

Spark Structured Streaming

Why Spark Streaming?

```
graph LR; A[Raw Data Streams] --> B[Distributed Stream Processing System]; B --> C[Processed Data]
```

The diagram illustrates the flow of data through a distributed stream processing system. It starts with a green arrow labeled "Raw Data Streams" pointing to a pink rounded rectangle labeled "Distributed Stream Processing System". An arrow exits the right side of the pink box, labeled "Processed Data".

Scales to hundreds of nodes

Achieves low latency

A photograph of a man standing behind a podium, speaking into a microphone. The podium has a "Spark Summit" logo on it. The background is a blue screen with some text or graphics.

Why Spark Streaming?

```
graph LR; A[Raw Data Streams] --> B[Distributed Stream Processing System]; B --> C[Processed Data]
```

The diagram illustrates the flow of data through a distributed stream processing system. It starts with a green arrow labeled "Raw Data Streams" pointing to a pink rounded rectangle labeled "Distributed Stream Processing System". An arrow exits the right side of the pink box, labeled "Processed Data".

Scales to hundreds of nodes

Achieves low latency

Efficiently recover from failures

Integrates with batch and interactive pro

A photograph of a man standing behind a podium, speaking into a microphone. The podium has a "Spark Summit" logo on it. The background is a blue screen with some text or graphics.

Why Spark Streaming?



Scales to hundreds of nodes

Achieves low latency

Efficiently recover from failures

Integrates with batch and interactive processing

 DATABRICKS

Integration with Batch Processing

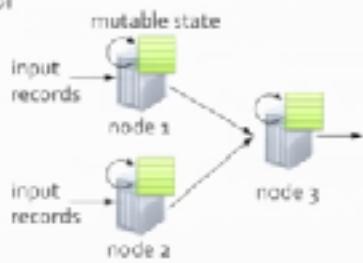
- Many environments require processing same data in live streaming as well as batch post-processing
- Existing frameworks cannot do both
 - Either, stream processing of 100s of MB/s with low latency
 - Or, batch processing of TBs of data with high latency



Fault-tolerant Stream Processing

- Traditional processing model

- Pipeline of nodes
 - Each node maintains mutable state
 - Each input record updates the state and new records are sent out



DATABRICKS

Existing Streaming Systems

- Storm

- Replays record if not processed by a node
 - Processes each record at *least* once
 - May update mutable state twice!
 - Mutable state can be lost due to failure!

- Trident – Use transactions to update state

- Processes each record *exactly* once
 - Per-state transaction to external database is slow

DATABRICKS

Spark Streaming

 DATABRICKS

Spark Streaming

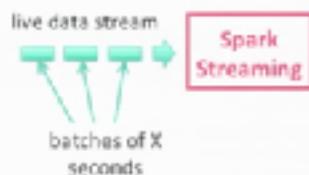
Run a streaming computation as a series of very small, deterministic batch jobs



Spark Streaming

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

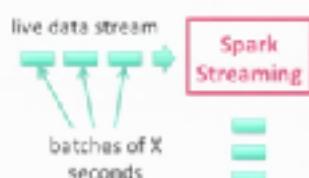
- Chop up the live stream into batches of X seconds



Spark Streaming

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

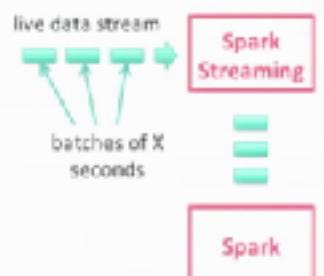
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations



Spark Streaming

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

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations

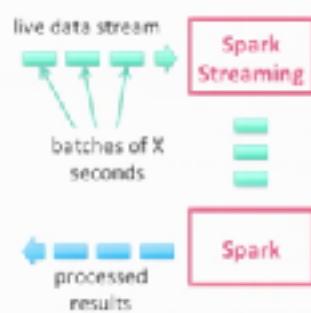


DATABRICKS

Spark Streaming

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

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches

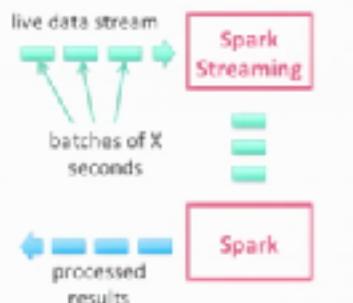


DATABRICKS

Spark Streaming

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

- Batch sizes as low as 1/2 sec, latency of about 1 sec
- Potential for combining batch processing and streaming processing in the same system



DATABRICKS

Programming Model - DStream

- Discretized Stream (DStream)
 - Represents a stream of data
 - Implemented as a sequence of RDDs



Programming Model - DStream

- Discretized Stream (DStream)
 - Represents a stream of data
 - Implemented as a sequence of RDDs
- DStreams can be either...
 - Created from streaming input sources
 - Created by applying transformations on existing DStreams

 DATABRICKS

Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
```



Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
```

Input DStream

Twitter Streaming API

batch @ t

batch @ t+1

batch @ t+2



Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
```

Input DStream

Twitter Streaming API

batch @ t

batch @ t+1

batch @ t+2



tweets DStream

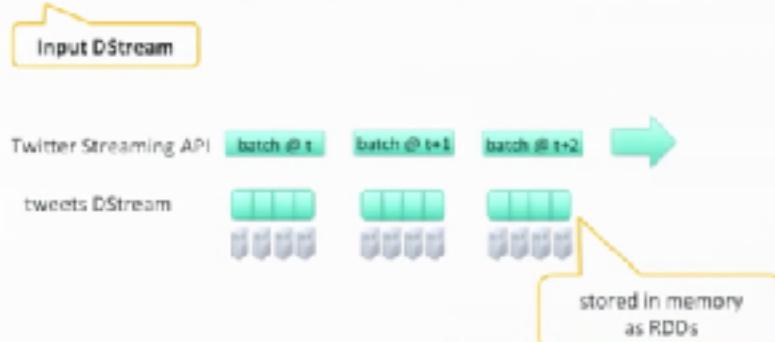


STO



Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(sparkContext, Seconds(1))  
val tweets = TwitterUtils.createStream(ssc, auth)
```



DATABRICKS

Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)  
val hashTags = tweets.flatMap(status => getTags(status))
```



DATABRICKS

Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)  
val hashTags = tweets.FlatMap(status => getTags(status))
```



DATABRICKS

Example – Get hashtags from Twitter

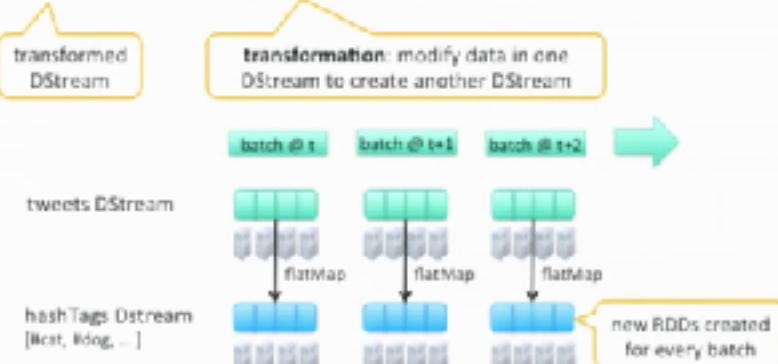
```
val tweets = TwitterUtils.createStream(ssc, None)  
val hashTags = tweets.FlatMap(status => getTags(status))
```



DATABRICKS

Example – Get hashtags from Twitter

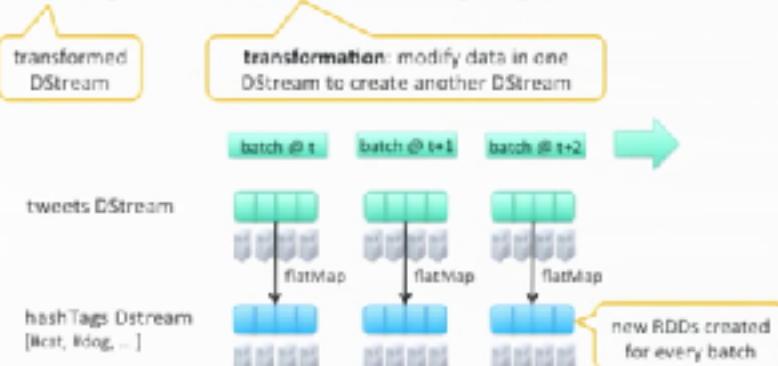
```
val tweets = TwitterUtils.createStream(ssc, None)  
val hashTags = tweets.flatMap(status => getTags(status))
```



DATABRICKS

Example – Get hashtags from Twitter

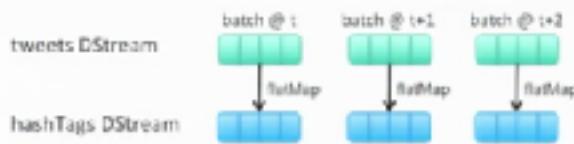
```
val tweets = TwitterUtils.createStream(ssc, None)  
val hashTags = tweets.flatMap(status => getTags(status))
```



DATABRICKS

Example – Get hashtags from Twitter

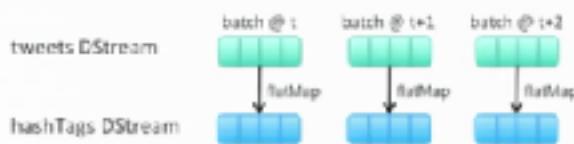
```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```



Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

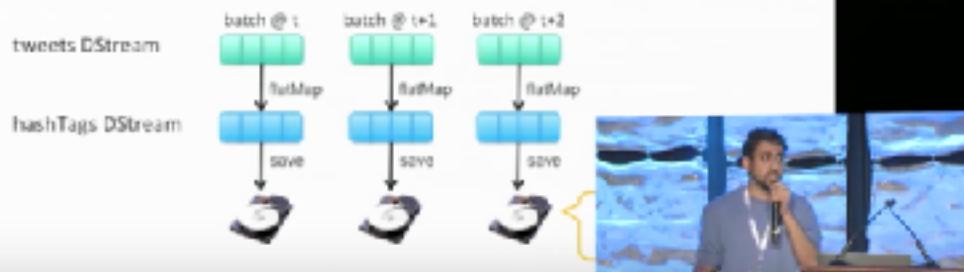
output operation: to push data to external storage



Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

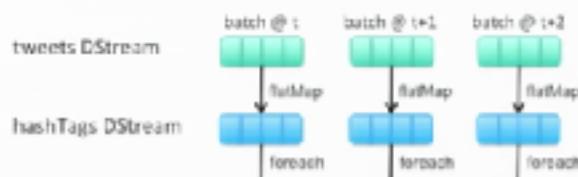
output operation: to push data to external storage



|| ▶ 12.23 / 29:52

Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreachRDD(hashTagRDD => { ... })
```



