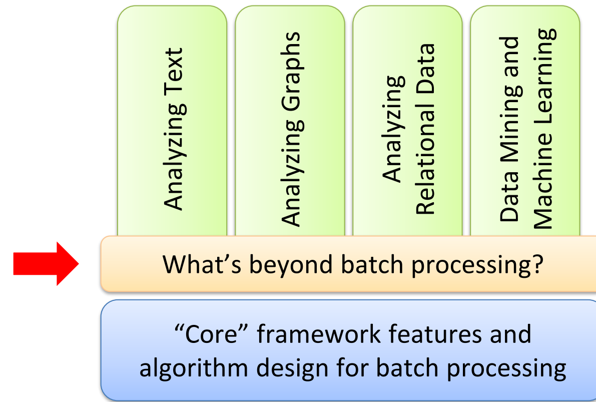
Structure of the Course



Stream Processing vs. Batch Processing

Batch processing	Stream processing
All the data	Continuously incoming data
Not real time	Latency critical (near real time)

Use Cases Across Industries

Credit

Identify fraudulent transactions as soon as they occur.

Transportation

Dynamic Re-routing Of traffic or Vehicle Fleet.

Retail

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

Consumer Internet & Mobile

Optimize user engagement based on user's current behavior.

Healthcare

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

Manufacturing

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

Surveillance

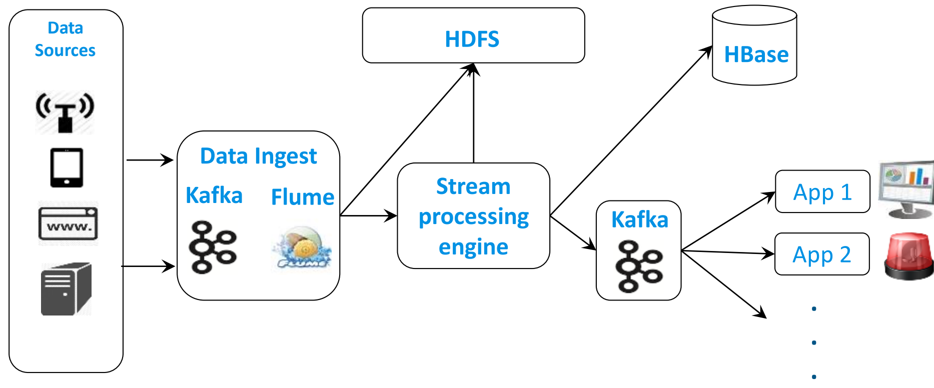
Identify threats and intrusions In real-time

Digital Advertising & Marketing

Optimize and personalize content based on real-time information.

4

Canonical Stream Processing Architecture



5

What is a data stream?

Sequence of items:

Structured (e.g., tuples)

Ordered (implicitly or timestamped)

Arriving continuously at high volumes

Sometimes not possible to store entirely

Sometimes not possible to even examine all items

What exactly do you do?

“Standard” relational operations:

Select
Project
Transform (i.e., apply custom UDF)
Group by
Join
Aggregations

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

Issues of Semantics

Group by… aggregate

When do you stop grouping and start aggregating?

Joining a stream and a static source

Simple lookup

Joining two streams

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

Joining two streams, group by and aggregation

When do you stop joining?

What's the solution?

Windows

Windows restrict processing scope:

Windows based on ordering attributes (e.g., time)

Windows based on item (record) counts

Windows based on explicit markers (e.g., punctuations)

Windows on Ordering Attributes

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

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

10

Windows on Counts

Window of size N elements (sliding, tumbling) over the stream

Windows from "Punctuations"

Application-inserted "end-of-processing"

Example: stream of actions… "end of user session"

Properties

Advantage: application-controlled semantics

Disadvantage: unpredictable window size (too large or too small)

Streams Processing Challenges

Inherent challenges

Latency requirements

Space bounds

System challenges

Bursty behavior and load balancing

Out-of-order message delivery and non-determinism

Consistency semantics (at most once, exactly once, at least once)

13

Producer/Consumers

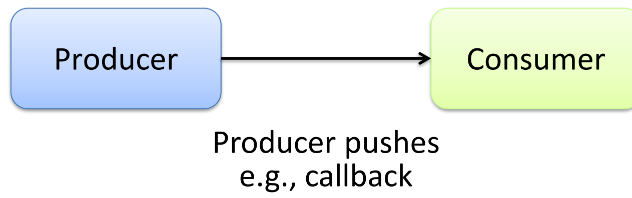
Producer

Consumer

How do consumers get data from producers?

