Lab 3: Introduction to Spark Streaming

Michalis Giannoulis

Outline

- Big Stream Analysis
- Streaming & Real Time Processing
- Streaming Systems
- λ vs k
- What is Spark Streaming
- Core Functionality
- Core Functionality Example
- Performance of Spark Streaming
- Real World Application Example (Twitter)

Big Stream Analysis

Big Fast Data

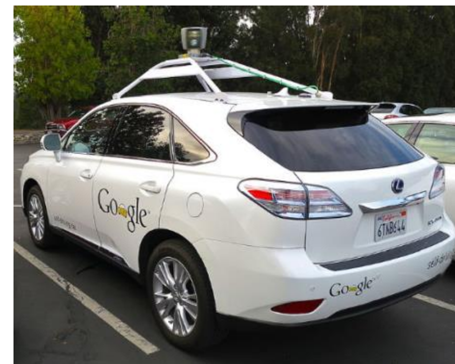
- Data is growing and can be evaluated
    - Tweets, social networks (statuses, check-ins, shared content), blogs, click streams, various logs, …
    - Facebook: > 845M active users, > 8B messages/day
    - Twitter: > 140M active users, > 340M tweets/day

Image: Michael Carey

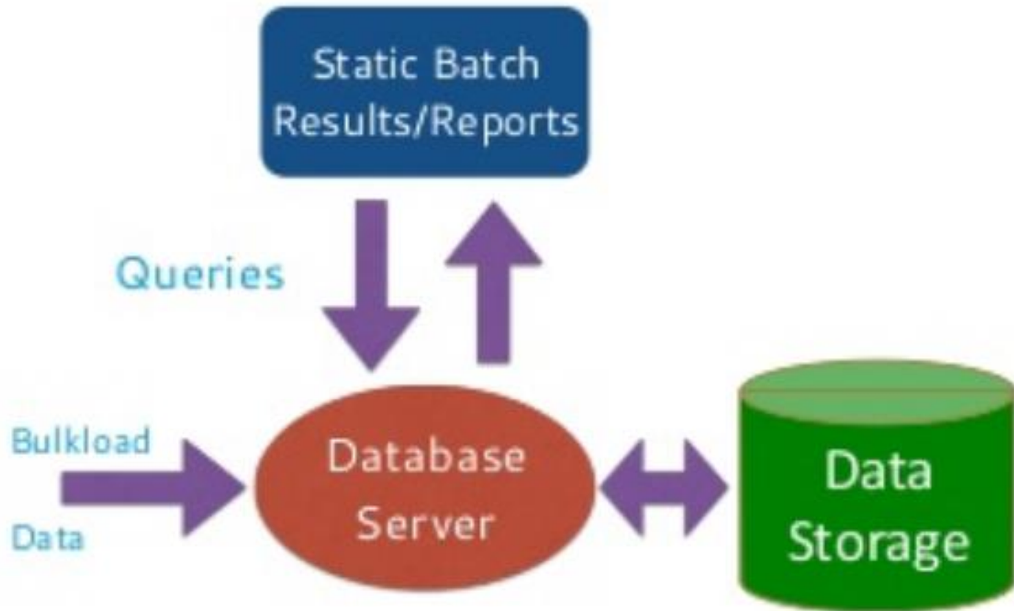
- Everyone is interested!

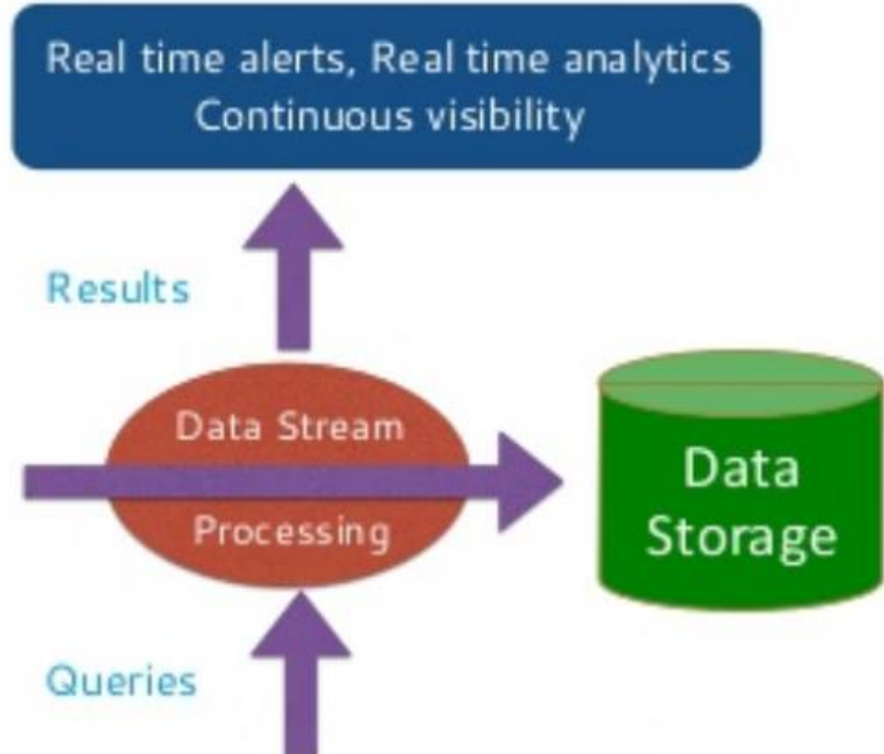
But there is so much more…

- Autonomous Driving
  - Requires rich navigation info
  - Rich data sensor readings
  - 1GB data per minute per car (all sensors)
- Traffic Monitoring
  - High event rates: millions events / sec
  - High query rates: thousands queries / sec
  - Queries: filtering, notifications, analytical
- Pre-processing of sensor data
  - CERN experiments generate ~1PB of measurements per second
  - Unfeasible to store or process directly, fast preprocessing is a must



Interactive vs Streaming Analytics



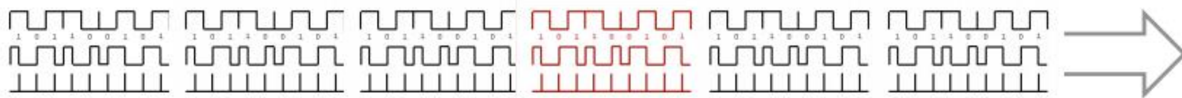


- https://www.slideshare.net/arunkejariwal/velocity-2015final

Big *Streaming* Data Processing

Fraud detection in bank transactions

Anomalies in sensor data



Cat videos in tweets

DATABRICKS

Requirements for Stream Data Processing Systems

- Automatic partitioning and distributed processing
  - Scales to large clusters (hundreds of nodes)
- Instantaneous processing and response
  - Achieves second-scale latency
- Handle data imperfections
  - Late, missing, unordered items
- Predictable outcomes (consistency, event time)
- Data safety and availability
  - Efficient fault tolerance in stateful computations
- Hybrid stream and batch (or interactive) processing

Why is this hard?

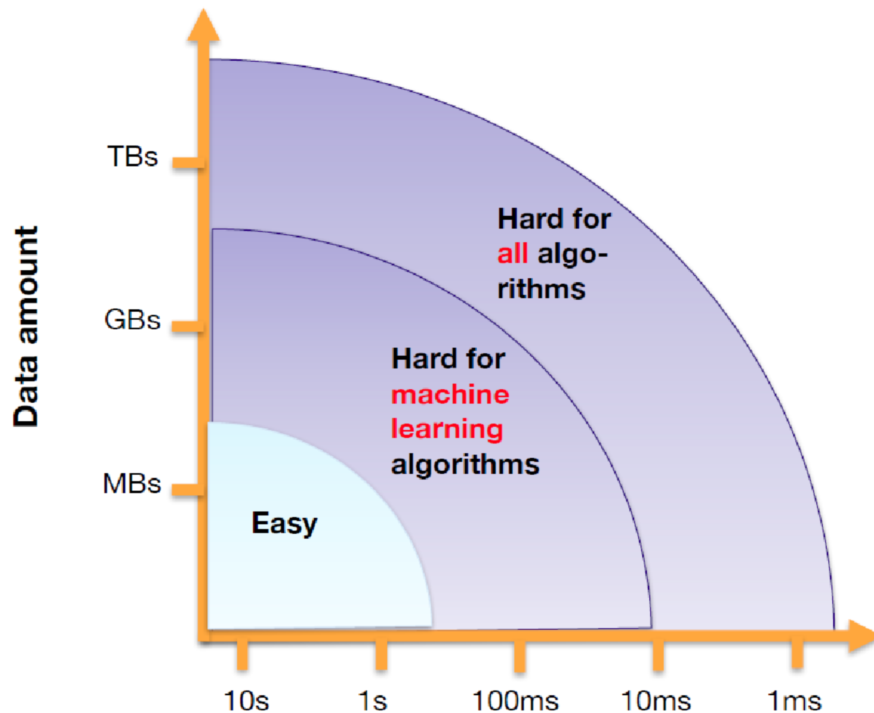


Image: Peter Pietzuch

- Tension between performance and algorithmic expressiveness

Streaming and Real Time Processing

Streaming and Real Time Processing

- **Online Processing**: a method that continuously process data as they flow through the system
  - no compulsory time limitations