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

DATABRICKS

Languages

Scala API

```
val tweets = TwitterUtils.createStream(ssc, auth)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Java API

```
JavaDStream<Status> tweets = ssc.twitterStream()
JavaDStream<String> hashTags = tweets.flatMap(new Function<Status, String>() {
    hashTags.saveAsHadoopFiles("hdfs://...")
```

Python API

... soon



Languages

Scala API

```
val tweets = TwitterUtils.createStream(ssc, auth)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Java API

```
JavaDStream<Status> tweets = ssc.twitterStream()
JavaDStream<String> hashTags = tweets.flatMap(new Function<Status, String>() {
    hashTags.saveAsHadoopFiles("hdfs://...")
```

Function object

Python API

... soon



Window-based Transformations

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

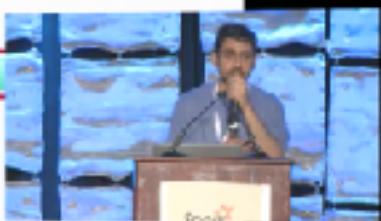


Window-based Transformations

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

sliding window
operation

DStream of data



Window-based Transformations

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



DATABRICKS

Window-based Transformations

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



DATABRICKS

Window-based Transformations

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



DATABRICKS

Arbitrary Stateful Computations

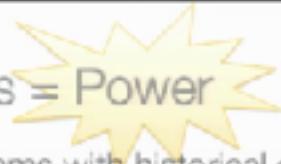
Specify function to generate new state based on previous state and new data

- Example: Maintain per-user mood as state, and update it with their tweets

```
def updateMood(newTweets, lastMood) => newMood  
  
val moods = tweetsByUser.updateStateByKey(updateMood _)
```

DATABRICKS

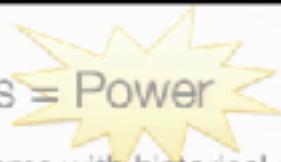
DStreams + RDDs = Power



- Combine live data streams with historical data
 - Generate historical data models with Spark, etc.
 - Use data models to process live data stream



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



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
- Query streaming data using SQL
 - select * from table_from_streaming_data



Databricks Keynote Demo

```
val ssc = new StreamingContext(sc, Seconds(5))
val sqlContext = new SQLContext(sc)
val tweets = TwitterUtils.createStream(ssc, auth)
val transformed = tweets.filter(isEnglish).window(Minutes(1))
```



Databricks Keynote Demo

```
val ssc = new StreamingContext(sc, Seconds(5))
val sqlContext = new SQLContext(sc)
val tweets = TwitterUtils.createStream(ssc, auth)
val transformed = tweets.filter(isEnglish).window(Minutes(1))

transformed.foreachRDD { rdd =>
    // Tweet is a case class containing necessary
    rdd.map(Tweet.apply(_)).registerAsTable("tweets")
}
```

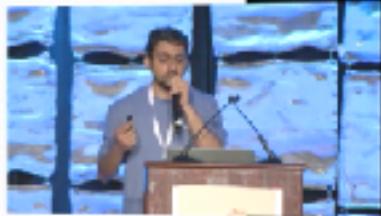
```
SELECT text FROM tweets WHERE similarity(tweet) > 0.61
SELECT getClosestCountry(lat, long) FROM tweets
```

 DATABRICKS

Advantage of an Unified Stack

- Explore data interactively to identify problems

```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = filtered.map(...)
```

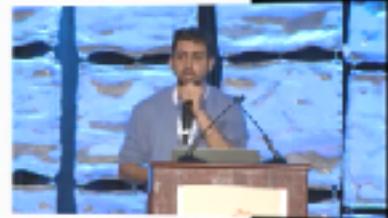


Advantage of an Unified Stack

- Explore data interactively to identify problems
- Use same code in Spark for processing large logs

```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = filtered.map(...)

object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
  }
}
```



Advantage of an Unified Stack

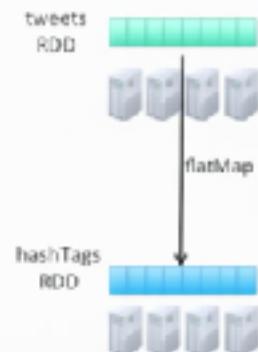
- Explore data interactively to identify problems
- Use same code in Spark for processing large logs
- Use similar code in Spark Streaming for realtime processing

```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = filtered.map(...)

object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
  }
}

object ProcessLiveStream {
  def main(args: Array[String]) {
    val ssc = new StreamingContext(...)
    val stream = KafkaUtil.createStream(...)
    val filtered = stream.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
  }
}
```

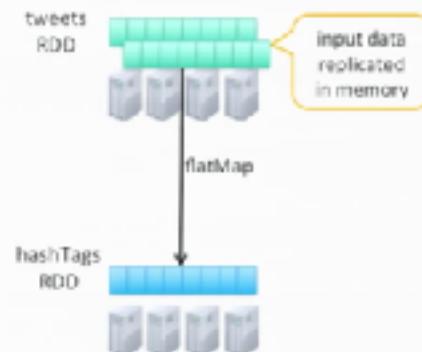
Fault-tolerance



DATABRICKS

Fault-tolerance

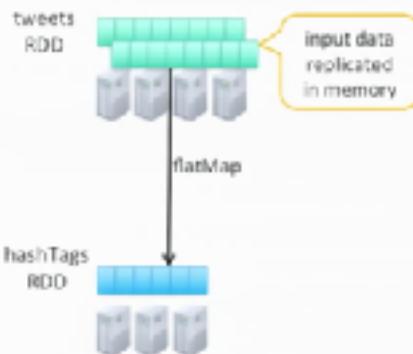
- Batches of input data are replicated in memory for fault-tolerance



DATABRICKS

Fault-tolerance

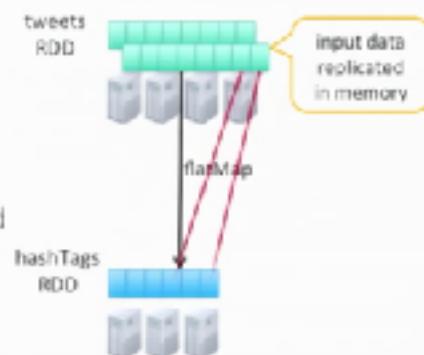
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data



DATABRICKS

Fault-tolerance

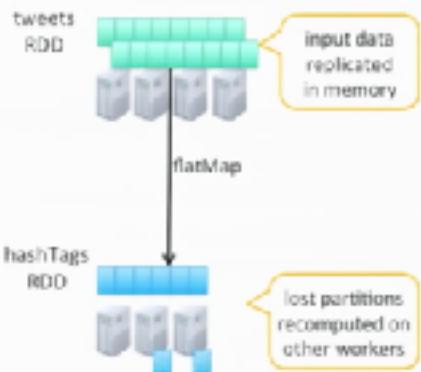
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data



DATABRICKS

Fault-tolerance

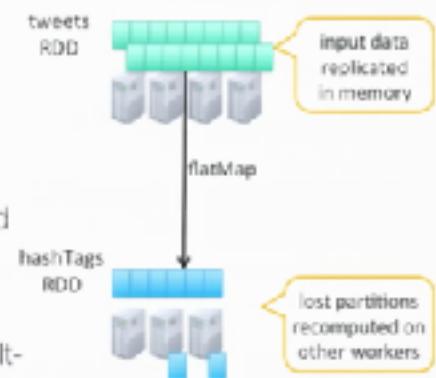
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data



DATABRICKS

Fault-tolerance

- 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 transformations are fault-tolerant, and exactly-once transformations



DATABRICKS

Input Sources

- Out of the box, we provide
 - Kafka, Flume, Akka Actors, Raw TCP sockets, HDFS, etc.
- Very easy to write a custom receiver
 - Define what to when receiver is started and stopped
- Also, generate your own sequence of RDDs, etc.
and push them in as a “stream”

 DATABRICKS

Future Directions!

- Better integration with SQL, MLlib, GraphX
- Improved integration with different sources
- Improved control over data rates
- Python API

 DATABRICKS

The screenshot shows the homepage of the Spark Streaming Programming Guide. The title "Conclusion" is prominently displayed at the top. Below it is a navigation bar with links to "Documentation", "Programming Guide", "API Docs", "Deploying", and "Home". The main content area is titled "Spark Streaming Programming Guide" and contains a table of contents with several sections: Overview, Getting Started, API Overview, Data Sources, Data Processing, Windowed Operations, Performance Tuning, and Examples. Under "Examples", there are links to "WordCount", "NetworkWordCount", "DStream API", "Java DStream API", "Java API Examples", and "Python Examples". A detailed description of Spark Streaming follows, mentioning its use in real-time processing of data streams from various sources like Kafka, Flume, and Twitter, and its output to HDFS, Redis, and MongoDB. At the bottom of the page is a diagram illustrating the data flow from sources to sinks, and the Databricks logo.

Spark Streaming Failures & Recovery

Executor /driver failures

<https://www.youtube.com/watch?v=d5UJorruHk>

Failures? Why care?

