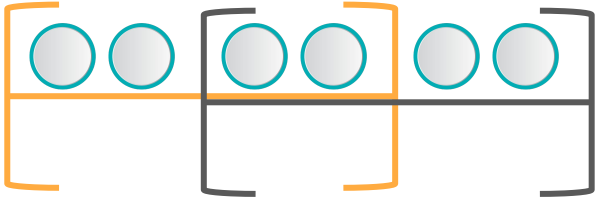
ETL Operations

- Transformations, joining or filtering of incoming data



Windowing

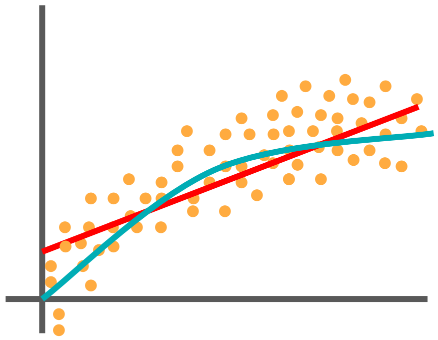
- Trends in bounded interval, like tweets or sales



Streaming Applications

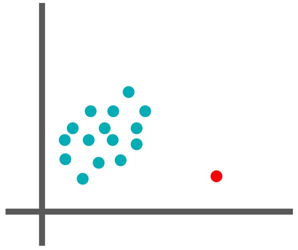
Machine Learning

- Clustering, Trend fitting, Classification



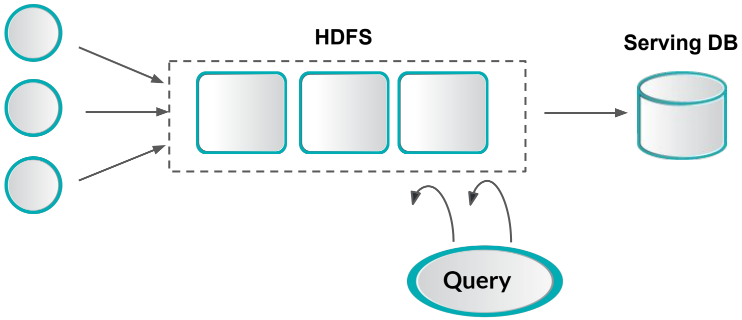
Pattern Recognition

- Fraud detection, Signal triggering, Anomaly detection

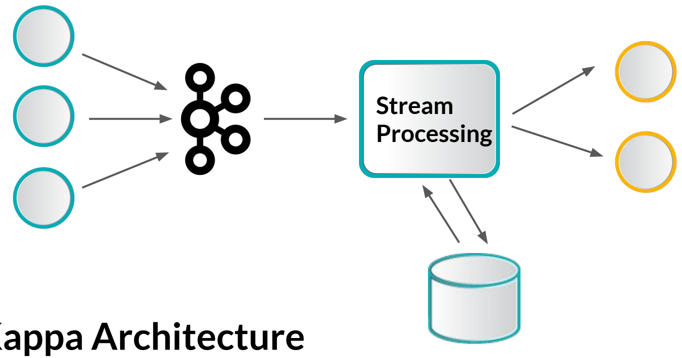