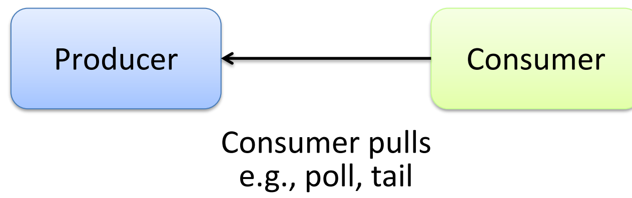
14

Producer/Consumers

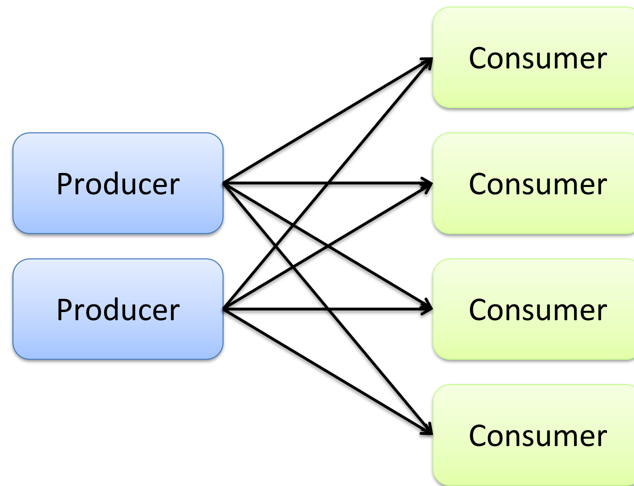


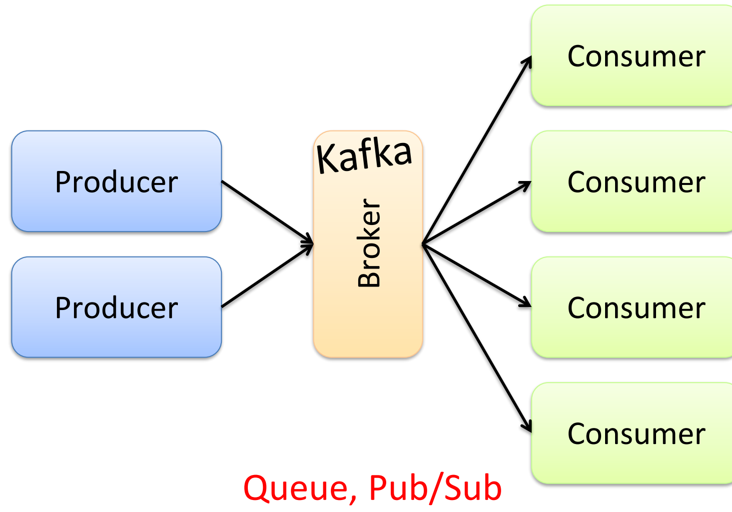
Producer pushes
e.g., callback

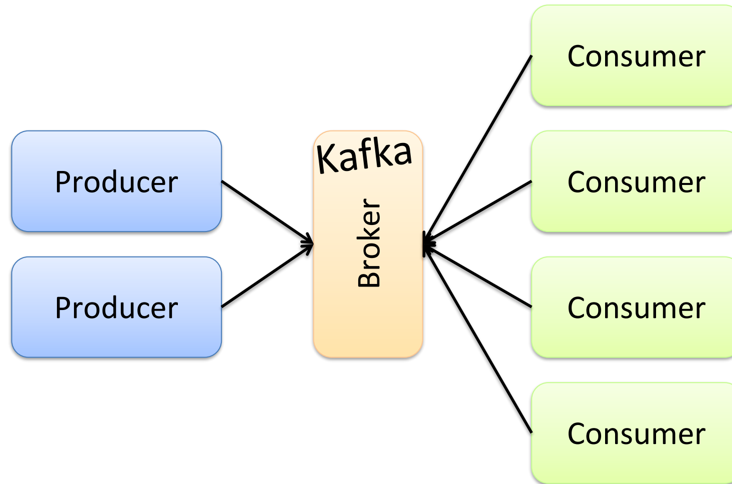
Producer/Consumers

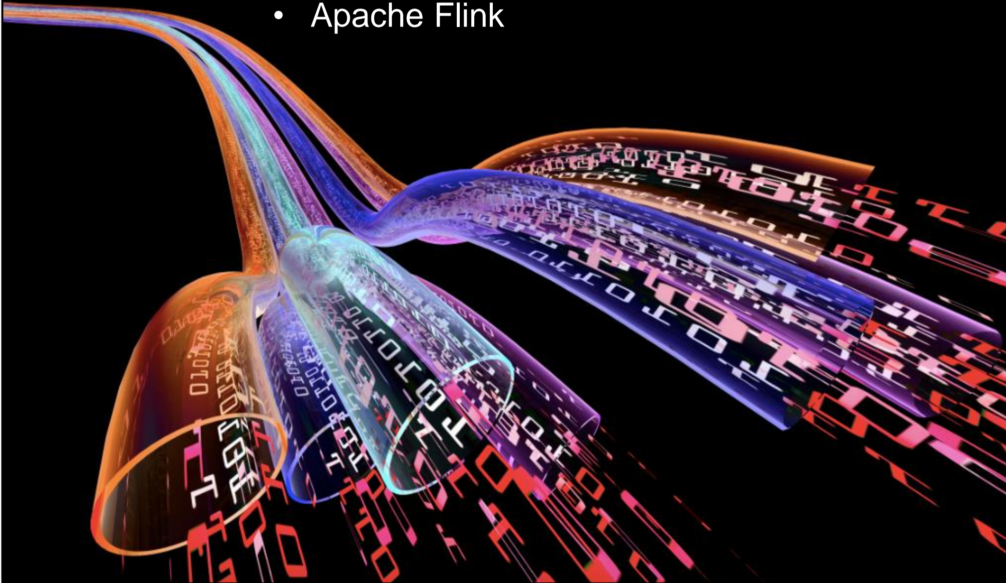


Consumer pulls
e.g., poll, tail


Producer/Consumers
Producer
Producer
Consumer
Consumer
Consumer
Consumer
17


Producer/Consumers
Producer
Producer
Kafka
Broker
Consumer
Consumer