Many streaming applications need zero data loss guarantees despite any kind of failures in the system

- At least once guarantee - every record processed at least once
- Exactly once guarantee - every record processed exactly once

Different kinds of failures - **executor** and **driver**

Some failures and guarantee requirements need additional configurations and setups

• databricks

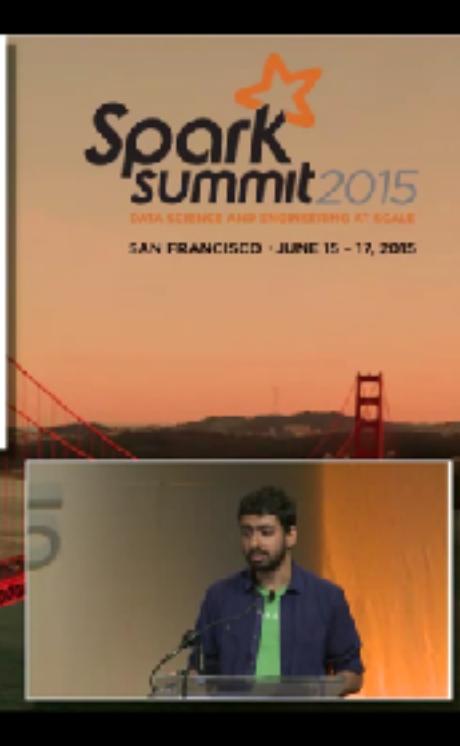
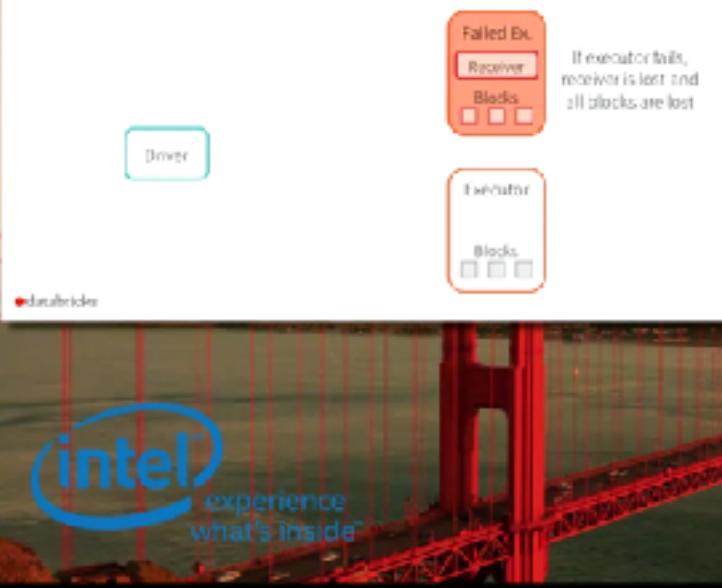


What if an executor fails?

• databricks

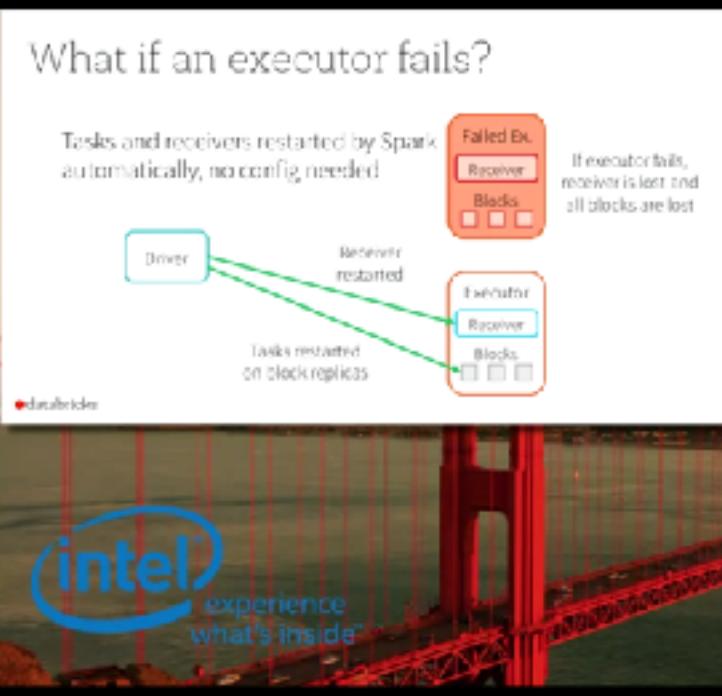


What if an executor fails?

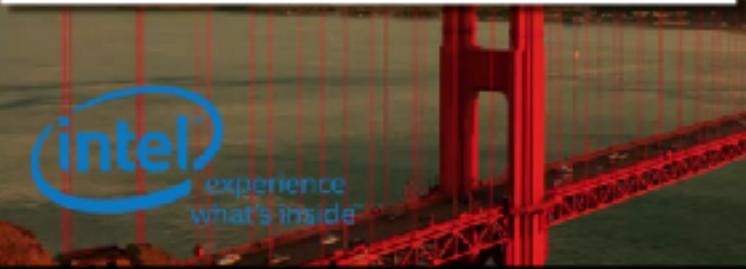


What if an executor fails?

Tasks and receivers restarted by Spark automatically, no config needed



What if the driver fails?



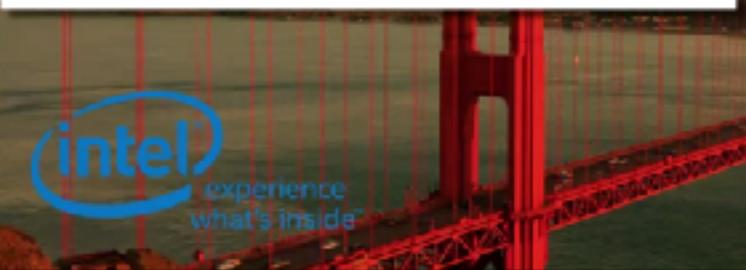
Spark
summit 2015

DATA SCIENCE AND ENGINEERING AT SCALE

SAN FRANCISCO • JUNE 15 - 17, 2015



What if the driver fails?



Spark
summit 2015

DATA SCIENCE AND ENGINEERING AT SCALE

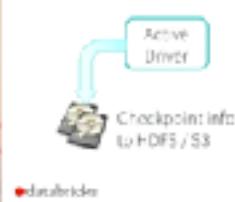
SAN FRANCISCO • JUNE 15 - 17, 2015



Recovering Driver with Checkpointing

DStream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage



SAN FRANCISCO · JUNE 15 - 17, 2015



Recovering Driver w/ DStream Checkpointing

DStream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage



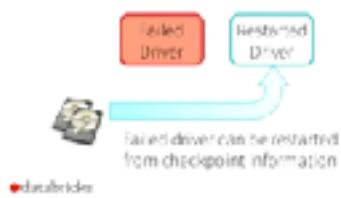
SAN FRANCISCO · JUNE 15 - 17, 2015



Recovering Driver w/ DStream Checkpointing

DStream Checkpointing

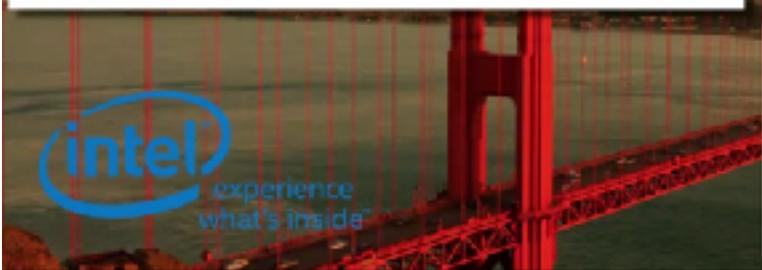
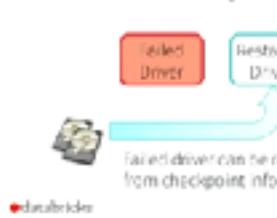
Periodically save the DWG of DStreams to fault-tolerant storage



Recovering Driver w/ DStream Checkpointing

05stream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage



Recovering Driver w/ DStream Checkpointing

1. Configure automatic driver restart:
All cluster managers support this
2. Set a checkpoint directory in a HDFS-compatible file system
`streamingContext.checkpoint(hdfsDirectory)`
3. Slightly restructure of the code to use checkpoints for recovery

•databricks

19



Configuring Automatic Driver Restart

Spark Standalone – Use spark-submit with “cluster” mode and “–supervise”

See <http://spark.apache.org/docs/latest/spark-standalone.html>

YARN – Use spark-submit in “cluster” mode

See YARN config “yarn.resourcemanager.am.max.attempts”

Mesos – Marathon can restart Mesos applications

•databricks



Restructuring code for Checkpointing

```
Create val context = new StreamingContext(...)  
+ val lines = KafkaUtils.createStream(...)  
val words = lines.flatMap(...)  
Setup ...
```

```
Start context.start()
```

edu@beider:



Restructuring code for Checkpointing

```
Create val context = new StreamingContext(...)  
+ val lines = KafkaUtils.createStream(...)  
val words = lines.flatMap(...)  
Setup ...
```

```
def createContext(): StreamingContext = {  
    val context = new StreamingContext(...)  
    val lines = KafkaUtils.createStream(...)  
    val words = lines.flatMap(...)  
    context.checkpoint(500L)  
}
```

Put all setup code into a function that returns a new StreamingContext

```
Start context.start()
```

edu@beider:



Restructuring code for Checkpointing

Create
+
Setup ...



```
def creatingFunc(): StreamingContext = {  
    val context = new StreamingContext(...)  
    val lines = KafkaUtils.createStream(...)  
    val words = lines.flatMap(...)  
    ...  
    context.checkpoint(hdfsDir)  
}
```

Put all setup code into a function that returns a new StreamingContext

Start context.start()

databricks

Restructuring code for Checkpointing

Create
+
Setup ...



```
def creatingFunc(): StreamingContext = {  
    val context = new StreamingContext(...)  
    val lines = KafkaUtils.createStream(...)  
    val words = lines.flatMap(...)  
    ...  
    context.checkpoint(hdfsDir)  
}
```

Put all setup code into a function that returns a new StreamingContext

Start context.start()



```
val context =  
StreamingContext.getOrCreate(  
    hdfsDir, creatingFunc)  
context.start()
```

Get context setup from HDFS dir OR create a new one with the function

databricks

Restructuring code for Checkpointing

StreamingContext.getOrCreate():

If HDFS directory has checkpoint info
recover context from info
else
call creatingFunc() to create
and setup a new context