Processing Architecture Evolution

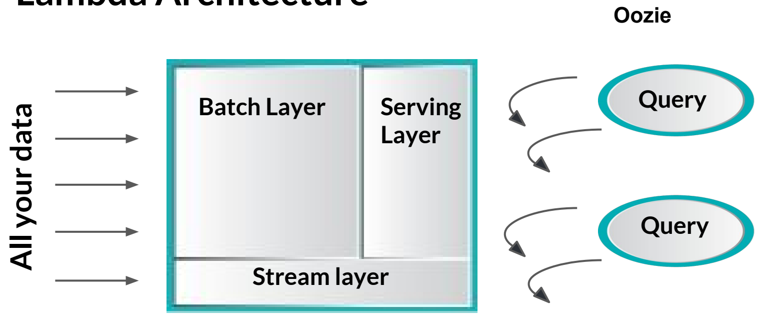
Batch Pipeline



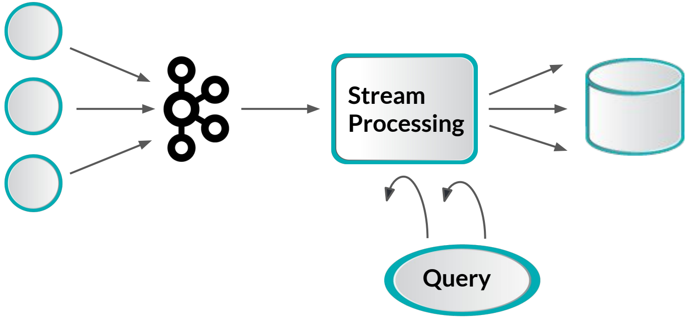
Standalone Stream Processing



Lambda Architecture

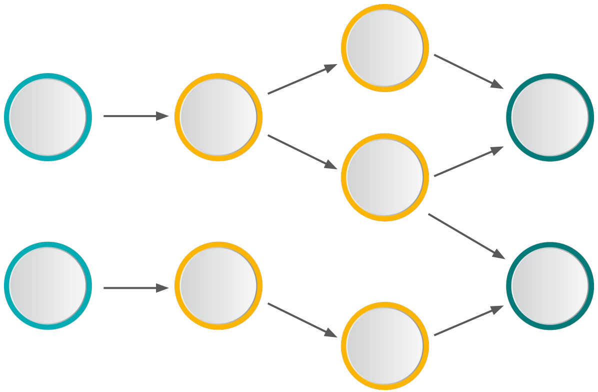


Kappa Architecture



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

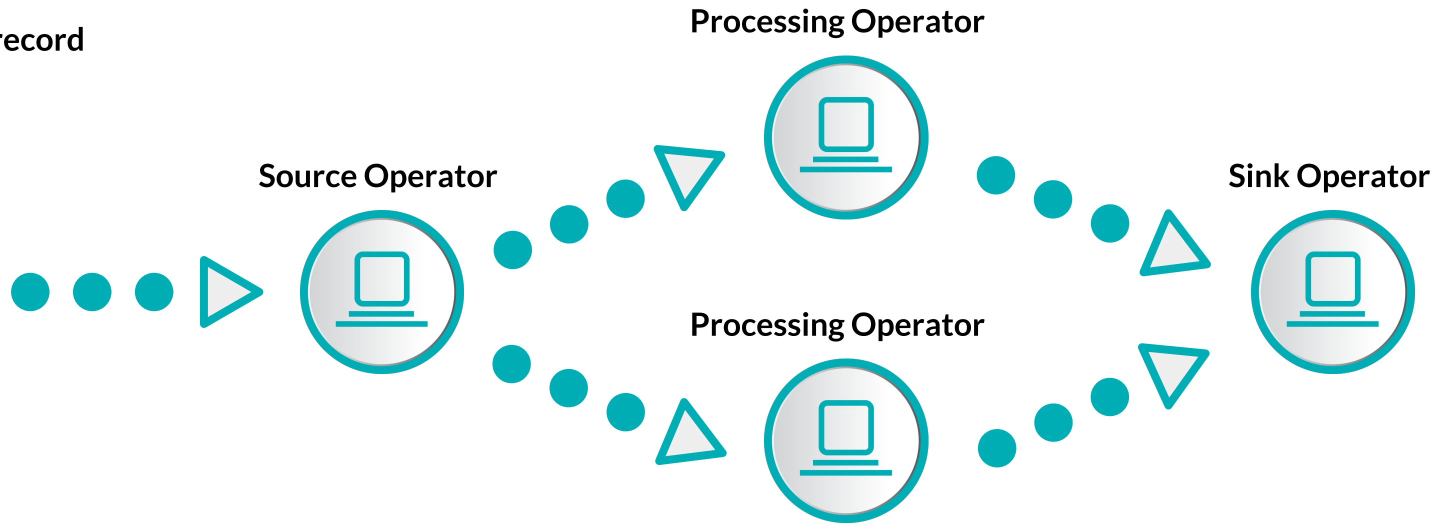
Runtime and Programming Model

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

Native Streaming

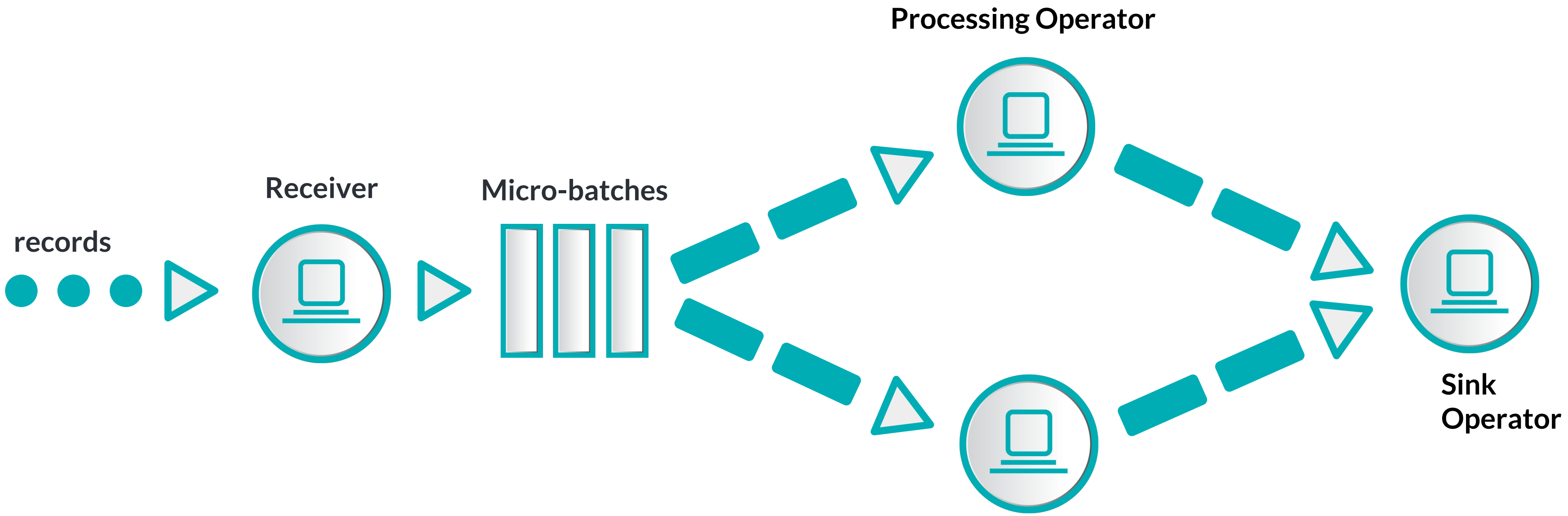
Native stream processing systems
continuous operator model

record



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

Compositional

- Provides basic building blocks as operators or sources
- Custom component definition
- Manual Topology definition & optimization
- Advanced functionality often missing

Declarative

- High-level API
- Operators as higher order functions
- Abstract data types
- Advance operations like state management or windowing supported out of the box
- Advanced optimizers