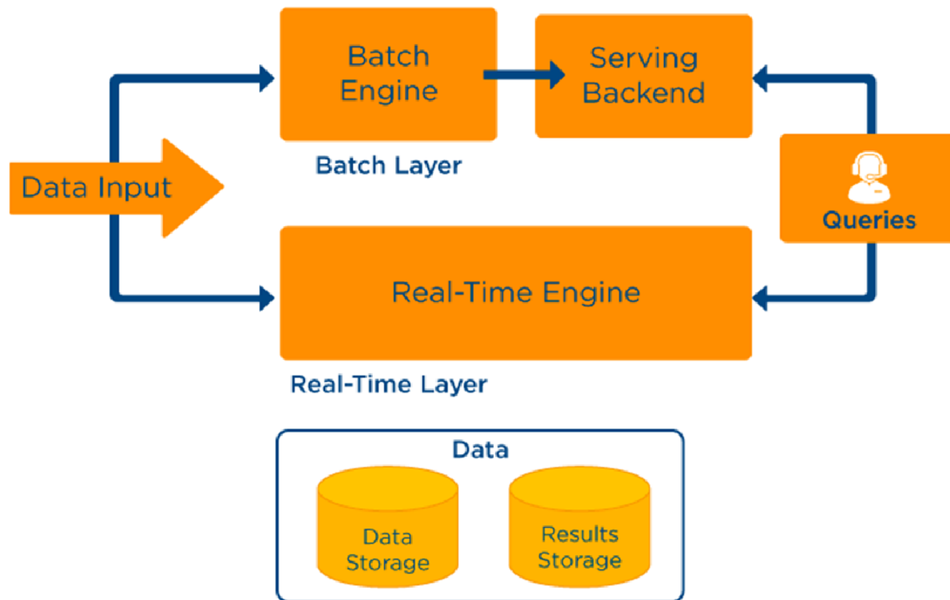
- **Real time Processing**: a method that process real-time data under tight deadlines in terms of time
  - Capturing events within 1 ms, is called real-time data of true streaming

(Near) Real-time Data Pipelines

- Lambda architecture: have 3 layers (Batch, Speed, Serving) to provide real-time streaming and compensate any data error occurs

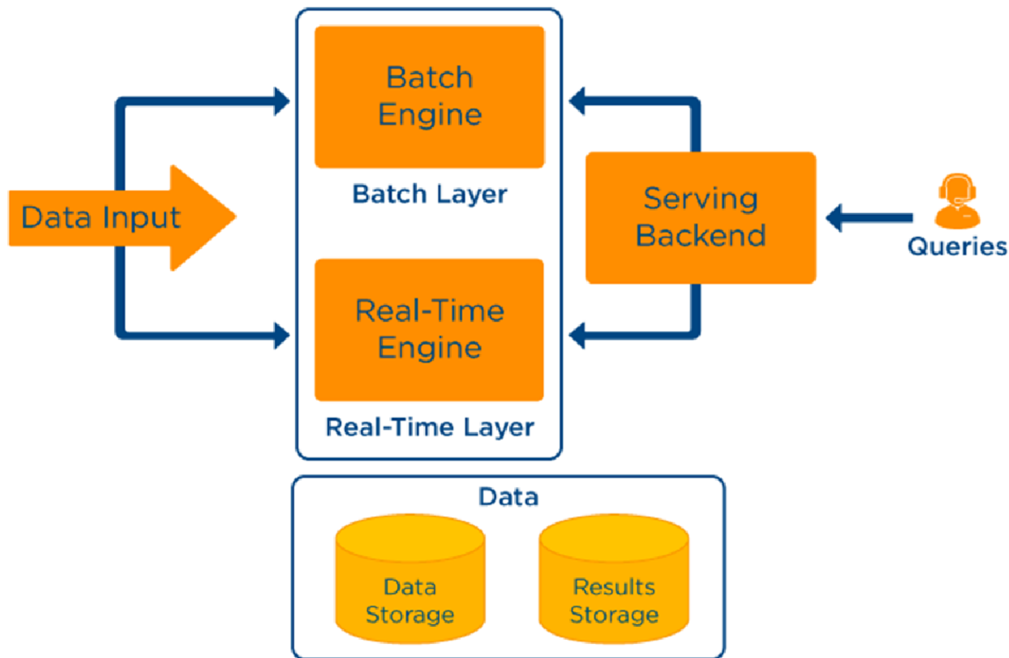
- Kappa architecture: handle both real-time and continuous data processing using a stream processing engine
  - avoids maintaining two separate code bases for the batch and speed layers

Lambda Architecture



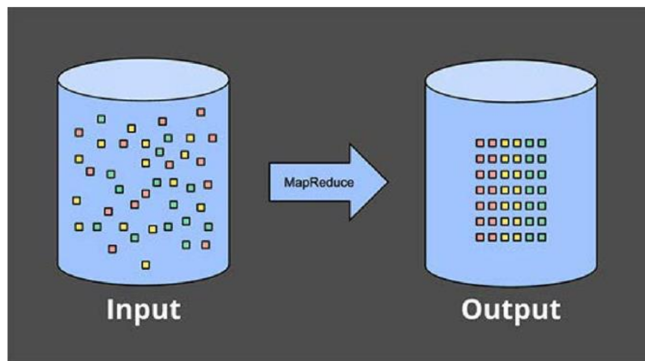
- The batch layer has two major tasks: (a) managing historical data; (b) re-computing results such as ML models
- The speed layer provides results in a low-latency, near real time fashion

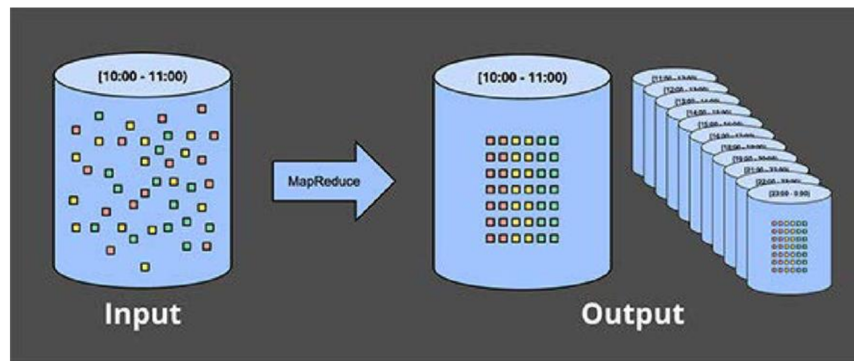
Kappa Architecture



- The key idea is to handle both real time processing and continuous data reprocessing using a single stream processing engine

Why MR is not a Solution for Fast Big Data



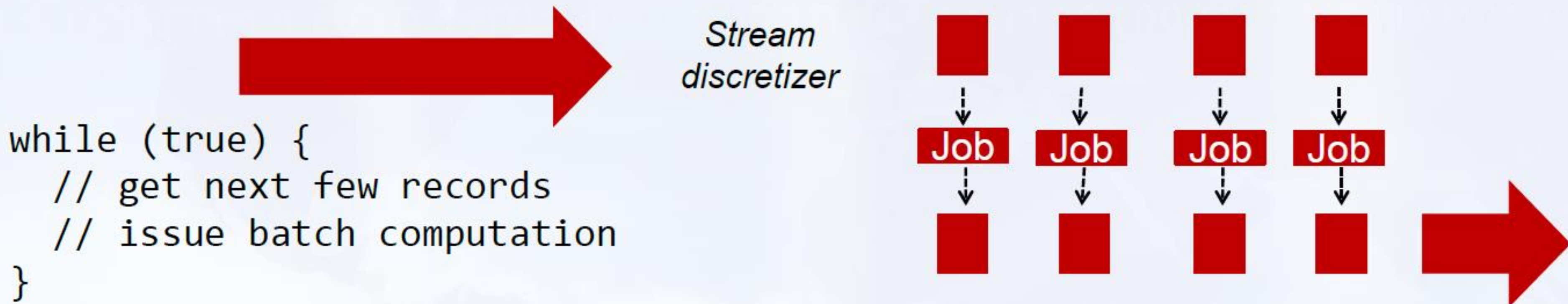


Images: Tyler Akidau

- Great for large amounts of static data
  - Data is not moving!
- For streams: only for large windows
- High latency, low efficiency

How to keep data moving?