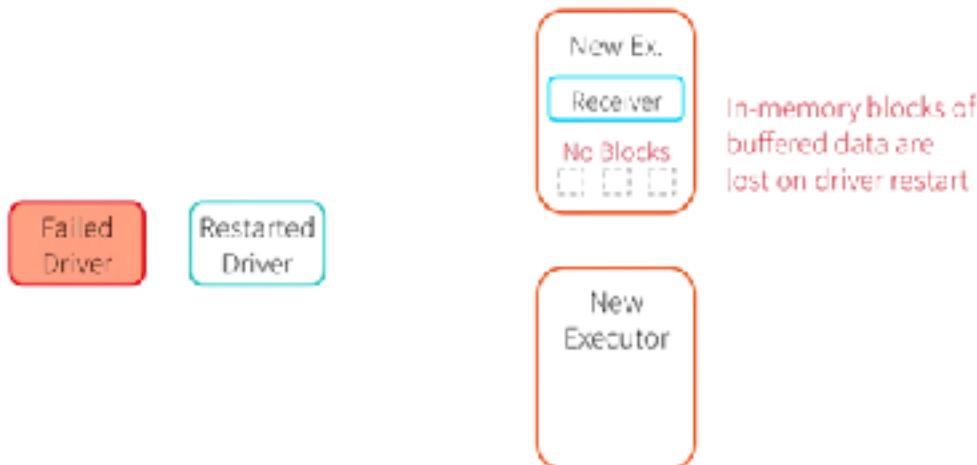
Restarted process can figure out whether
to recover using checkpoint info or not

```
def creatingFunc(): StreamingContext = {  
    val context = new StreamingContext(...)  
    val lines = KafkaUtils.createStream(...)  
    val words = lines.flatMap(...)  
    ...  
    context.checkpoint(hdfsDir)  
}
```

```
val context =  
StreamingContext.getOrCreate(  
    hdfsDir, creatingFunc)  
context.start()
```

#databricks[

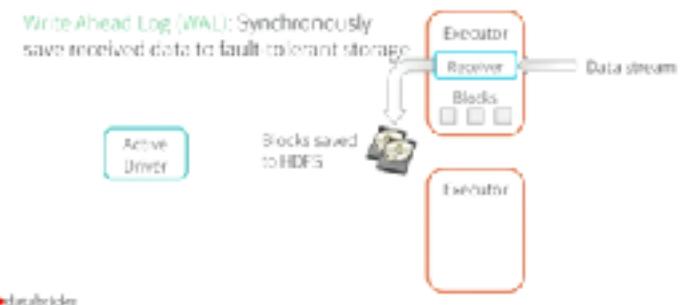
Received blocks lost on Restart!



#databricks[

Recovering data with Write Ahead Logs

Write-Ahead Log (WAL): Synchronously save received data to fault-tolerant storage.



→ danielbrides

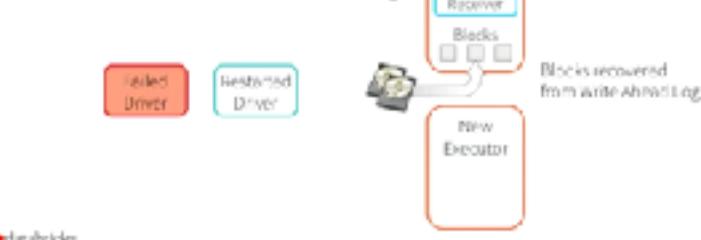


Spark
summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO • JUNE 15 - 17, 2015



Recovering data with Write Ahead Logs

Write-Ahead Log (WAL): Synchronously save received data to fault-tolerant storage.



→ danielbrides



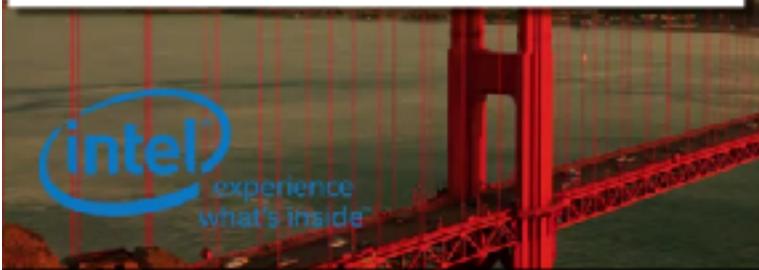
Spark
summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO • JUNE 15 - 17, 2015



Recovering data with Write Ahead Logs

1. Enable checkpointing, logs written in checkpoint directory
2. Enabled WAL in SparkConf configuration
`sparkConf.set("spark.streaming.receiver.writeAheadLog.enable", "true")`
3. Receiver should also be reliable
Acknowledge source only after data saved to WAL.
Unacked data will be relayed from source by restarted receiver
4. Disable in-memory replication (already replicated by HDFS)
`StorageLevel.MEMORY_AND_DISK_2` for input DStreams

• danielbehrer

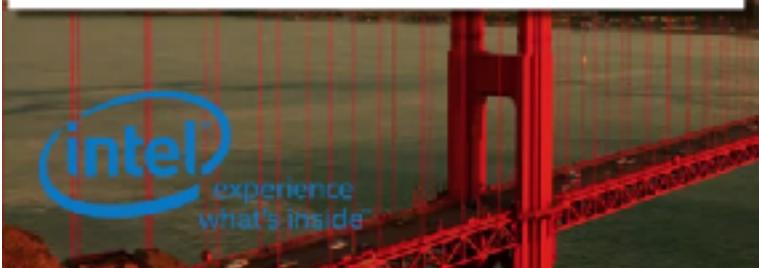


RDD Checkpointing

- Stateful stream processing can lead to long RDD lineages
- Long lineage = bad for fault tolerance, too much recomputation
- RDD checkpointing saves RDD data to the fault-tolerant storage to limit lineage and recomputation

More: <http://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing>

• danielbehrer



Fault-tolerance Semantics



Zero data loss = every stage processes each event **at least once** despite any failure



Spark
summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO • JUNE 15 - 17, 2015

Fault-tolerance Semantics

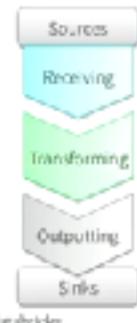


Exactly once, as long as received data is not lost



Spark
summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO • JUNE 15 - 17, 2015

Fault-tolerance Semantics



At least once, w/ Checkpointing + WAL + Reliable receivers

Exactly once, as long as received data is not lost



Fault-tolerance Semantics



At least once, w/ Checkpointing + WAL + Reliable receivers

Exactly once, as long as received data is not lost

Exactly once, if outputs are idempotent or transactional



Note:

Idempotent: not changed/ unchanged

Fault-tolerance Semantics



At least once, w/ Checkpointing + WAL + Reliable receivers

Exactly once, as long as received data is not lost

Exactly once, if outputs are idempotent or transactional

End-to-end semantics:
At-least once



DATA SCIENCE AND ENGINEERING AT SCALE

SAN FRANCISCO • JUNE 15 - 17, 2015



Fault-tolerance Semantics



Exactly once receiving with new **Kafka Direct** approach

Treats Kafka like a replicated log and reads it like a file

Does not use receivers

No need to create multiple DStreams and union them

No need to enable WriteAhead Logs

`val directFileStream = KafkaUtils.createDirectStream(...)`

<http://deepsolver.csail.mit.edu/paper/00000000000000000000000000000000.pdf>
The paper goes into more details of running Kafka integration test



DATA SCIENCE AND ENGINEERING AT SCALE

SAN FRANCISCO • JUNE 15 - 17, 2015



Fault-tolerance Semantics



Exactly once receiving with new Kafka Direct approach

Exactly once, as long as received data is not lost

Exactly once, if outputs are idempotent or transactional

End-to-end semantics:
Exactly once!



Spark
summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO • JUNE 15 - 17, 2015



NETFLIX

The Netflix Tech Blog

Wednesday, January 21, 2015

Can Spark Streaming survive Choke Monkey?

By Bharat Patel, Praveen Pattnaik, Nitish Agarwal, Nitin Pawar

Apache Flink is a distributed stream system that allows event data to be processed in an **exactly once** and, very fraudulently, in real time. Processing in the context of complex transformations and state management is often referred to as “stateful computation”. In this post, we will show how to use Flink’s stateful computation API to implement a “Choke Monkey” which handles watermarking problems of distributed windowing.

Topics on [Amazon Web Services \(AWS\)](#) is relevant to us as Flink processes out of the AWS cloud. When processing multiple input, it is important to know resources on AWS are ephemeral, temporal, durable and transient.



<http://techblog.netflix.com/2015/01/can-spark-streaming-survive-choke-monkey.html>

edu4life



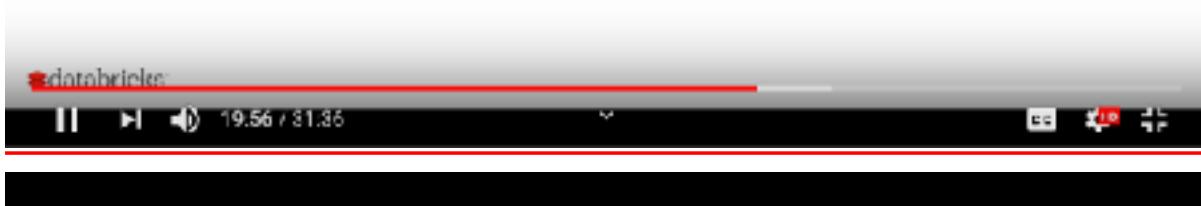
Spark
summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO • JUNE 15 - 17, 2015



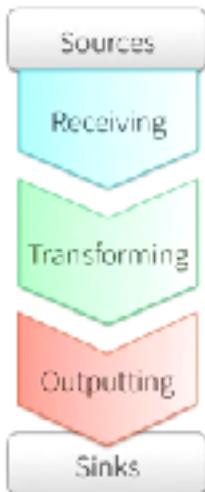
Fault-tolerance and Semantics

Performance and Stability

Monitoring and Upgrading



Achieving High Throughput

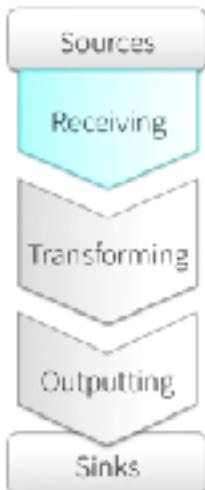


High throughput achieved by sufficient parallelism at all stages of the pipeline



Throughput: the amount of data passing through process(data passing/ data travelling)

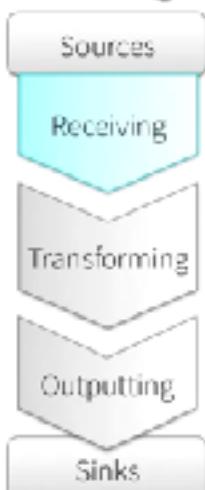
Scaling the Receivers



Sources must be configured with parallel data streams
#partitions in Kafka topics, #shards in Kinesis streams, ...