Apache Streaming Landscape

STORM

TRIDENT STORM

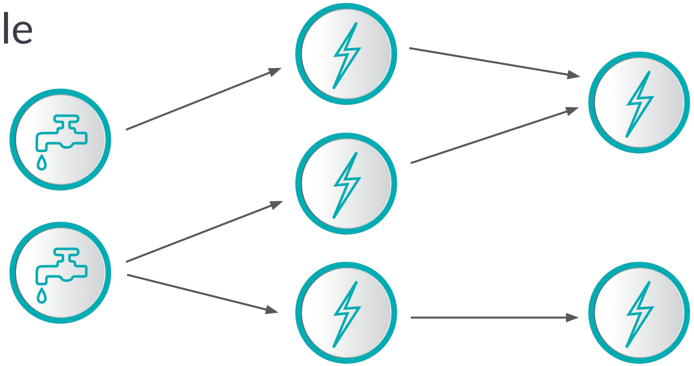
Spark Streaming

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

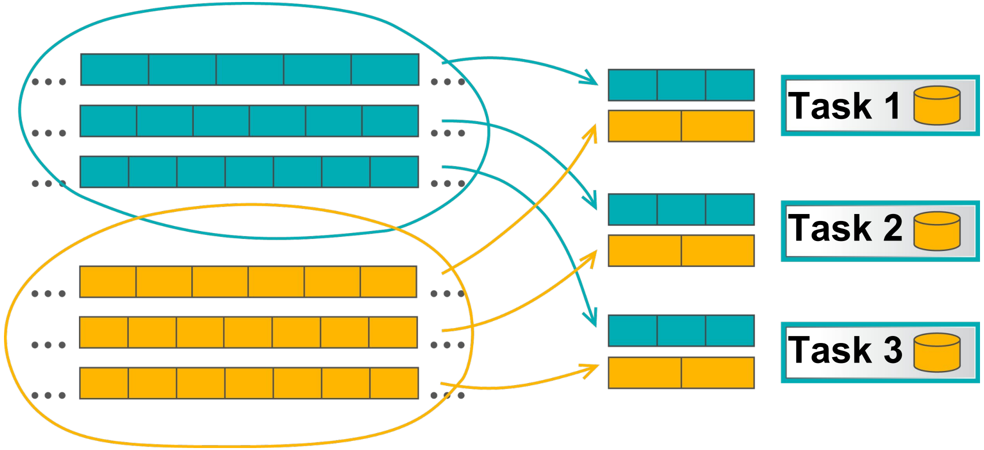
- Higher level micro-batching system build atop Storm
- Stream is partitioned into a small batches
- Simplifies building topologies
- Java, Clojure and Scala API
- Provides exactly once delivery
- Adds higher level operations
    - Aggregations
    - State operations
    - Joining, merging , grouping, windowing, etc.

Spark Streaming

- 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

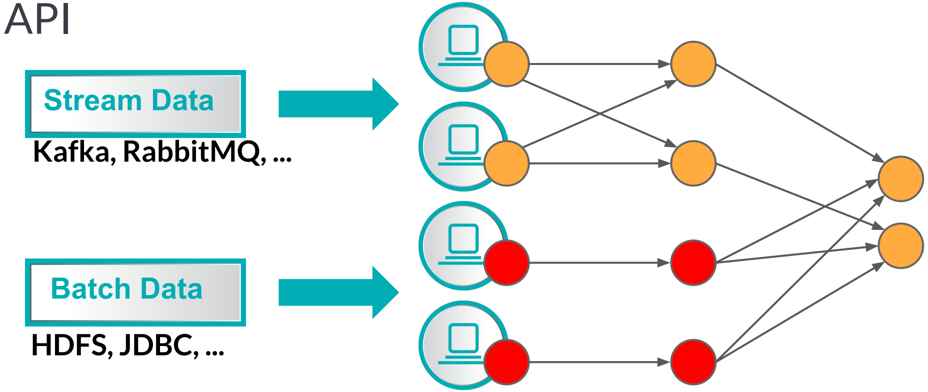
Samza

- 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

		TRIDENT	Spark Streaming	samza	
Streaming Model	Native	Micro-batching	Micro-batching	Native	Native
API	Compositional		Declarative	Compositional	Declarative
Guarantees	At-least-once	Exactly-once	Exactly-once	At-least-once	Exactly-once
Fault Tolerance	Record ACKs		RDD based Checkpointing	Log-based	Checkpointing
State Management	Not build-in	Dedicated Operators	Dedicated DStream	Stateful Operators	Stateful Operators
Latency	Very Low	Medium	Medium	Low	Low
Throughput	Low	Medium	High	High	High
Maturity	High		High	Medium	Low

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)

Storm

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new Split(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));

   ...