Consumer
Consumer
Queue, Pub/Sub
18


Producer/Consumers
Producer
Producer
Kafka
Broker
Consumer
Consumer
Consumer
Consumer
19


Stream Processing Frameworks
• Apache Spark Streaming
• Apache Storm
• Apache Flink

Spark
Streaming

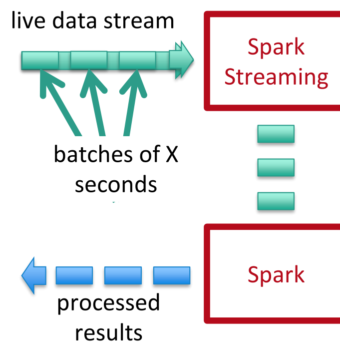
Spark Streaming: Discretized Streams

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

Chop up the stream into batches of *X* seconds

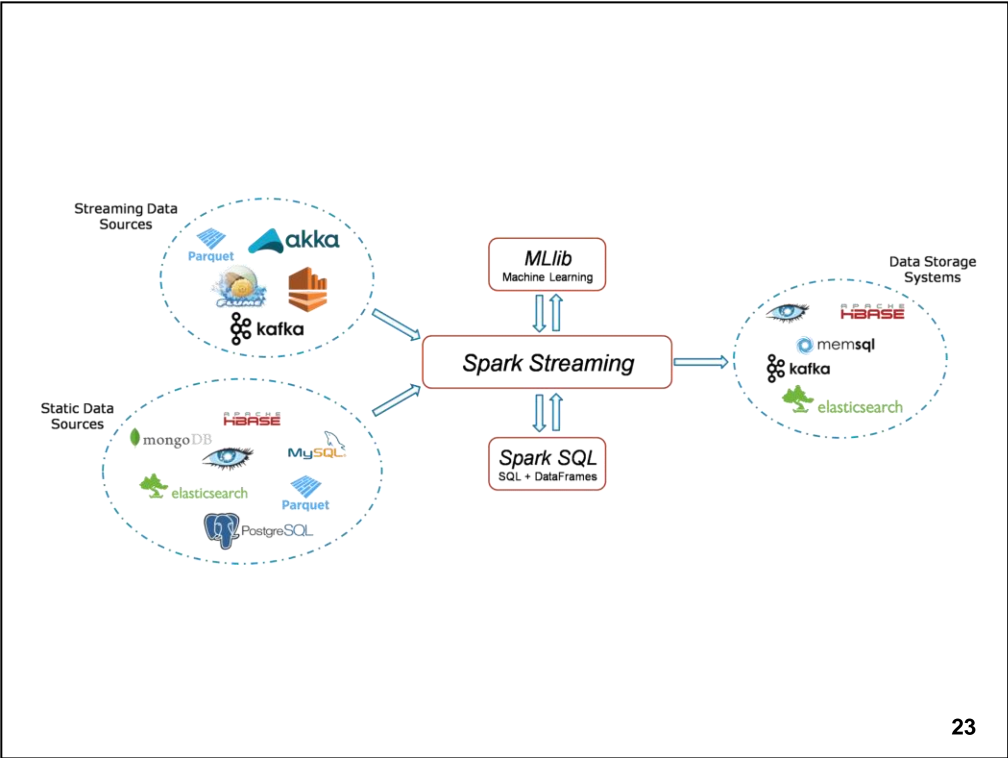
Process as RDDs!