#databricks

Scaling the Receivers



Sources must be configured with parallel data streams
#partitions in Kafka topics, #shards in Kinesis streams, ...

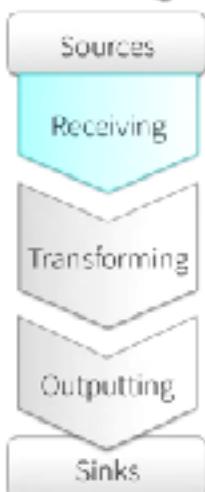
Streaming app should have multiple receivers that receive the data streams in parallel

Multiple input DStreams, each running a receiver
Can be unioned together to create one DStream

```
val kafkaStream1 = KafkaUtils.createStream(...)  
val kafkaStream2 = KafkaUtils.createStream(...)  
val unionedStream = kafkaStream1.union(kafkaStream2)
```

#databricks

Scaling the Receivers



• databricks

Sufficient number of executors to run all the receivers

Absolute necessity: #cores > #receivers

Good rule of thumb: #executors > #receivers, so that no more than 1 receiver per executor, and network is not shared between receivers

Scaling the Receivers



• databricks

Sufficient number of executors to run all the receivers

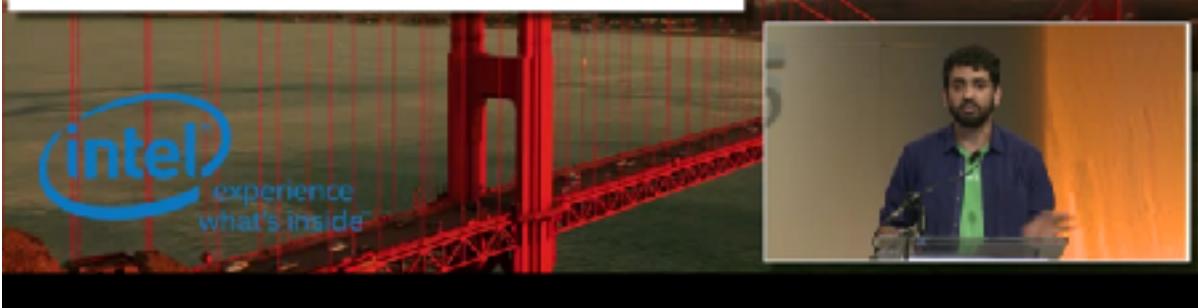
Absolute necessity: #cores > #receivers

Good rule of thumb: #executors > #receivers, so that no more than 1 receiver per executor, and network is not shared between receivers

Kafka Direct approach does not use receivers

Automatically parallelizes data reading across executors

Parallelism = # Kafka partitions



Stability in Processing



For stability, must process data as fast as it is received

Must ensure $\text{avg. batch processing times} < \text{batch interval}$
Previous batch is done by the time next batch is received



Stability in Processing



For stability, must process data as fast as it is received

Must ensure $\text{avg. batch processing times} < \text{batch interval}$
Previous batch is done by the time next batch is received

Otherwise, new batches keeps queuing up waiting for previous batches to finish, scheduling delay goes up



Reducing Batch Processing Times



- More receivers!
Executor running receivers do lot of the processing
- Repartition the received data to explicitly distribute load
`UnboundedStream.repartition(48)`



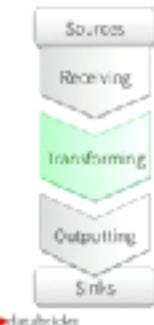
Spark
summit 2015

DATA SCIENCE AND ENGINEERING AT SCALE

SAN FRANCISCO • JUNE 15 - 17, 2015



Reducing Batch Processing Times



- More receivers!
Executor running receivers do lot of the processing
- Repartition the received data to explicitly distribute load
`UnboundedStream.repartition(48)`
- Set rpartitions in shuffles, make sure its large enough
`TransformedStream.reduceByKey(reduceFunc, 48)`
- Get more executors and cores!



Spark
summit 2015

DATA SCIENCE AND ENGINEERING AT SCALE

SAN FRANCISCO • JUNE 15 - 17, 2015



Reducing Batch Processing Times

Sources
Receiving
Transforming
Outputting
Sinks

edu@drdeeps

Use Kryo serialization to reduce serialization costs
Register classes for best performance
See configurations spark.kryo.*
<http://spark.apache.org/docs/latest/configuring-spark.html#kryo-configuration>

intel experience what's inside

Spark SUMMIT 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO • JUNE 15 - 17, 2015

San Francisco, CA

Golden Gate Bridge at sunset

Speaker at podium

Reducing Batch Processing Times

Sources
Receiving
Transforming
Outputting
Sinks

edu@drdeeps

Use Kryo serialization to reduce serialization costs
Register classes for best performance
See configurations spark.kryo.*
<http://spark.apache.org/docs/latest/configuring-spark.html#kryo-configuration>

Larger batch durations improve stability
More data aggregated together, amortized cost of shuffle

Limit ingestion rate to handle data surges
See configurations spark.streaming.messaging.*
<http://spark.apache.org/docs/latest/streaming-programming-guide.html#configuration>

intel experience what's inside

Spark SUMMIT 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO • JUNE 15 - 17, 2015

San Francisco, CA

Golden Gate Bridge at sunset

Speaker at podium

Amortized: reduce the cost (i.e gradually write off the initial cost of (an asset) over a period)

Data surge: data overflow/ a sudden powerful forward or upward movement

Speeding up Output Operations



Write to data stores efficiently

foreach: inefficient

- detail: foreach: 1 event →
 - ↳ open connection
 - ↳ insert single event
 - ↳ close connection



SAN FRANCISCO • JUNE 15 - 17, 2015



experience
what's inside™



Speeding up Output Operations



Write to data stores efficiently

foreach: inefficient

- detail: foreach: 1 event →
 - ↳ open connection
 - ↳ insert single event
 - ↳ close connection

foreachPartition: efficient

- detail: foreachPartition: 1 partition →
 - ↳ open connection
 - ↳ insert all events in partition
 - ↳ close connection



SAN FRANCISCO • JUNE 15 - 17, 2015



experience
what's inside™

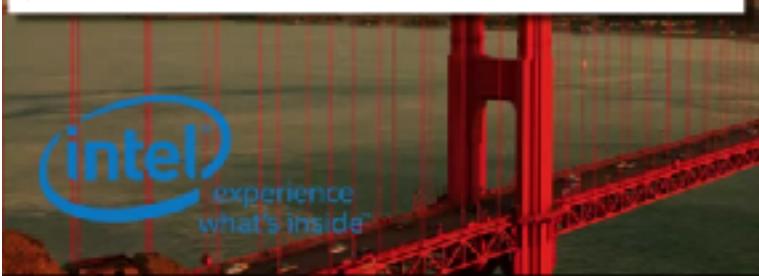


Speeding up Output Operations

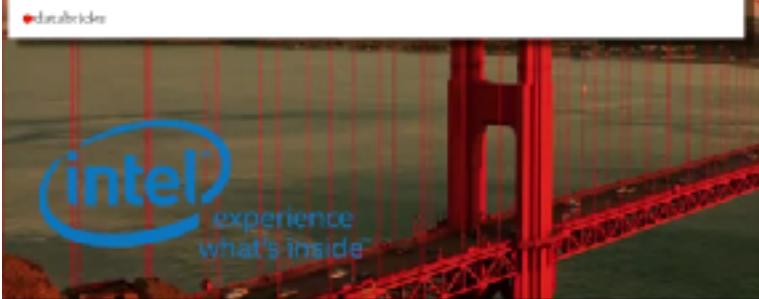


Write to data stores efficiently

```
private void partition() {
    // initialize pool or get one connection from pool in constructor
    // insert all rows in partition
    // return connection to pool
```



Monitoring and Upgrading



Streaming in Spark Web UI



Stats over last 1000 batches

New in Spark 1.4

For stability

Scheduling delay should be approx 0

Processing Time approx < batch interval

databricks

Streaming in Spark Web UI

Details of individual batches

Active Batches (1)					
Batch Time	Input Size	Scheduling Delay <small>(ms)</small>	Processing Time <small>(ms)</small>	Status	
2015/06/08 11:12:45	3 events	0 ms	-	processing	
Completed Batches (last 794 out of 794)					
Batch Time	Input Size	Scheduling Delay <small>(ms)</small>	Processing Time <small>(ms)</small>	Total Delay <small>(ms)</small>	
2015/06/08 11:12:45	2 events	0 ms	0.2 s	0.2 s	
2015/06/08 11:12:44	1 events	0 ms	0.4 s	0.4 s	
2015/06/08 11:12:43	2 events	0 ms	0.7 s	0.7 s	
2015/06/08 11:12:42	2 events	0 ms	0.3 s	0.3 s	
2015/06/08 11:12:41	2 events	0 ms	0.9 s	0.9 s	
2015/06/08 11:12:40	1 events	0 ms	0.4 s	0.4 s	

databricks

Streaming in Spark Web UI

Details of individual batches

Active batches (1)

Batch Time	Input Size	Scheduling Delay	Processing Time	Status
2015/06/08 11:10:45	3 events	0 ms	-	processing

Completed Batches (last 10 min)

Batch Time
2015/06/08 11:10:45
2015/06/08 11:10:44
2015/06/08 11:10:43
2015/06/08 11:10:42
2015/06/08 11:10:41
2015/06/08 11:10:40

Details of batch at 2015/06/08 11:10:35

Batch Duration	Input data size	Scheduling delay	Processing time	Total delay
1 s	4 records	2 ms	1 s	1 s

Details of Spark jobs run in a batch

Output Depth	Description	Duration	Job ID	Duration	Stages: Succeeded/Total	Tasks (for 1)
0	foreachRDD at StreamingApp.scala:25	1 s	2950	0 ms	2/2	
1	foreachRDD at StreamingApp.scala:27	0 ms	2951	2 ms	1/1 (1 stages)	
			2952	1 ms	1/1 (1 stages)	