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

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new Split(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));
```

```
   ...

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

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new Split(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));

   ...
```

```
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));
}
```

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"));

   ...

}
```

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"));

   ...

}
```

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

```
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

```
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

```
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

```
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()
```

Samza

```
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))

  }
```

Samza

```
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))

  }
```

Samza

```
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))

  }
```

Flink

```
val env = ExecutionEnvironment.getExecutionEnvironment

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

counts.print()

env.execute("wordcount")
```

Flink

```
val env = ExecutionEnvironment.getExecutionEnvironment

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

counts.print()

env.execute("wordcount")
```

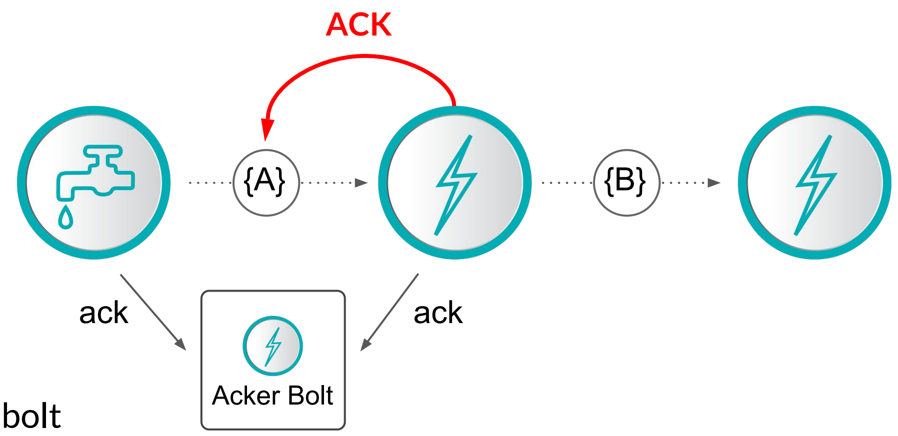
Fault Tolerance

- Fault tolerance in streaming systems is inherently harder that 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