Return results in batches



Typical batch window ~1s

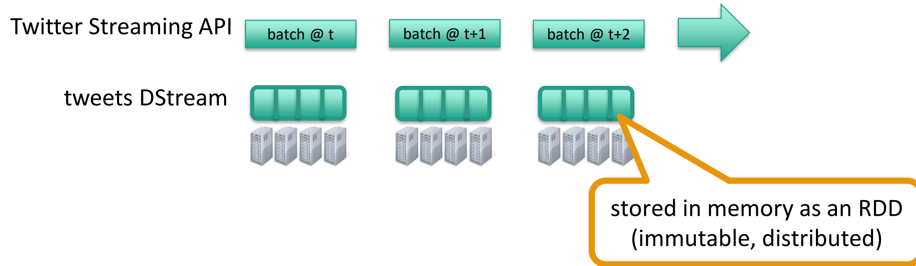
Source: All following Spark Streaming slides by Tathagata Das


Streaming Data Sources
Parquet
akka
kafka
Static Data Sources
HBASE
mongoDB
MySQL
elasticsearch
Parquet
PostgreSQL
MLlib
Machine Learning
Spark Streaming
Spark SQL
SQL + DataFrames
Data Storage Systems
HBASE
memsql
kafka
elasticsearch
23

Example: Get hashtags from Twitter

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

DStream: a sequence of RDD representing a stream of data

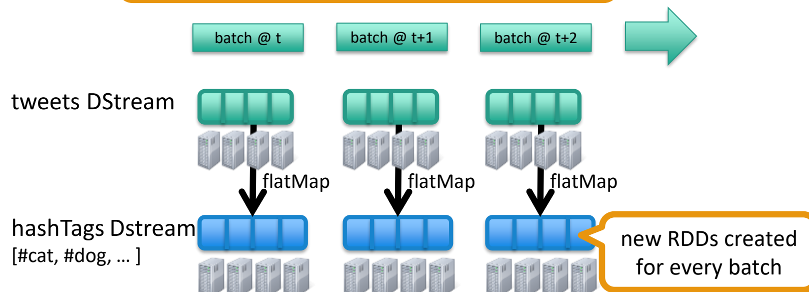


Example: Get hashtags from Twitter

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

new DStream

transformation: modify data in one
Dstream to create another DStream

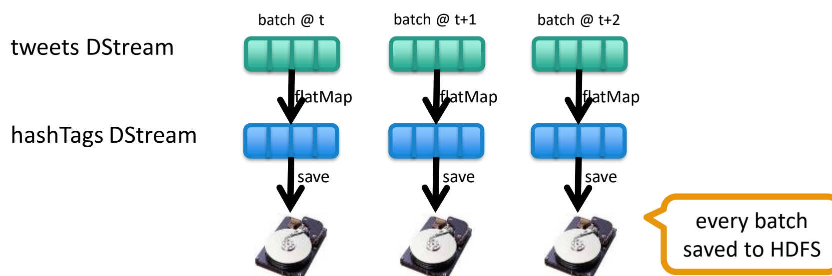


25

Example: Get hashtags from Twitter

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

output operation: to push data to external storage

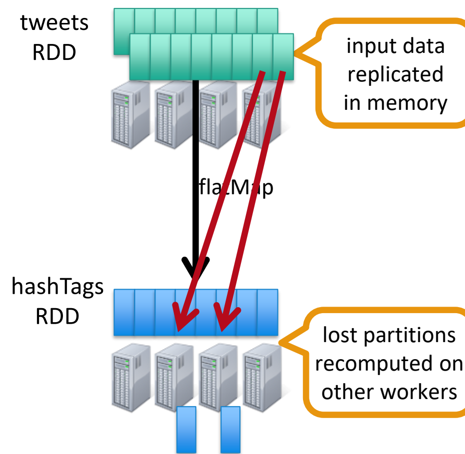


Fault Tolerance

Bottom line: they're just RDDs!

Fault Tolerance

Bottom line: they're just RDDs!