• databricks.com

Operational Monitoring

Streaming app stats published through [Codahale](#) metrics

Ganglia sink, Graphite sink, custom Codahale metrics sinks

Can see long term trends, across hours and days

Configure the metrics using `$SPARK_HOME/conf/metrics.properties`

Need to compile Spark with Ganglia LGPL profile for Ganglia support
(see <http://spark.apache.org/docs/latest/monitoring.html#metrics>)

• databricks.com

44

Programmatic Monitoring

`StreamingListener` – Developer interface to get internal events
onBatchSubmitted, onBatchStarted, onBatchCompleted,
onReceiverStarted, onReceiverStopped, onReceiverError

Take a look at `StreamingJobProgressListener` (private class) for inspiration

• [databrides](#)



SAN FRANCISCO • JUNE 15 - 17, 2015



Upgrading Apps

1. Shutdown your current streaming app `gracefully`
Will process all data before shutting down cleanly
`streamingContext.stop(stopGracefully = true)`
2. Update app code and start it again

Cannot upgrade from previous checkpoints if code changes or Spark version changes

• [databrides](#)



SAN FRANCISCO • JUNE 15 - 17, 2015



Much to say I have ... but time I have not

Memory and GC tuning
Using SQLContext
DStream.transform operation

Refer to online guide

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

#databricks

 Overview Programming Guide API Docs Deploying

Spark Streaming Programming Guide

- Directories
- [Quick Start](#)
- [Basic Concepts](#)
 - [Links](#)
 - [Initializing StreamingContext](#)
 - [Operated Stream \(Consumer\)](#)
 - [Input DStreams and Broadcasts](#)
 - [Transformations on DStreams](#)
 - [Output Operations on DStreams](#)
 - [DataFrames and SQL Operations](#)
 - [Machine Learning](#)
 - [Caching / Persistence](#)
 - [Checkpointing](#)
 - [Deploying Applications](#)
 - [Monitoring Applications](#)
 - [Performance Tuning](#)
 - [Reducing the Data Processing Time](#)
 - [Setting maxBatchSize / maxFusion](#)
 - [Memory Tuning](#)
 - [Fault-tolerance Semantics](#)
 - [Migration Guide from 0.8.1 or below to 1.0](#)
- [Where to Get more Help](#)

Overview

Spark streaming is an extension of the core spark API that enables scalable, high-throughput streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis and TCP sockets. It provides high-level API for defining complex processing logic over streams. These algorithms are expressed with high-level functions like map, reduce, join and window. Finally, it provides low-level API for direct interaction with the distributed storage.

Spark - stateful stream processing on SQL Engine

<https://www.youtube.com/watch?v=aV0L7R8NH7Y&t=32s>

The video player interface shows a presentation slide. On the left, there is a small video thumbnail of a man speaking at a podium. The main content area has a dark background with a teal diagonal stripe pattern. At the top right, the title "Structured Streaming" is displayed. Below it, the subtitle "stream processing on Spark SQL engine" is followed by the text "fast, scalable, fault-tolerant". Further down, the subtitle "rich, unified, high level APIs" is followed by the text "deal with complex data and complex workloads". At the bottom, the subtitle "rich ecosystem of data sources" is followed by the text "integrate with many storage systems". In the bottom right corner of the slide, there is a small "databricks" logo. The video player interface includes standard controls like play/pause, volume, and progress bar.

The video player interface shows a presentation slide. On the left, there is a small video thumbnail of a man speaking at a podium. The main content area has a dark background with a teal diagonal stripe pattern. In the center, there is a large, bold, orange text quote: "you should not have to reason about streaming". In the bottom right corner of the slide, there is a small "databricks" logo. The video player interface includes standard controls like play/pause, volume, and progress bar.

you
should write simple queries
&
Spark
should continuously update the answer

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

Treat Streams as Unbounded Tables

data stream unbounded input table

new data in the data stream
≡
new rows appended to a unbounded table

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

Anatomy of a Streaming Query

Example

Read JSON data from Kafka
Parse nested JSON
Store in structured Parquet table
Get end-to-end failure guarantees

```
graph LR; JSON["{json}"] --> Kafka["Kafka"]; Kafka --> ETL[ETL]; ETL --> Parquet["Parquet"]; style JSON fill:#0072bc,color:#fff; style Parquet fill:#0072bc,color:#fff;
```

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")  
.option("kafka.bootstrap.servers", "...")  
.option("subscribe", "topic")  
.load()
```

Source
Specify where to read data from
Built-in support for Files / Kafka / Kinesis*
Can include multiple sources of different types using `join()` / `union()`

*Available only on [Databricks Runtime](#)

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks



Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", ...)
    .option("subscribe", "topic")
    .load()
    .selectExpr("cast (value as string) as json")
    .select(from_json("json", schema).as("data"))
```

Transformations

- Cast bytes from Kafka records to a string, parse it as a json, and generate nested columns
- 100s of built-in, optimized SQL functions like `from_json`
- User-defined functions, lambdas, function literals with `map`, `flatMap`...

● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY ● databricks



Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", ...)
    .option("subscribe", "topic")
    .load()
    .selectExpr("cast (value as string) as json")
    .select(from_json("json", schema).as("data"))
    .writeStream
    .foreach(parquet())
    .option("path", "/parquetTable/")
```

Sink

- Write transformed output to external storage systems
- Built-in support for files / Kafka
- Use `foreach` to execute arbitrary code with the output data
- Some sinks are transactional and exactly once (e.g. files)

● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY ● databricks

Anatomy of a Streaming Query

```

spark.readStream.format("kafka")
.option("kafka.bootstrap.servers", ...)
.option("subscribe", "topic")
.load()
.selectExpr("cast (value as string) as json")
.select(from_json($"json", schema).as("data"))
.writeStream
.format("parquet")
.option("path", "/parquetTable/")
.trigger("1 minute")
.option("checkpointLocation", "-")
.start()

```

Processing Details

Trigger: when to process data

- Fixed interval micro-batches
- As fast as possible micro-batches
- Continuously (new in Spark 2.3)

Checkpoint location: for tracking the progress of the query

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

Triggers:

The trigger settings of a streaming query defines the timing of streaming data processing, whether the query is going to executed as micro-batch query with a fixed batch interval or as a continuous processing query.

Trigger Types:

1. Not Specified- default one:

If no trigger setting is explicitly specified, then by default, the query will be executed in micro-batch mode, where micro-batches will be generated as soon as the previous micro-batch has completed processing.

```
import org.apache.spark.sql.streaming.Trigger
```

```
// Default trigger (runs micro-batch as soon as it can)

df.writeStream
  .format("console")
  .start()
```

2. Fixed Interval(micro-batches):

The query will be executed with micro-batches mode, where micro-batches will be kicked off at the user-specified intervals.

```
// ProcessingTime trigger with two-seconds micro-batch interval
```

```
df.writeStream
```

```
  .format("console")
```

```
.trigger(Trigger.ProcessingTime("2 seconds"))
.start()
```

- If the previous micro-batch completes within the interval, then the engine will wait until the interval is over before kicking off the next micro-batch.
- If the previous micro-batch takes longer than the interval to complete (i.e. if an interval boundary is missed), then the next micro-batch will start as soon as the previous one completes (i.e., it will not wait for the next interval boundary).
- If no new data is available, then no micro-batch will be kicked off.

3. One-time(micro-batch)

The query will execute *only one* micro-batch to process all the available data and then stop on its own. This is useful in scenarios you want to periodically spin up a cluster, process everything that is available since the last period, and then shutdown the cluster. In some cases, this may lead to significant cost savings.

```
// One-time trigger
df.writeStream
  .format("console")
  .trigger(Trigger.Once())
.start()
```

4. Continuous with fixed checkpoint interval

The query will be executed in the new low-latency, continuous processing mode. Read more about this in the [Continuous Processing section](#) below.

```
// Continuous trigger with one-second checkpointing interval
df.writeStream
  .format("console")
  .trigger(Trigger.Continuous("1 second"))
.start()
```

Recovering from Failures with Checkpointing

```
.option("checkpointLocation", "path/to/HDFS/dir")
```

In case of a failure or intentional shutdown, you can recover the previous progress and state of a previous query, and continue where it left off. This is done using checkpointing and write-ahead

logs. You can configure a query with a checkpoint location, and the query will save all the progress information (i.e. range of offsets processed in each trigger) and the running aggregates (e.g. word counts in the [quick example](#)) to the checkpoint location. This checkpoint location has to be a path in an HDFS compatible file system, and can be set as an option in the DataStreamWriter when [starting a query](#).

The slide features a photograph of a speaker on the left and a diagram on the right. The diagram illustrates the execution process:

```
graph LR; A[DataFrames, Datasets, SQL] --> B[Logical Plan]; B --> C[Optimized Plan]; C --> D[Kafka Source]; C --> E[Globalized Operator Placement]; C --> F[Parallel Task]
```

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

● databricks

The slide continues the execution process diagram from the previous slide:

```
graph LR; A[DataFrames, Datasets, SQL] --> B[Logical Plan]; B --> C[Optimized Plan]; C --> D[Kafka Source]; C --> E[Globalized Operator Placement]; C --> F[Parallel Task]
```

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

● databricks

Spark automatically streamifies!

Spark SQL converts batch-like query to a series of incremental execution plans operating on new micro-batches of data

● databricks

The diagram illustrates the process of streamification:

- DataFrames, Datasets, SQL** lead to the **Logical Plan**, which consists of:
 - Input Data
 - Project
 - Map
 - Filter
 - Window
- The **Logical Plan** leads to the **Optimized Plan**, which consists of:
 - Kafka Source
 - Globalized Generator
 - Project
- The **Optimized Plan** leads to a **Series of Incremental Execution Plans**, shown as three parallel horizontal boxes labeled $t=0$, $t=1$, and $t=2$. Each box contains a **Processor** icon and a **Write-ahead Log** icon.

Fault-tolerance with Checkpointing

Checkpointing

Saves processed offset info to stable storage
Saved as JSON for forward-compatibility

Allows recovery from any failure

Can resume after limited changes to your streaming transformations (e.g. adding new filters to drop corrupted data, etc.)

● databricks

The diagram illustrates fault-tolerance using checkpoints:

- A **wide shared log** is shown on the left, connected by arrows to three **Processor** boxes labeled $t=0$, $t=1$, and $t=2$.
- Curved green arrows indicate the flow of data from the log to each processor.
- Below the processors, the text **end-to-end exactly-once guarantees** is displayed.

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", ...)
    .option("subscribe", "topic")
    .load()
    .selectExpr("cast (value as string) as json")
    .select(from_json($"json", "schemas").as("data"))
    .writeStream
    .format("parquet")
    .option("path", "/parquetTable")
    .trigger("1 minute")
    .option("checkpointLocation", "...")
    .start()
```



Raw data from Kafka available
as structured data in seconds,
ready for querying.