Discretized Streams (mini-batch)



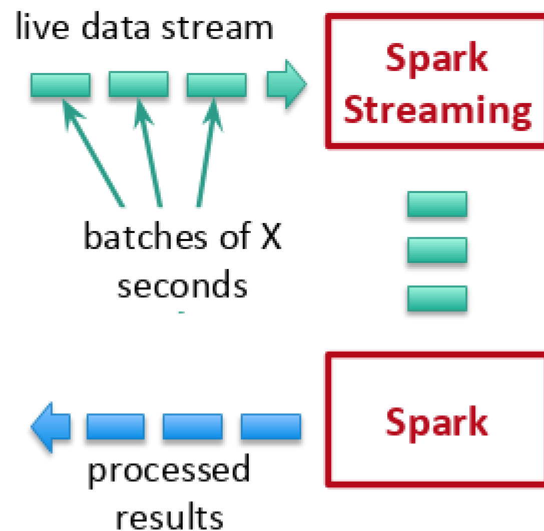
Native streaming

```
while (true) {
   // process next record
}
```

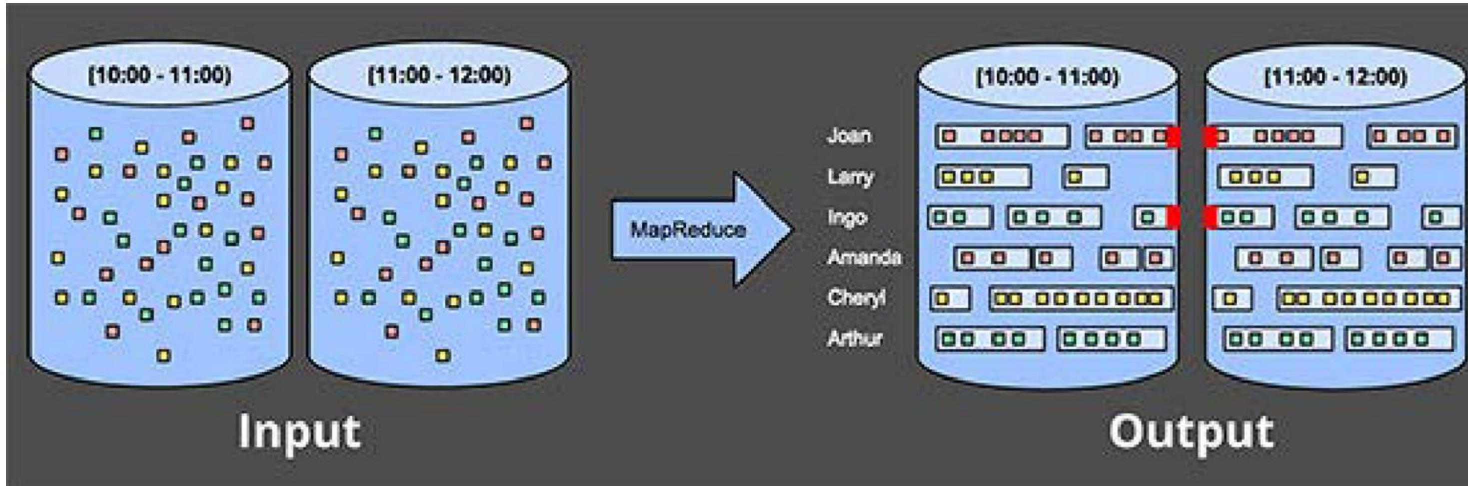
Long-standing operators

Discretized Stream Processing

- Run a streaming computation as a series of very small deterministic batch jobs
  - Chop the live stream into batches of X seconds
  - Batch sizes as low as ½ second, latency of about 1 second



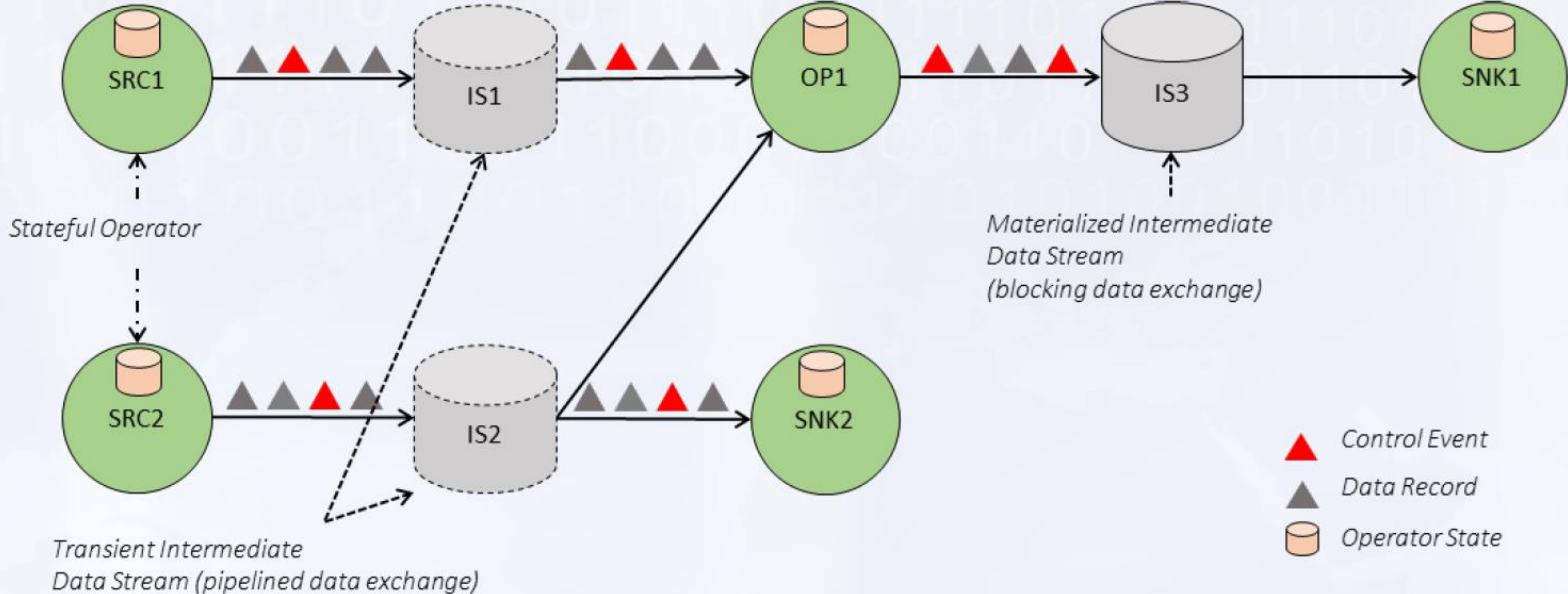
Discussion of Mini-Batch



Images: Tyler Akidau

- Easy to implement
- Easy consistency and fault-tolerance
- Potential for combining stream with batch processing in the same system
- Hard to do event time and sessions

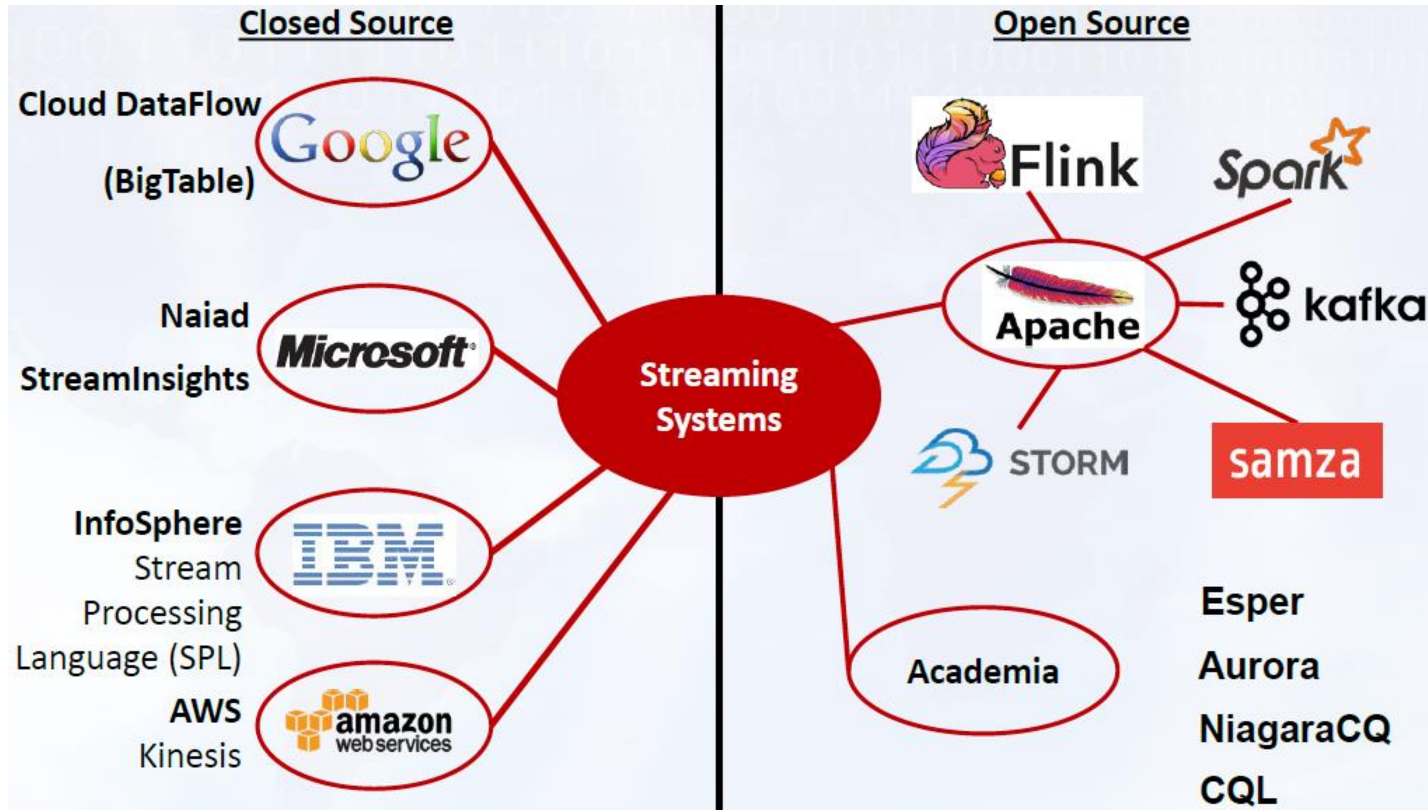
Fully Fledged Streaming Architecture



- Program = DAG* of operators and intermediate streams
- Operator = computation + state
- Intermediate streams = logical stream of records