Storm & Trident

- 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

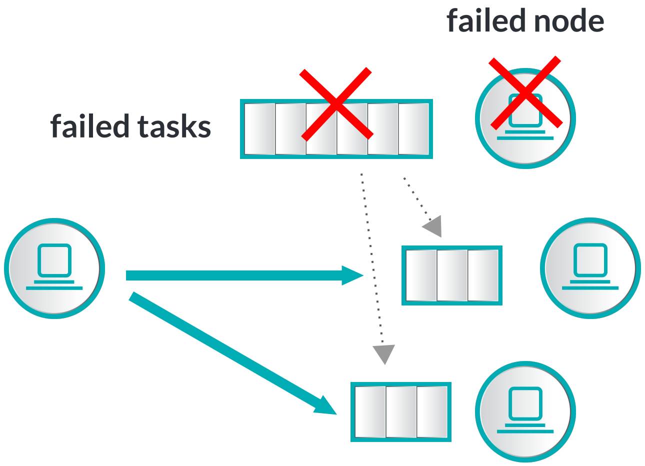
Reliable Processing



Acks are delivered via a system-level bolt

Spark Streaming

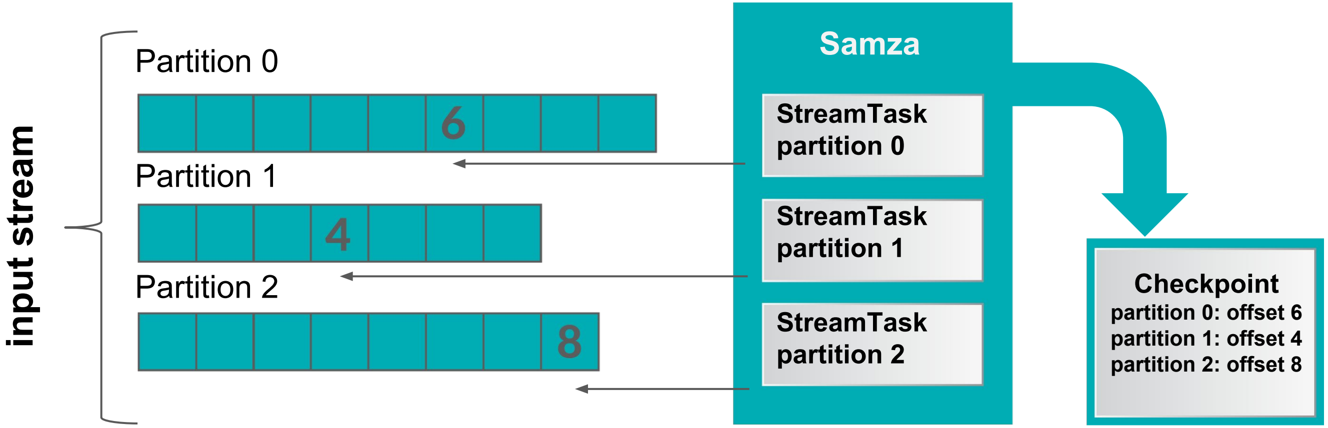
- Failed DStreams can be recomputed using their lineage
- Checkpointing to persistent data storage
    - Reduce lineage length
    - Recover metadata
- Computation redistributed from failed node
- Similar to restarting Spark's failed batch job