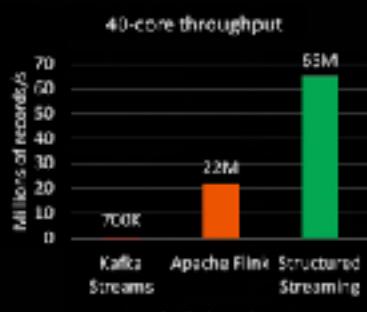
● databricks

SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks

Performance: YAHOO! Benchmark

Structured Streaming reuses
the **Spark SQL Optimizer**
and **Tungsten Engine**

3x
faster



More details in our [blog post](#).

● databricks

SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks



Stateful Stream Processing

SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks



What is Stateless Stream Processing?

Stateless streaming queries (e.g. ETL) process each record independent of other records

```
df.select(from_json("json", schema).as("data"))  
    .where("data.type = "typeB")
```

Every record is parsed into a structured form and then selected (or not) by the filter



SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks



What is Stateful Stream Processing?

Stateful streaming queries combine information from multiple records together

State is the information that is maintained for future use

```
df = select * from json("json", "adreno.json")  
      .where("data.type = 'typeB'")  
      .count()
```

Count is the streaming state and every selected record increments the count



● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY ● databricks



Stateful Micro-Batch Processing

State is versioned between micro-batches while streaming query is running



● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY ● databricks

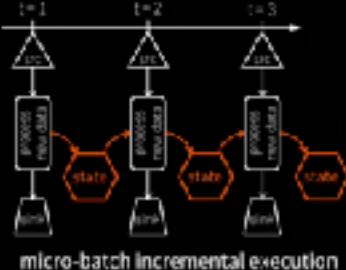


Stateful Micro-Batch Processing

State is versioned between micro-batches while streaming query is running

Each micro-batch reads previous version state and updates it to new version

Versions used for fault recovery



micro-batch incremental execution

● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY  databricks



Distributed, Fault-tolerant State

State data is distributed across executors

State stored in the executor memory

Micro-batch tasks update the state



driver → tasks → executor 1 (state)
executor 2 (state)

● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY  databricks

Distributed, Fault-tolerant State

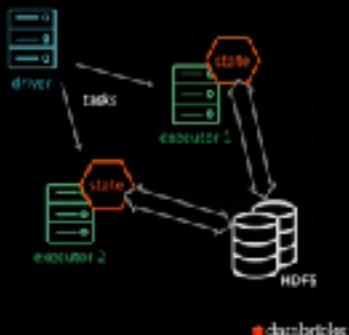
State data is distributed across executors

State stored in the executor memory

Micro-batch tasks update the state

Changes are checkpointed with version to given checkpoint location (e.g. HDFS)

Recovery from failure is automatic



SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks

Philosophy of Stateful Operations

SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks



Distributed, Fault-tolerant State

State data is distributed across executors

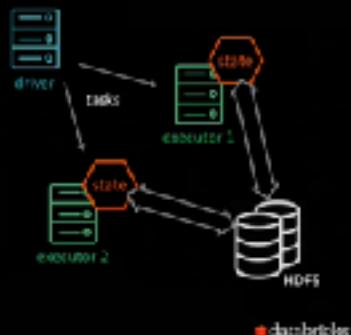
State stored in the executor memory

Micro-batch tasks update the state

Changes are checkpointed with version to given checkpoint location (e.g. HDFS)

Recovery from failure is automatic

Exactly-once fault-tolerance guarantees!

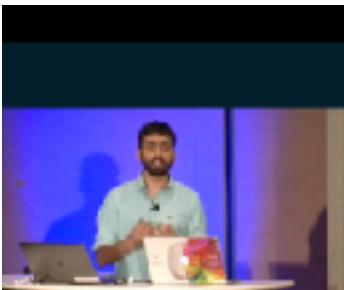


SPARK+AI
SUMMIT 2018

ORGANIZED BY databricks

|| ▶ 🔍 12.28 / 48.89

...



Philosophy of Stateful Operations

SPARK+AI
SUMMIT 2018

ORGANIZED BY databricks

databricks



Two types of Stateful Operations

Automatic State
Cleanup

User-defined State
Cleanup

 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks



Rest of this talk

Explore [built-in stateful operations](#)

[How to use watermarks to control state size](#)

[How to build arbitrary stateful operations](#)

[How to monitor and debug stateful queries](#)

 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks



Streaming *Aggregation*

 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks



Aggregation by key and/or time windows

Aggregation by key only

```
events
    .groupByKey("key")
    .count()
```

 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks



Aggregation by key and/or time windows

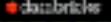
Aggregation by key only

```
events
  .groupby("key")
  .count()
```

Aggregation by event time windows

```
events
  .groupby(window("timestep","10 mins"))
  .avg("value")
```

SPARK+AI SUMMIT 2018
ORGANIZED BY 





Aggregation by key and/or time windows

Aggregation by key only

```
events
  .groupby("key")
  .count()
```

Aggregation by event time windows

```
events
  .groupby(window("timestep","10 mins"))
  .avg("value")
```

Aggregation by both

Supports multiple aggregations,
user-defined functions (UDAFs)!

```
events
  .groupby(
    col("key"),
    window("timestep","10 mins"))
  .agg(avg("value"), corr("value"))
```

SPARK+AI SUMMIT 2018
ORGANIZED BY 





Automatically handles Late Data

Keeping state allows late data to update counts of old windows

But size of the state increases indefinitely if old windows are not dropped

red = state updated with late data



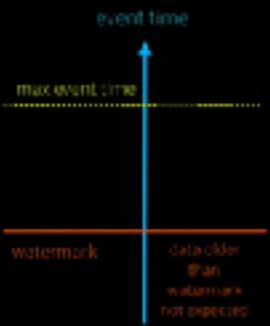
● datibricks

SPARK+AI SUMMIT 2018
ORGANIZED BY datibricks



Watermarking

Watermark - moving threshold of how late data is expected to be and when to drop old state



● datibricks

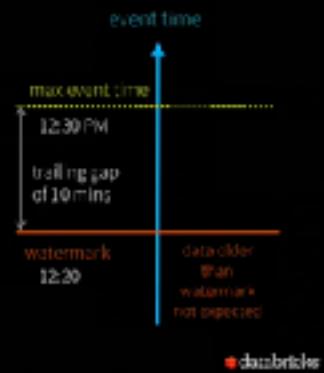
SPARK+AI SUMMIT 2018
ORGANIZED BY datibricks

Watermarking

Watermark - moving threshold of how late data is expected to be and when to drop old state

Trails behind **max event time** seen by the engine

Watermark delay = trailing gap



SPARK+AI
SUMMIT 2018
SPONSORED BY databricks

|| ▶ 🔍 17:46 / 48:39

✖ ✎ 🔍 ↻

Watermarking

Data newer than watermark may be late, but allowed to aggregate



SPARK+AI
SUMMIT 2018
SPONSORED BY databricks

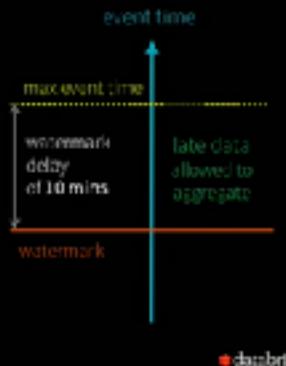
|| ▶ 🔍 17:49 / 48:39

✖ ✎ 🔍 ↻



Watermarking

Data newer than watermark may be late, but allowed to aggregate



● dazbricks

SPARK+AI SUMMIT 2018
ORGANIZED BY  dazbricks



Watermarking

Data newer than watermark may be late, but allowed to aggregate

Data older than watermark is "too late" and dropped

Windows older than watermark automatically deleted to limit state



● dazbricks

SPARK+AI SUMMIT 2018
ORGANIZED BY  dazbricks

Watermarking

```
parsedData.  
withTimestamp("timestamp", "10 minutes")  
.groupByKey("timestamp", "5 minutes")  
.count()
```

Used only in stateful operations
Ignored in non-stateful streaming queries and batch queries

event time

max event time

watermark delay of 10 mins

late data allowed to aggregate

watermark

data too late, dropped

● dazbricks

SPARK+AI SUMMIT 2018
ORGANIZED BY dazbricks

Watermarking

```
parsedData.  
withTimestamp("timestamp", "10 minutes")  
.groupByKey("timestamp", "5 minutes")  
.count()
```

Event Time

Processing Time

More details in my [blog post](#).

● dazbricks

SPARK+AI SUMMIT 2018
ORGANIZED BY dazbricks

Watermarking

EventTime
12:05
12:00
12:00
Processing Time
12:05 12:06 12:07 12:08 12:09 12:10 12:11 12:12

graph LR; A((12:05)) --> B((12:06)); B --> C((12:07)); C --> D((12:08)); D --> E((12:09)); E --> F((12:10)); F --> G((12:11)); G --> H((12:12))

10 min

watermark: 12:04

data is late, but considered in counts

More details in my [blog post](#).

Organized by databricks

Watermarking

EventTime
12:05
12:00
12:00
Processing Time
12:05 12:06 12:07 12:08 12:09 12:10 12:11 12:12

graph LR; A((12:05)) --> B((12:06)); B --> C((12:07)); C --> D((12:08)); D --> E((12:09)); E --> F((12:10)); F --> G((12:11)); G --> H((12:12))

10 min

watermark: 12:04

data is late, but considered in counts

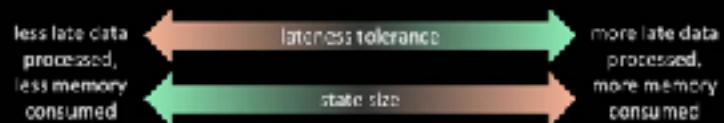
data too late, ignored in counts, state dropped

More details in my [blog post](#).