Streaming Systems

Data Stream Processing Systems Overview



Closed Source/Commercial Systems

Cloud DataFlow:
- Unified primitives for batch and stream processing
- Runs in Google's cloud only
- Open Source SDK (programs can run on other systems)
- Check out the Apache Beam Project! (http://beam.apache.org/)

BigTable:
- Not a real streaming solution
- Allows to feed streams as source into a google DB
- Data can be immediately queried

Naiad:
- Goals of Naiad:
  - High throughput (typical for batch processors)
  - Low latency (known from single system stream processors)
- Is able to process iterative data flows
- Can discretize windows only based on time

StreamInsights:
- Available through Microsoft's cloud
- Windows based on count-, time- and punctuation/snapshot
- Optimized for .NET framework applications

InfoSphere: Stream Processing Language (SPL)
- Well specified in several publications
- Can be deployed in customer clusters
- Own SQL-like query language enables many optimization means
- window discretization based on trigger- and eviction policies

Open Source Systems by Apache (1/2)

- Reliable handling of huge numbers of concurrent reads and writes
- Can be used as data-source / data-sink for Storm, Samza, Flink, Spark and many more systems
- Fault tolerant: Messages are persisted on disk and replicated within the cluster. Messages (reads and writes) can be repeated

- True streaming over distributed dataflow
- Low level API: Programmers have to specify the logic of each vertex in the flow graph
- Full understanding and hard coding of all used operators is required
- Enables very high throughput (single purpose programs with small overhead)

samza

- True streaming built on top of Apache Kafka and Hadoop YARN
- State is first class citizen
- Low level API

Open Source Systems by Apache (2/2)

Spark implements a batch execution engine

- The execution of a job graph is done in stages
- Operator outputs are materialized in memory (or disk) until the consuming operator is ready to consume the materialized data

Spark uses Discretized Streams (D-Streams)

- Streams are interpreted as a series of deterministic batch-processing jobs
- Micro batches have a fixed granularity
- All windows defined in queries must be multiples of this granularity

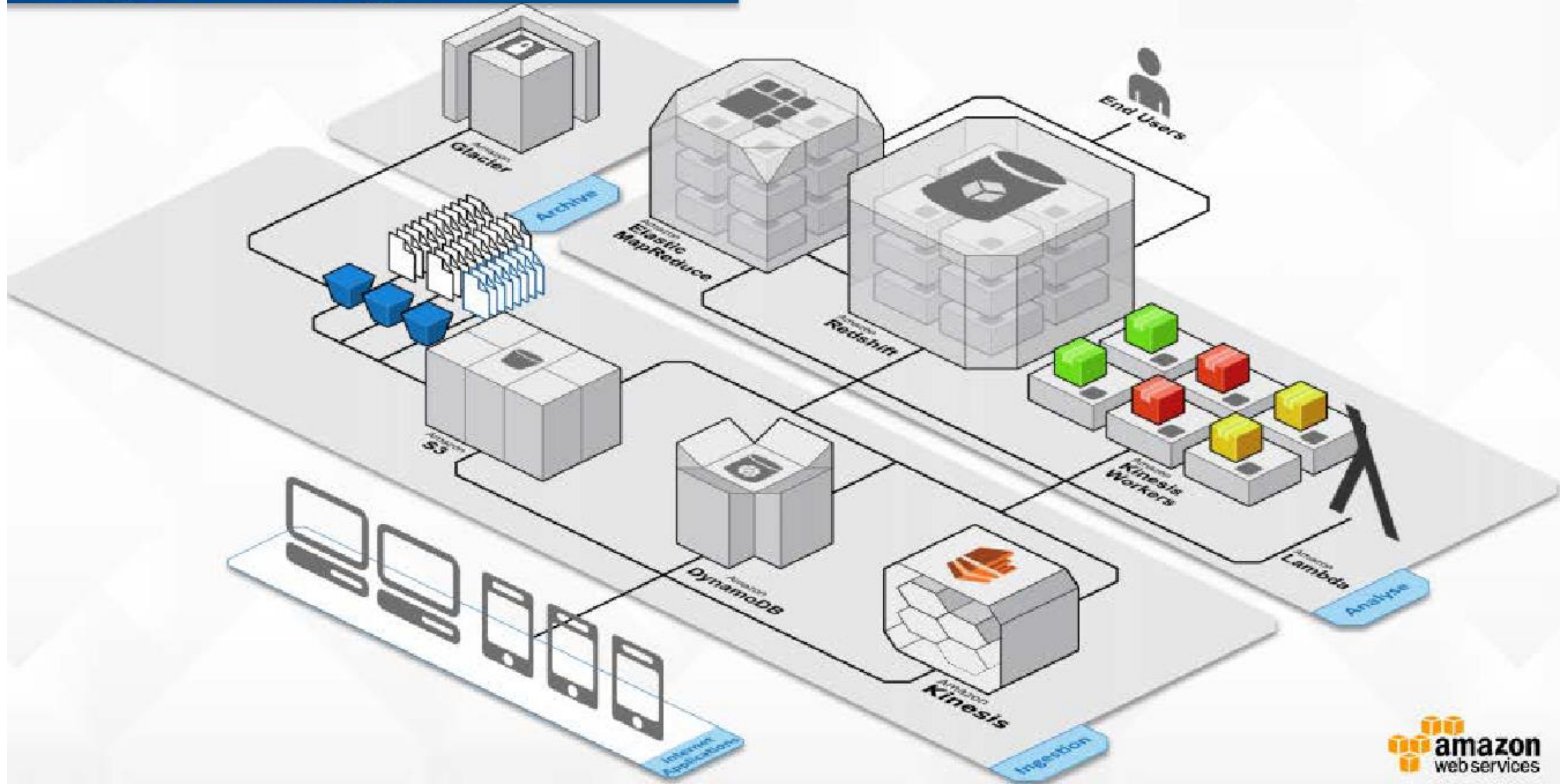
Flinks runtime is a native streaming engine

- Based on Nephele/PACTs
- Queries are compiled to a program in the form of an operator DAG
- Operator DAGs are compiled to job graphs
- Job graphs are generic streaming programs

Flink implements "true streaming"

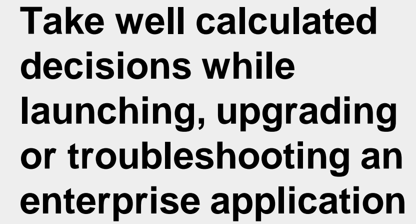
- The whole job graph is deployed concurrently in the cluster
- Operators are long-running: Continuously consume input and produce output
- Output tuples are immediately forwarded to succeeding operators and are available for further processing (enables pipeline parallelism)

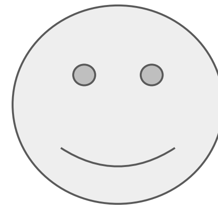
Cloud-Based Streaming Systems (example)



-Rapid growth of social media applications
-Cloud based systems
-Internet of things
-Unending spree of innovations

λ vs κ


Take well calculated decisions while launching, upgrading or troubleshooting an enterprise application


Data Analyst

λ vs k

λ vs k

Dealing with huge amount of data in an efficient manner

Increased throughput, reduced latency and negligible errors

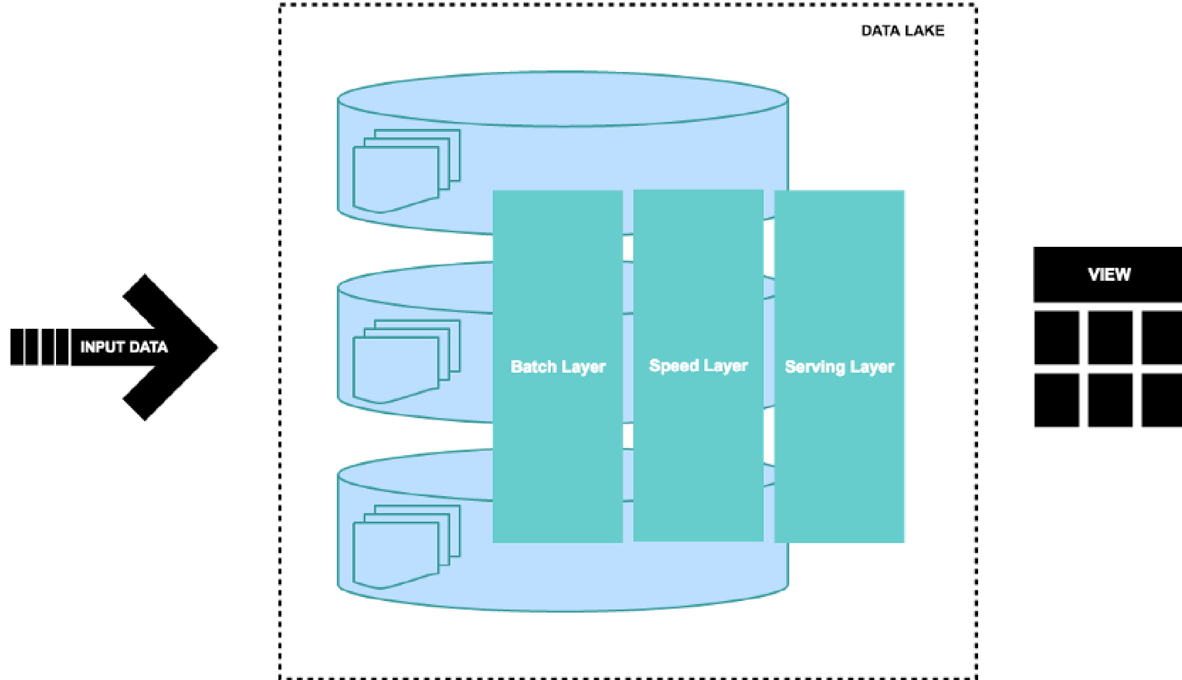
Data processing deals with the **event** streams

Introducing three distinct layers.