faster recovery by using multiple nodes for recomputations

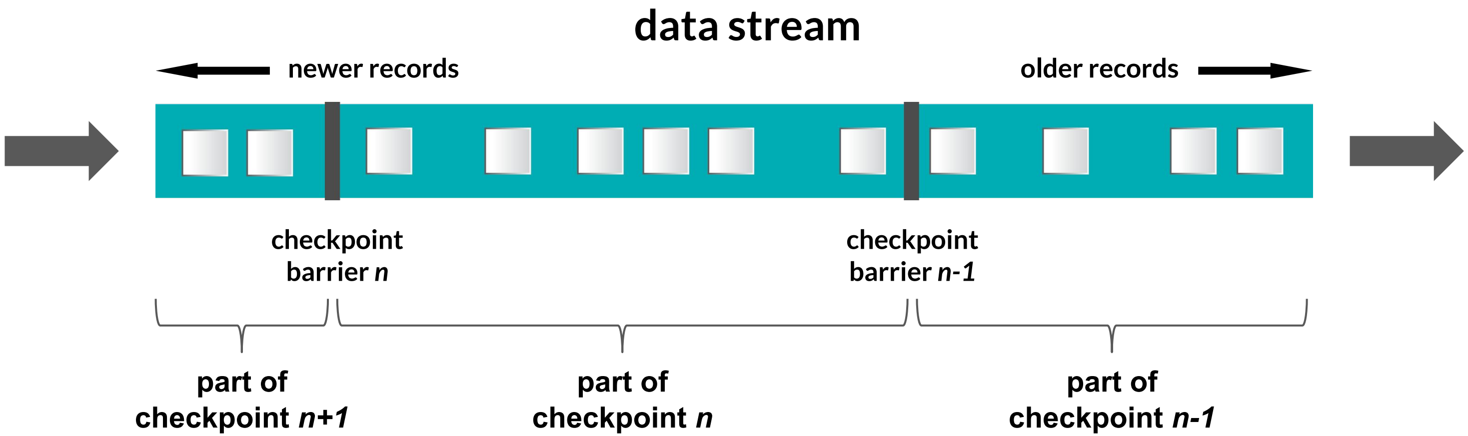
Samza

- 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



Managing State

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

  f: (input, state) -> (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

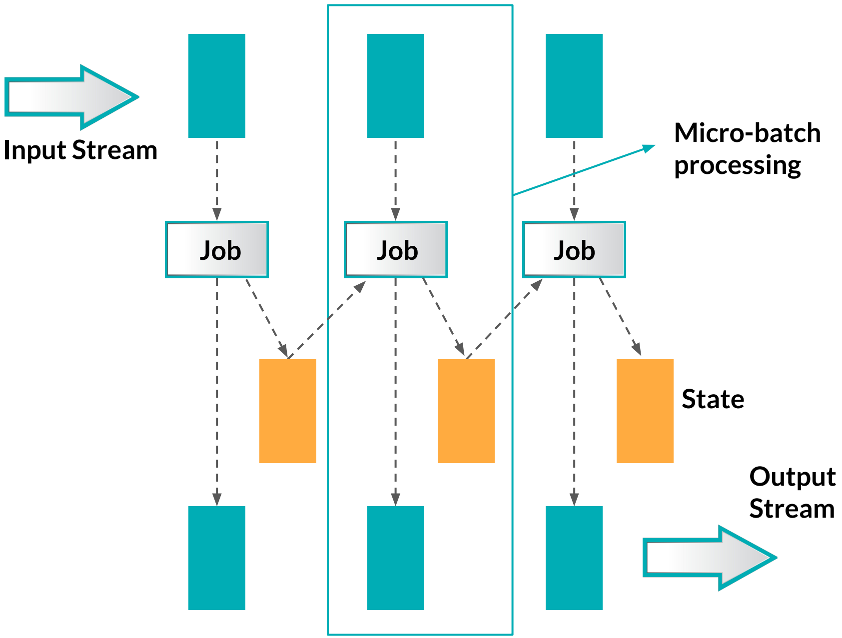
Storm & Trident

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

		State		
Spout		Non-transactional	Transactional	Opaque transactional
	Non-transactional	No	No	No
	Transactional	No	Yes	Yes
	Opaque transactional	No	No	Yes

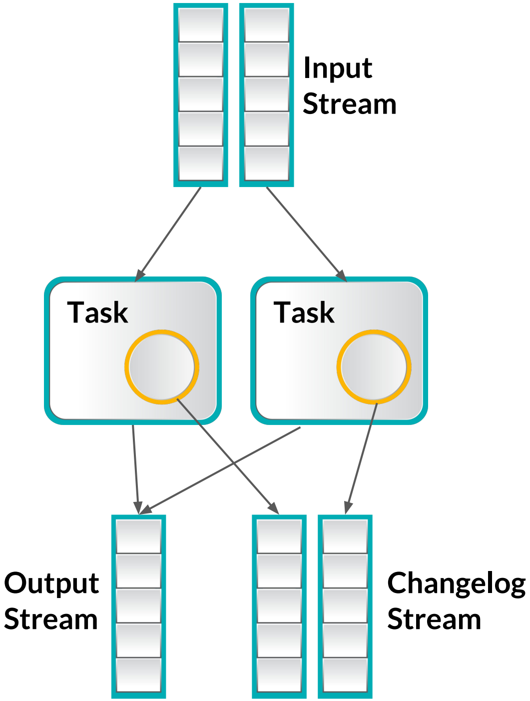
Spark Streaming

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



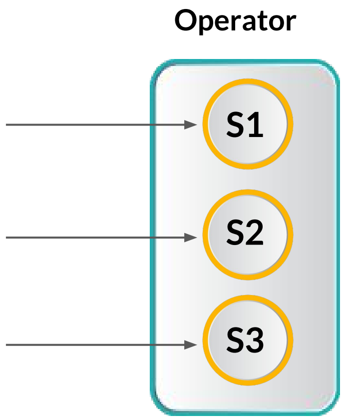
Samza

- 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



Flink

- 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)

Trident

```
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);
```

Trident

```
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))
```

Samza

```
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)))
    }
  }
```

Samza

```
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)))
    }
  }
```

Samza

```
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)))
    }
  }
```