28

Key Concepts

DStream – sequence of RDDs representing a stream of data

Twitter, HDFS, Kafka, Flume, TCP sockets

Transformations – modify data from on DStream to another

Standard RDD operations – map, countByValue, reduce, join, …

Stateful operations – window, countByValueAndWindow, …

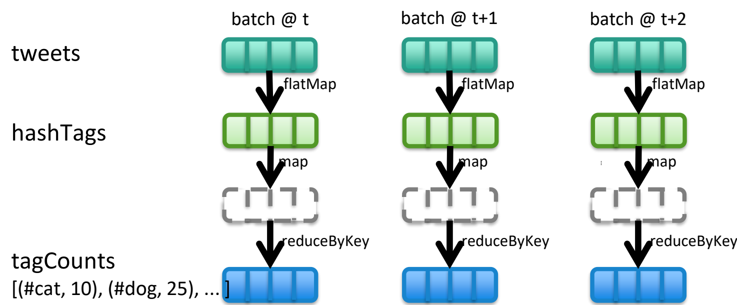
Output Operations – send data to external entity

saveAsHadoopFiles – saves to HDFS

foreach – do anything with each batch of results

Example: Count the hashtags

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

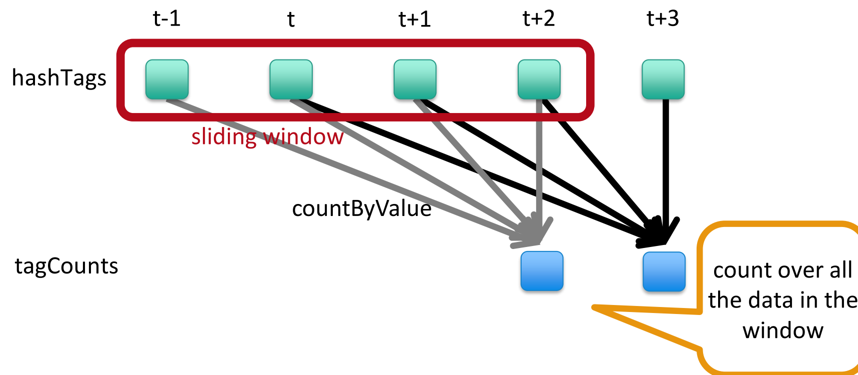


Example: Count the hashtags over last 10 mins

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

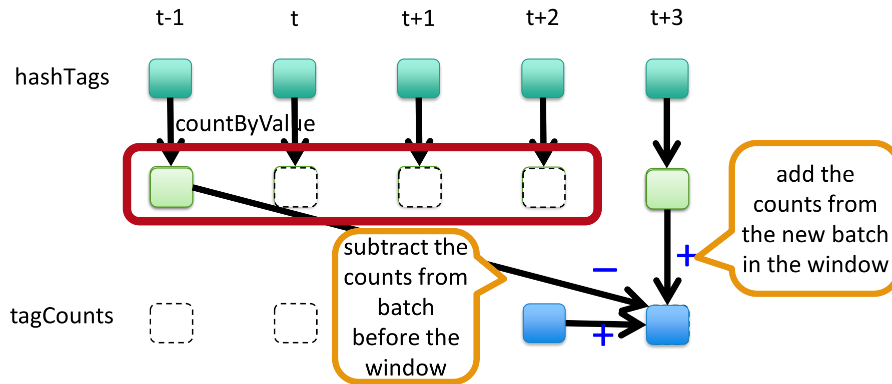
Example: Count the hashtags over last 10 mins

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



Smart window-based countByValue

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



Smart window-based reduce

Incremental counting generalizes to many reduce operations

Need a function to "inverse reduce" ("subtract" for counting)

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

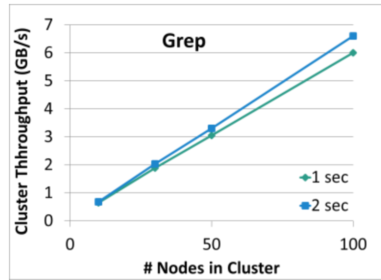
```
val tagCounts = hashtags
    .reduceByKeyAndWindow(_ + _, _ - _, Minutes(10), Seconds(1))
```

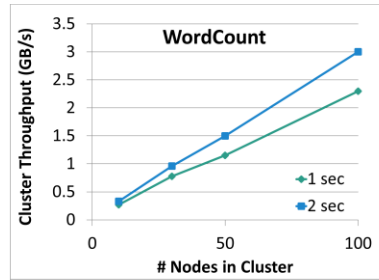
```
tagCounts = hashtags
    .reduceByKeyAndWindow(lambda x,y:x+y, lambda x,y:x-y,
Minutes(10), Seconds(1))
```

Performance

Can process **6 GB/sec (60M records/sec**) of data on 100 nodes at **sub-second** latencyTested

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





Comparison with Storm

Higher throughput than Storm

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