Batch Layer, Speed Layer (also known as Stream layer) and Serving Layer

λ VS k

<u>Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)</u>



λ vs κ

Pros and Cons of Lambda Architecture

Pros

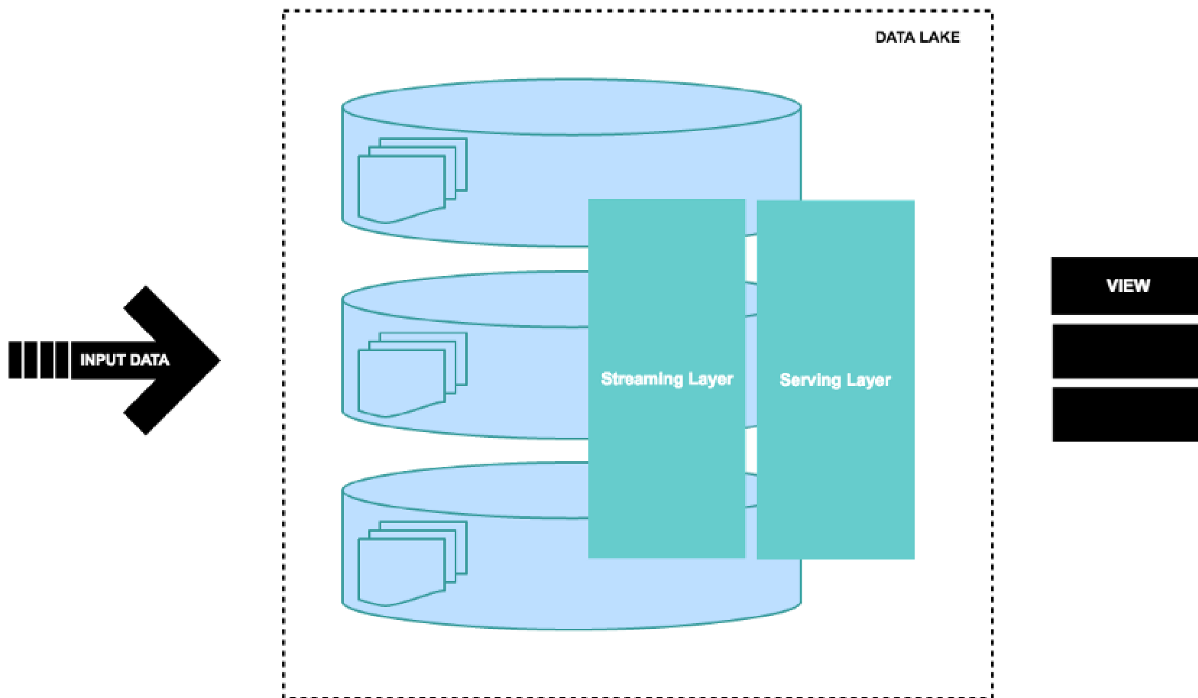
- Batch layer of Lambda architecture manages historical data with the fault tolerant distributed storage which ensures low possibility of errors even if the system crashes.
- It is a good balance of speed and reliability.
- Fault tolerant and scalable architecture for data processing.

Cons

- It can result in coding overhead due to involvement of comprehensive processing.
- Re-processes every batch cycle which is not beneficial in certain scenarios.
- A data modeled with Lambda architecture is difficult to migrate or reorganize.

λ vs k

Query = K (New Data) = K (Live streaming data)



λ vs k

Pros and Cons of Kappa architecture

Pros

- Kappa architecture can be used to develop data systems that are online learners and therefore don’t need the batch layer.
- Re-processing is required only when the code changes.
- It can be deployed with fixed memory.
- It can be used for horizontally scalable systems.
- Fewer resources are required as the machine learning is being done on the real time basis.

Cons

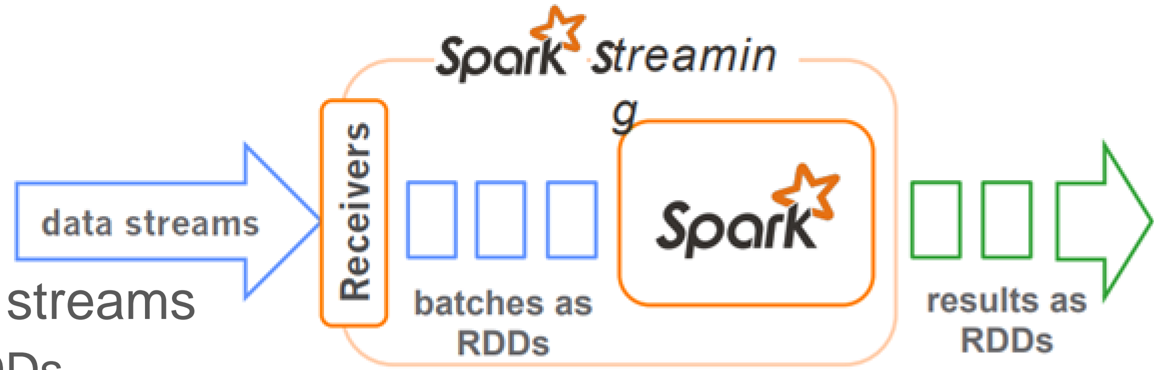
Absence of batch layer might result in errors during data processing or while updating the database that requires having an exception manager to reprocess the data or reconciliation.

Requirements

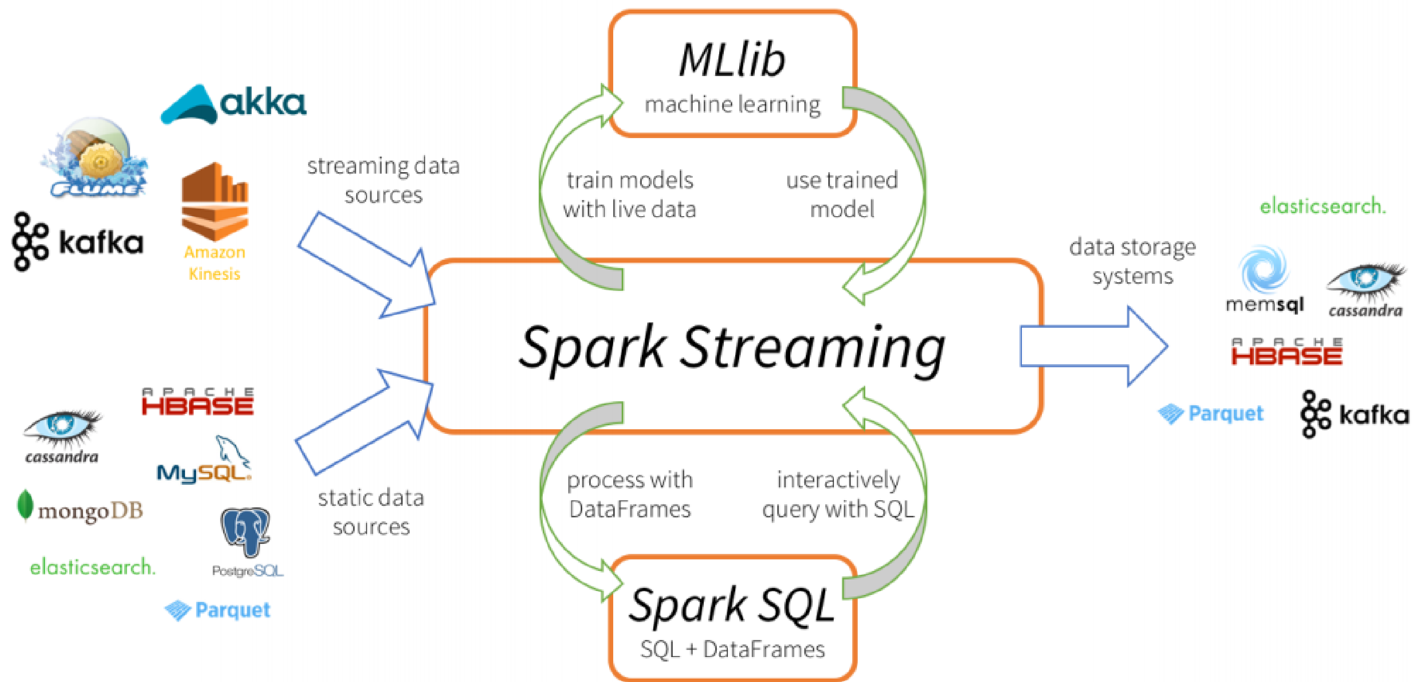
- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing
- **Efficient fault-tolerance** in stateful computations