Flink

```
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

```
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))
    }
}

...
```

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

		TRIDENT	Spark Streaming	samza	
Streaming Model	Native	Micro-batching	Micro-batching	Native	Native
API	Compositional		Declarative	Compositional	Declarative
Guarantee	At-least-once	Exactly-once	Exactly-once	At-least-once	Exactly-once
Fault Tolerance	Record ACKs		RDD based Checkpointing	Log-based	Checkpointing
State Management	Not in-build	Dedicated Operators	Dedicated DStream	Stateful Operators	Stateful Operators
Latency	Very Low	Medium	Medium	Low	Low
Throughput	Low	Medium	High	High	High
Maturity	High		High	Medium	Low

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

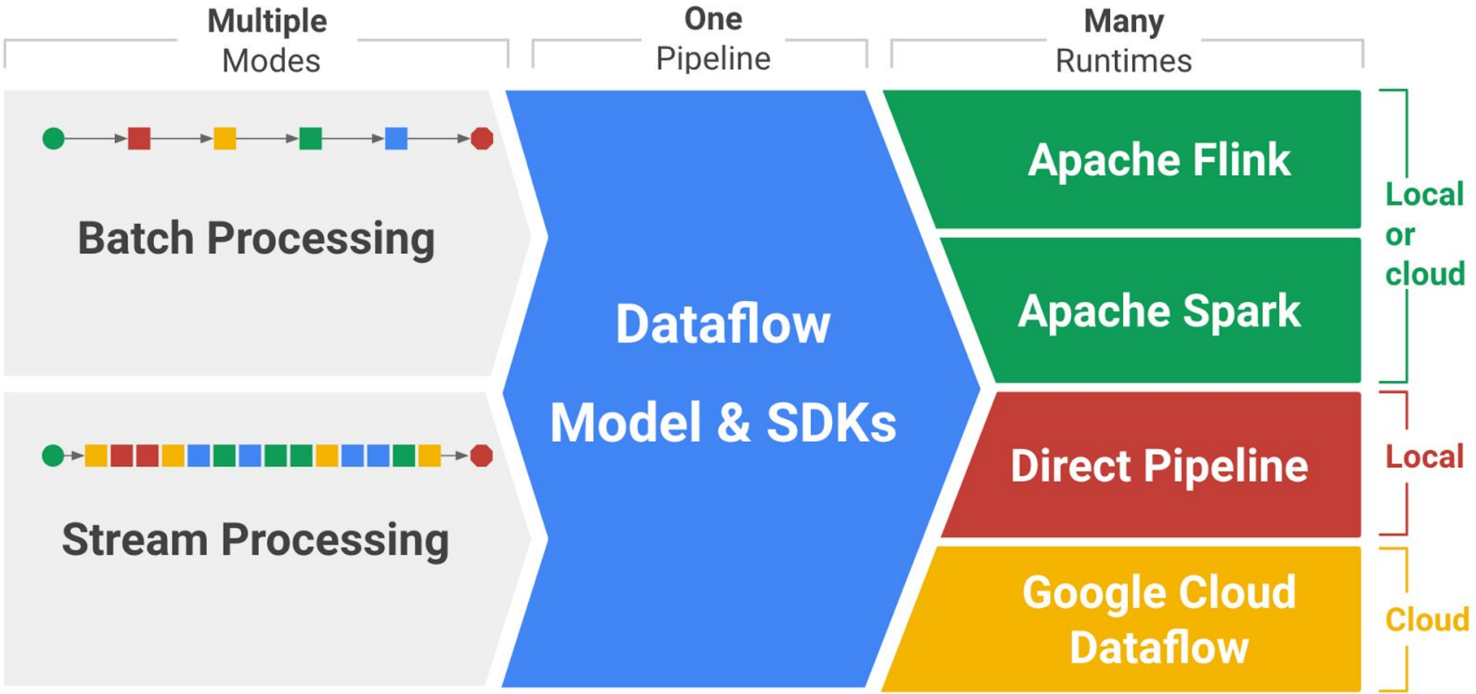
Framework Recommendations

- 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



Questions

Thank you

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