What is Spark Streaming

Spark Streaming



- Key abstraction: discretized streams
  - micro-batch = series of RDDs
  - stream computation = series of deterministic batch computation at a given time interval
  - processed results are pushed out in micro-batches
- API very similar to Spark core (Java, Scala, Python)
  - (stateless) transformations on DStreams: map, filter, reduce, repartition, cogrop…
  - Stateful operators: time-based window operations, incremental aggregation, time-skewed joins
  - Also DataFrame/SQL and Mlib operations
- Exactly-once semantics using checkpoints (asyn. replication of state RDDs)
- No event time windows

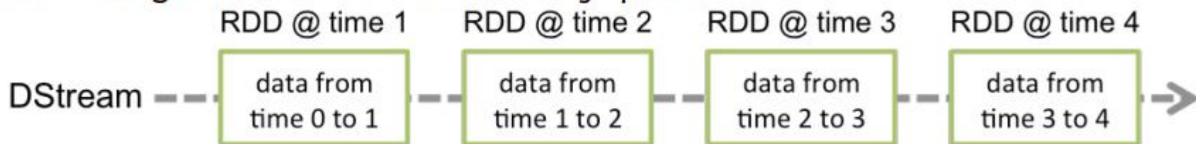


https://www.datanami.com/2015/11/30/spark-streaming-what-is-it-and-whos-using-it/

Core Functionality

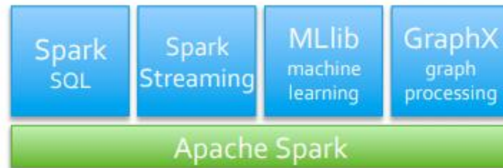
DStream

- The basic high level abstraction for streaming in spark is called discretized stream or DStream
  - represents a continuous stream of data
- DStreams can be created
  - from input data streams from sources such as Kafka, Flume, and Kinesis
  - by applying high-level operations on other DStreams
- A DStream is represented as a sequence of RDDs
- StreamingContext - the main entry point



DStreams + RDDs = Power

- Combine live data streams with historical data
  - Generate historical data models with Spark, etc.
  - Use data models to process live data stream
- Combine streaming with MLlib, GraphX algos
  - Offline learning, online prediction
  - Online learning and prediction



- Interactively query streaming data using SQL
  - select * from table_from_streaming_data

StreamingContext

A StreamingContext object has to be created (batch interval)

- Define the input sources by creating input DStreams
- Define the streaming computations (transformations/output operations) to DStreams
- Start receiving data and processing it (`start()`)
- Wait for the processing to stop (`awaitTermination()`)
- Manually stop using it (`stop()`)

```
import org.apache.spark._
import org.apache.spark.streaming._

val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

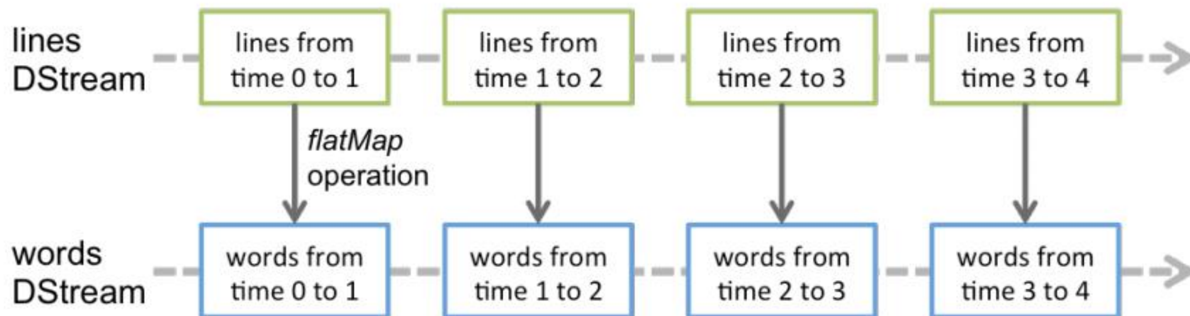
StreamingContext: Remember

- Once a context has been started no new computations can be set up
- Once a context has been stopped it cannot be restarted
- Only one StreamingContext can be active in a JVM at the same time
- stop() also stops the SparkContext (set the optional parameter of stop() called stopSparkContext to false)
- A SparkContext can be re-used to create multiple StreamingContexts as long as there is only one StreamingContext active

Operations applied on DStream

Operations applied on a DStream are translated to operations on the underlying RDDs

<u>Use Case:</u> Converting a stream of lines to words by applying the operation flatMap on each RDD in the "lines DStream".



File Streams

- Besides sockets, the StreamingContext API provides methods for creating DStreams from files
- Reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.)
- Spark Streaming will monitor a directory and process any files created in that directory

Input DStream and Receivers

- Every input DStream except file stream is associated with a `Receiver`
  - `Receiver` receives data from a source and stores it in spark memory
- Two build-in streaming sources
  - Basic Sources: Sources directly available in the Streaming Context API (file systems and socket connections)
  - Advanced Sources: Kafka, Flume, Kinesis (need more dependencies – check the bitbucket repository given at the end for kafka example)
- Reliable and Unreliable receivers (regarding loss of data due to failure)
  - A reliable receiver sends ack to a reliable source when the data has been received and stored in Spark with replication

http://spark.apache.org/docs/2.2.0/streaming-custom-receivers.html

Check-points

A streaming application must be resilient to failures

Two types of check-points

- **Metadata check-pointing**

  Store information regarding the streaming computation to HDFS

  Recovery from failures - configuration, operations, incomplete batches

- **Data check-pointing**

  Store generated RDDs to HDFS

  This is necessary in some stateful transformations

Transformations on DStreams

- DStreams support most of the RDD transformations

`map` `flatMap` `filter` `repartition` `countByValue` `count` `reduce` `union` `reduceByKey` `join` `cogroup`

- Also introduces special transformations related to state & windows

Stateless vs Stateful Operations

- By design streaming operators are stateless
  - they know nothing about any previous batches
- Stateful operations have a dependency on previous batches of data
  - continuously accumulate metadata overtime
  - data check-pointing is used for saving the generated RDDs to a reliable stage

DStreams transformations

`map(func)`	Return a new DStream by passing each element of the source DStream through a function func.
`flatMap(func)`	Similar to map, but each input item can be mapped to 0 or more output items.
`filter(func)`	Return a new DStream by selecting only the records of the source DStream on which func returns true.
`repartition(numPartitions)`	Changes the level of parallelism in this DStream by creating more or fewer partitions.
`union(otherStream)`	Return a new DStream that contains the union of the elements in the source DStream and otherDStream.
`count()`	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.

DStreams transformations (cont.)

`countByValue()`	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
`reduceByKey(func,[numTasks])`	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. Note: By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property spark.default.parallelism) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks.
`join(otherStream, [numTask])`	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
`cogroup(otherStream, [numTask])`	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.

DStreams transformations (cont.)

`transform(func)`	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
`updateStateByKey(func)`	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

UpdateStateByKey Operation

A stateful operation that allows you to maintain arbitrary state while continuously updating it with new information

Three requirements:

- Define the state - The state can be an arbitrary data type
- Define the "*state update function*" used for updating the current state using the previous state and the new values from an input stream
- Regardless of whether they have new data in a batch or not
  - If the update function returns None then the key-value pair will be eliminated.
- Requires check-pointing to be configured

Transform Operation

A stateless operation that allows arbitrary RDD-to-RDD functions to be applied on a DStream

It can be used to apply any RDD operation that is not exposed in the DStream API

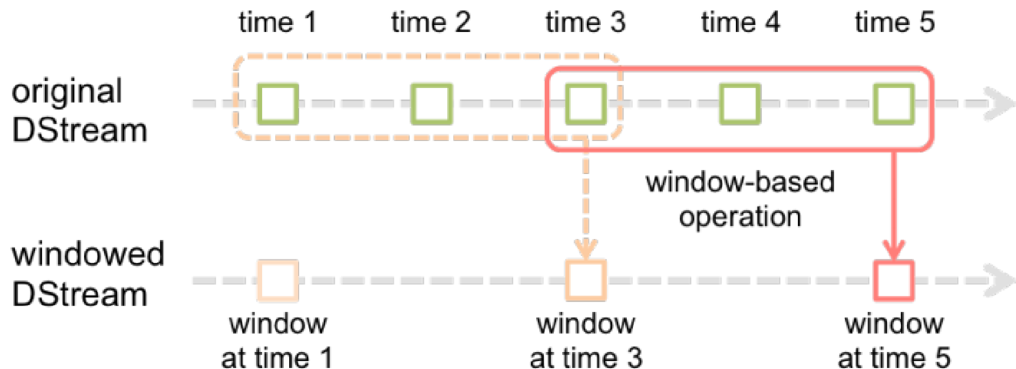
Example:

```
val dataset: RDD[String, String] = … // RDD
val wordCounts = … // Stream from first example
val joinedStream = wordCounts.transform {
      // transform
    rdd ⇒ rdd.join(dataset)
```

Window Operations

Windowed computations apply transformations over a sliding window of data

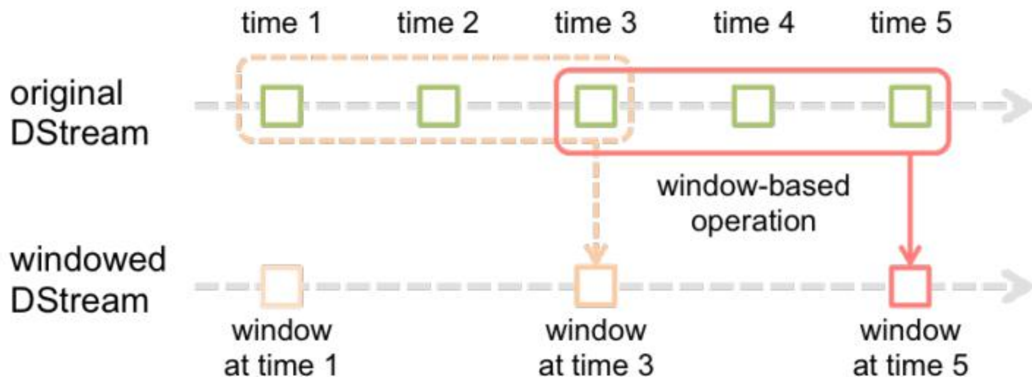
The window slides over a source DStream and combines the RDDs that fall within the window



Window Operations

Any window operation needs to specify two parameters:

- window length - The duration of the window (3 in the figure)
- sliding interval - The interval at which the window operation is performed (2 in the figure)



Join Operations

In each batch interval the RDD generated by stream1 can be joined with the RDD generated by stream2

```
val s1: DStream[String, String] = … //

val s2: DStream[String, String] = … //

val jS1 = s1.join(s2)

val jS2 = s1.leftOuterJoin(s2)

val jS3 = s1.rightOuterJoin(s2)

val jS4 = s1.fullOuterJoin(s2)

// The same with windowed streams
val wS1 = s1.window(Seconds(20))
val wS2 = s2.window(Minutes(1))
val wJS = wS1.join(wS2)
```

Output operations

Operation	Description
`print()`	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging.
`saveAsTextFiles(prefix, [suffix])`	Save this DStream's contents as text files. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".
`saveAsObjectFiles(prefix, [suffix])`	Save this DStream's contents as SequenceFiles of serialized Java objects. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".
`saveAsHadoopFiles(prefix, [suffix])`	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".
`foreachRDD(func)`	The most generic output operator that applies a function, func, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database.

Core Functionality Example

DStream - Example

```
// Create a DStream that will connect to a server
// listening on a TCP socket, say <IP>:9990
val ssc = new StreamingContext(conf, Seconds(5))
val lines = ssc.socketTextStream("<Some_IP>", 9990)

// Word count again
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word.trim, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()

// Start the computation
ssc.start()
// Wait for the application to terminate
ssc.awaitTermination()
// ssc.stop() forces application to stop
```

UpdateStateByKey Example

```
val ssc = new StreamingContext(conf, Seconds(5))
// Setting checkpoint directory in HDFS
ssc.checkpoint("hdfs://139.91.183.88:9000/checkpointDir")
// apply it on a DStream containing pairs(word, 1)
def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int]
   = {
       // add the new values to the previous running count to get the new count
       val newCount = newValues.sum + runningCount.getOrElse(0)
       Some(newCount)
}
val lines = ssc.socketTextStream("139.91.183.88", 9990)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word.trim, 1))
val runningCounts = pairs.updateStateByKey[Int](updateFunction _)

runningCounts.print()

ssc.start()                // Start the computation
ssc.awaitTermination()     // Wait for the computation to finish
```

UpdateStateByKey Example

1. this is spark tutorial

```
-----------------------------------------
Time: t1
-----------------------------------------
(this,1)
(is,1)
(tutorial,1)
(spark,1)
```

2. spark is fast

```
-----------------------------------------
Time: t2
-----------------------------------------
(this,1)
(is,2)
(fast,1)
(tutorial,1)
(spark,2)
```

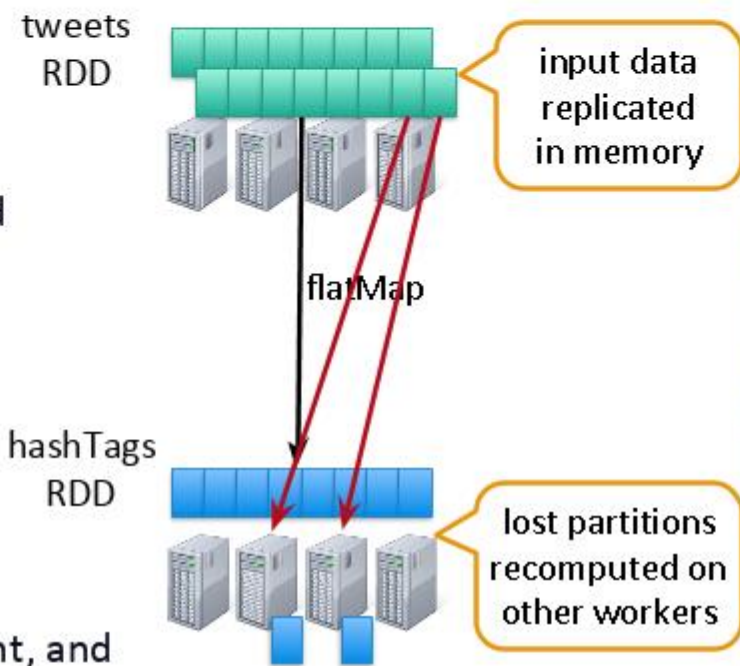
3. apache spark

```
-----------------------------------------
Time: t3
-----------------------------------------
(this,1)
(is,2)
(fast,1)
(apache,1)
(tutorial,1)
(spark,3)
```

Performance of Spark Streaming

Fault-tolerance: Worker

- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- All transformed data is fault-tolerant, and exactly-once transformations

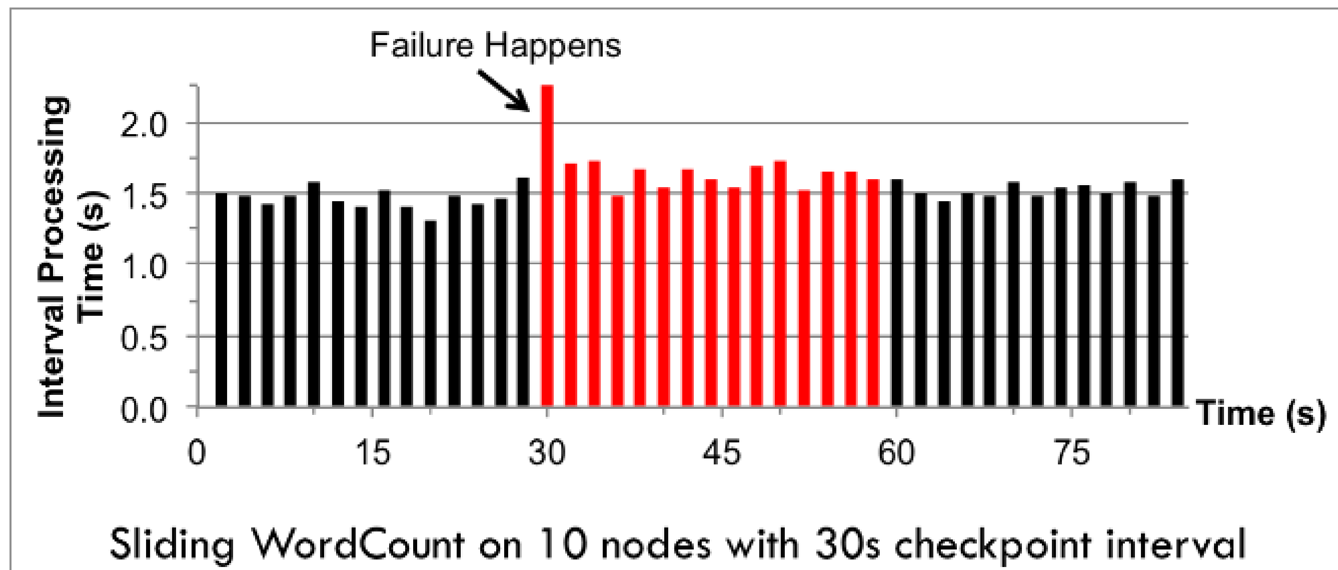


Fault-tolerance: Master

- Master saves the state of the DStreams to a checkpoint file
  - Checkpoint file saved to HDFS periodically
- If master fails, it can be restarted using the checkpoint file
- More information in the Spark Streaming guide
  - Link later in the presentation
- Automated master fault recovery coming soon

Fast Fault Recovery

Recovers from faults/stragglers within **1 sec**

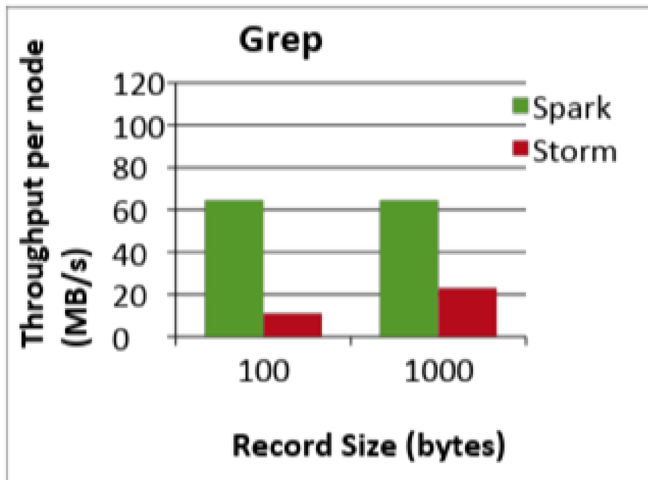


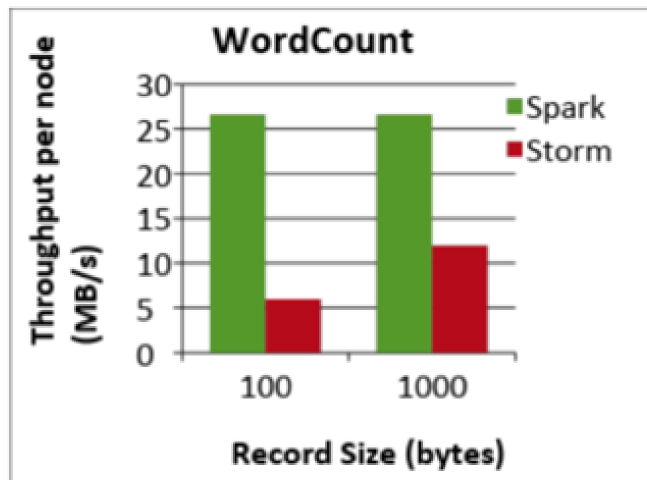
Sliding WordCount on 10 nodes with 30s checkpoint interval

Storm vs Spark

Higher throughput than Storm

- Spark Streaming: **670k** records/second/node
- Storm: **115k** records/second/node
- Apache S4: 7.5k records/second/node





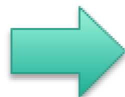
Real World Application Example (Twitter)

Example – Get hashtags from Twitter

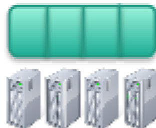
```
val tweets = ssc.twitterStream()
```

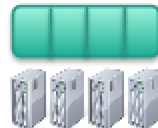
DStream: a sequence of RDDs representing a stream of data

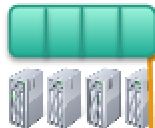
Twitter Streaming API batch @ t batch @ t+1 batch @ t+2



tweets DStream







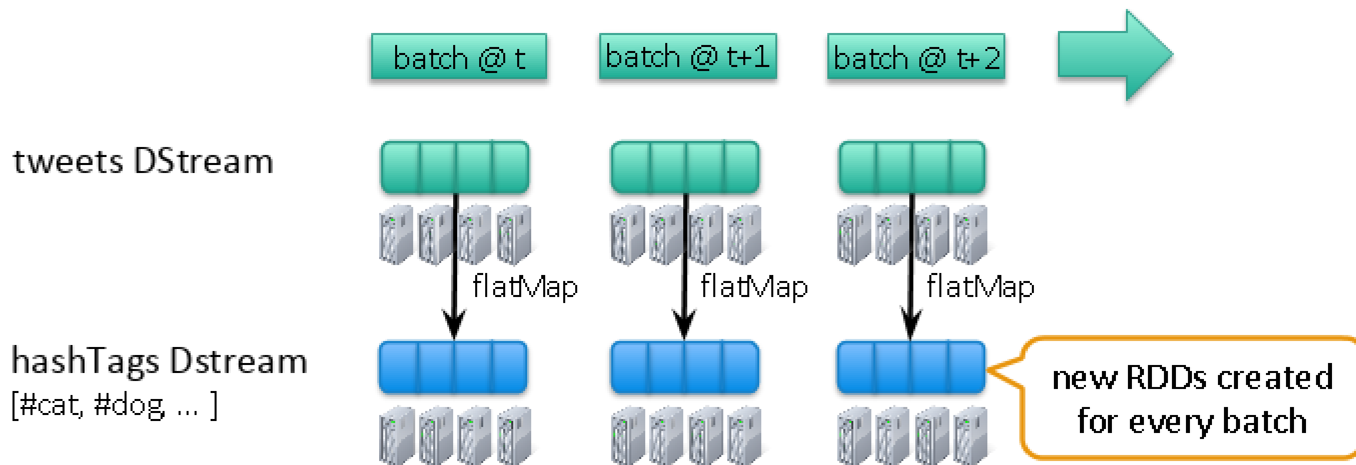
stored in memory as an RDD (immutable, distributed)

Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

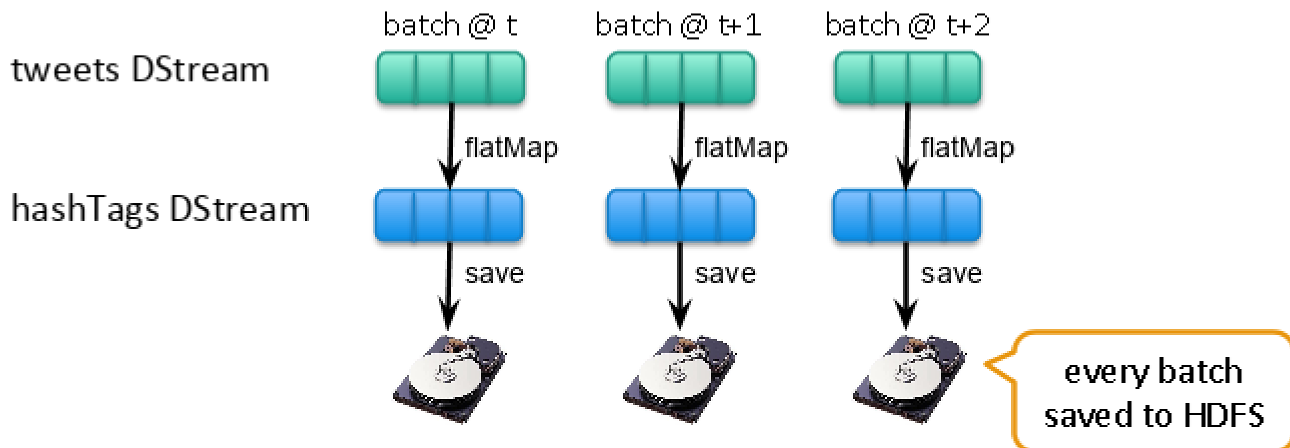
transformation: modify data in one DStream to create another DStream



Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

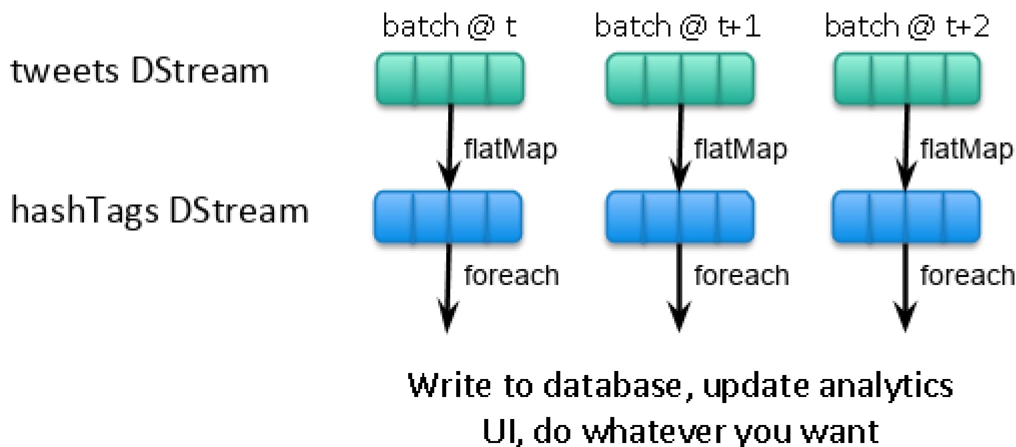
output operation: to push data to external storage



Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.foreach(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



Write to database, update analytics UI, do whatever you want

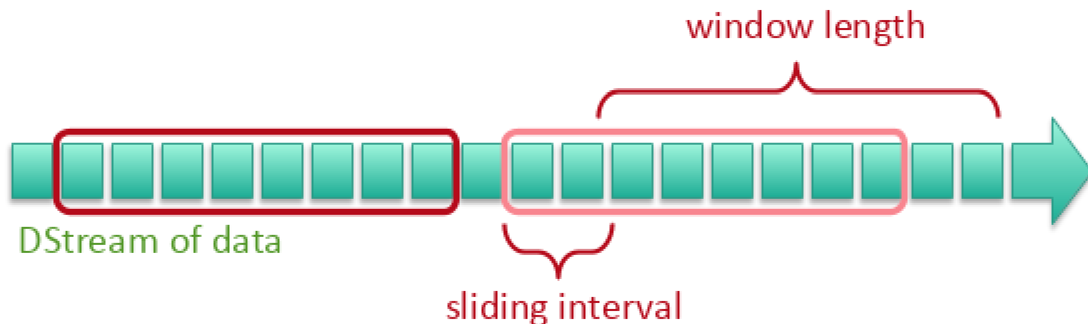
Window-based Transformations

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
```

sliding window operation

window length

sliding interval



Hall of Fame

Spark Streaming Tathagata Das (TD)

https://stanford.edu/~rezab/sparkclass/slides/td_streaming.pdf

Big Data Stream Processing - Berlin Big Data Center Tilmann Rabl

www.bbdc.berlin/fileadmin/news/photos/BD.../StreamProcessing-TilmannRabl.pdf

Spark Streaming: Large-scale near-real-time stream processing Tathagata Das (TD)

http://ampcamp.berkeley.edu/wp-content/uploads/2013/07/Spark-Streaming-AMPCamp-3.pptx

Spark Streaming Programming Guide

http://spark.apache.org/docs/latest/streaming-programming-guide.html