Organized by databricks

Watermarking

Trade off between lateness tolerance and state size



 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks

Streaming *Deduplication*

 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks



Streaming Deduplication

Drop duplicate records in a stream

Specify columns which uniquely identify a record

```
userActions  
.dropDuplicates("uniqueRecordId")
```

Spark SQL will store past unique column values as state and drop any record that matches the state

● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY  databricks



Streaming Deduplication with Watermark

Timestamp as a unique column along with watermark allows old values in state to dropped

Records older than watermark delay is not going to get any further duplicates

```
userActions  
.withWatermark("timestamp")  
.dropDuplicates(  
"uniqueRecordId",  
"timestamp")
```

Timestamp must be same for duplicated records

● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY  databricks

Streaming Joins

SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks

Streaming Joins

Spark 2.0+ supports joins between streams and static datasets
Spark 2.3+ supports joins between multiple streams

SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks

Streaming Joins

Spark 2.0+ supports joins between streams and static datasets

Spark 2.3+ supports joins between multiple streams

Example: Ad Monetization

Join stream of ad **impressions** with another stream of their corresponding user **clicks**



• databricks



Streaming Joins

Spark 2.0+ supports joins between streams and static datasets

Spark 2.3+ supports joins between multiple streams

Example: Ad Monetization

Join stream of ad **impressions** with another stream of their corresponding user **clicks**



• databricks



Streaming Joins

Most of the time click events arrive after their impressions
Sometimes, due to delays, impressions can arrive after clicks

Each stream in a join needs to buffer past events as **state** for matching with future events of the other stream

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

Simple Inner Join

Inner join by ad ID column

```
impressions.join(  
  clicks,  
  expr("clickAdId = impressionAdId")  
)
```

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

|| ▶ ⏪ 25.26 / 48.89 ⏴ 🔍 ↻

Simple Inner Join

Inner join by ad ID column

Need to buffer all past events as state, a match can come on the other stream any time in the future

```
impressions.join(  
  clicks,  
  expr("clickAdId = impressionAdId")  
)
```



© databricks

 SPARK+AI
SUMMIT 2018
ORGANIZED BY 

Simple Inner Join

Inner join by ad ID column

Need to buffer all past events as state, a match can come on the other stream any time in the future

```
impressions.join(  
  clicks,  
  expr("clickAdId = impressionAdId")  
)
```



© databricks

 SPARK+AI
SUMMIT 2018
ORGANIZED BY 



Inner Join + Time constraints + Watermarks

Time constraints

Let's assume

- Impressions can be 2 hours late
- Clicks can be 3 hours late
- A click can occur within 1 hour after the corresponding impression

```
val impressionsWithWatermark = impressions.  
    .withWatermark("impressionTime", "2 hours")  
  
val clicksWithWatermark = clicks.  
    .withWatermark("clickTime", "3 hours")  
  
impressionsWithWatermark.join(  
    clicksWithWatermark,  
    expr(  
        "clickId = impressionId AND  
        clickTime >= impressionTime AND  
        clickTime <= impressionTime + interval 1 hour  
    ))
```

Range Join

● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY ● databricks



Arbitrary Stateful Operations

● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY ● databricks



Arbitrary Stateful Operations

Many use cases require more complicated logic than SQL ops

Example: Tracking user activity on your product

Input: User actions (login, clicks, logout, ...)
Output: Latest user status (online, active, inactive, ...)

SPARK+AI SUMMIT 2018
ORGANIZED BY  databricks



Arbitrary Stateful Operations

Many use cases require more complicated logic than SQL ops

Example: Tracking user activity on your product

Input: User actions (login, clicks, logout, ...)
Output: Latest user status (online, active, inactive, ...)

Solution: `MapGroupsWithState` / `FlatMapGroupsWithState`
General API for per-key user-defined stateful processing
Since Spark 2.2, for Scala and Java only

SPARK+AI SUMMIT 2018
ORGANIZED BY  databricks

MapGroupsWithState / FlatMapGroupsWithState

No automatic state clean up and dropping of late data

Adding watermark does not automatically manage late and stale data

Explicit state clean up by the user

More powerful + efficient than DStream's mapWithState and updateStateByKey



databricks

MapGroupsWithState - How to use?

1. Define the data structures

- Input event: UserAction
- State store: UserStatus
- Output event: UserStatus
[can be different from state]

```
case class UserAction(  
    userId: String, action: String)  
  
case class UserStatus(  
    userId: String, active: Boolean)
```



databricks



MapGroupsWithState - How to use?

2. Define function to update state of each grouping key using the new data

- Input
 - Grouping key: userId
 - New data: new user actions
 - Previous state: previous status of this user

```
case class UserAction(userId: String, action: String)
case class UserStatus(userId: String, active: Boolean)

def updateState(
    userId: String,
    actions: Iterator[UserAction],
    state: GroupState[UserStatus]): UserStatus = {
```

● dazbricks

SPARK+AI SUMMIT 2018
ORGANIZED BY  dazbricks



MapGroupsWithState - How to use?

2. Define function to update state of each grouping key using the new data

- Body
 - Get previous user status
 - Update user status with actions
 - Update state with latest user status
 - Return the status

```
def updateState(
    userId: String,
    actions: Iterator[UserAction],
    state: GroupState[UserStatus]): UserStatus = {
    val prevState = state.getOption.getOrElse(
        new UserStatus()
    )
    actions.foreach { action =>
        prevState.updateWith(action)
    }
    state.update(prevState)
    return prevState
}
```

● dazbricks

SPARK+AI SUMMIT 2018
ORGANIZED BY  dazbricks

MapGroupsWithState - How to use?

3. Use the **user-defined function** on a grouped Dataset

```
def updateState(  
    userId: String,  
    actions: Iterator[UserAction],  
    state: GroupState[UserStatus]): UserStatus = {  
    // process actions, update and return status  
}  
  
userActions  
.groupByKey(_._userId)  
.mapGroupsWithState(updateState)
```

● databricks



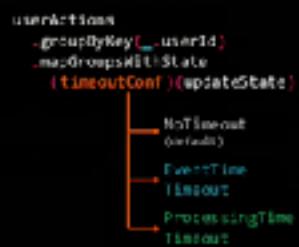
Timeouts

Example: Mark a user as inactive when there is no actions in 1 hour

Timeouts: When a group does not get any event for a while, then the function is called for that group with an empty iterator

Must specify a global timeout type, and set per-group timeout timestamp/duration

Ignored in a batch queries



● databricks





Event-time Timeout - How to use?

1. Enable EventTimeTimeout in mapGroupsWithState
2. Enable watermarking
3. Update the mapping function
 - Every time function is called, set the timeout timestamp using the max seen event timestamp + timeout duration

```
userstate from
    .withWatermark("timestamp")
    .groupByKey(_> userId)
    .mapGroupsWithState
        (EventTimeTimeout)(updateState)
```

```
def updateState(...): UserStatus = {
    // track maxActionTimestamp while
    // processing actions and updating state
    state.setTimestamp(maxActionTimestamp +
        maxActionTimestamp, "1 hour")

    // return user status
}
```

SPARK+AI SUMMIT 2018
ORGANIZED BY 



Event-time Timeout - How to use?

1. Enable EventTimeTimeout in mapGroupsWithState
2. Enable watermarking
3. Update the mapping function
 - Every time function is called, set the timeout timestamp using the max seen event timestamp + timeout duration
 - Update state when timeout occurs

```
userstate from
    .withWatermark("timestamp")
    .groupByKey(_> userId)
    .mapGroupsWithState
        (EventTimeTimeout)(updateState)
```

```
def updateState(...): UserStatus = {
    if (!state.hasTimeout) {
        // track maxActionTimestamp while
        // processing actions and updating state
        state.setTimestamp(maxActionTimestamp +
            maxActionTimestamp, "1 hour")
    } else { // if normal timeout
        userstatus.handleTimeout()
        state.remove()
    }
    // return user status
}
```

SPARK+AI SUMMIT 2018
ORGANIZED BY 



Event-time Timeout - When?

Watermark is calculated with max event time across all groups

For a specific group, if there is no event till watermark exceeds the timeout timestamp,

Then

Function is called with an empty iterator, and hasTimedOut = true

Else

Function is called with new data, and timeout is disabled

Needs to explicitly set timeout timestamp every time

● databricks



ORGANIZED BY ● databricks

Function Output Mode

Function output mode* gives Spark insights into the output from this opaque function

Update Mode - Output events are key-value pairs, each output is updating the value of a key in the result table

Append Mode - Output events are independent rows that being appended to the result table

Allows Spark SQL planner to correctly compose FlatMapGroupsWithState with other operations

*NOT to be confused with output mode of the query



● databricks



ORGANIZED BY ● databricks



Managing Stateful Streaming Queries

 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks



Optimizing Query State

Set # shuffle partitions to 1-3 times number of cores

Too low = not all cores will be used → lower throughput

Too high = cost of writing state to HDFS will increase → higher latency

 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks

 databricks

Optimizing Query State

Set # shuffle partitions to 1-3 times number of cores

Too low = not all cores will be used → lower throughput

Too high = cost of writing state to HDFS will increases → higher latency

Total size of state per worker

Larger state leads to higher overheads of snapshotting, JVM GC pauses, etc.



ORGANIZED BY databricks

|| ⏪ ⏹ 43:40 / 48:39

databricks

Monitoring the state of Query State

Get current state metrics using the last progress of the query

Total number of rows in state

Total memory consumed (approx.)

```
val progress = query.lastProgress  
print(progress.json)
```

```
{  
  "TotalOperations": 1,133,  
  "QualifiedTables": 6,658,949,  
  "MemoryUsedMB": 1,128,513,847  
}
```



ORGANIZED BY databricks

databricks



Monitoring the state of Query State

Get current state metrics using the last progress of the query

Total number of rows in state
Total memory consumed (approx.)

```
val progress = query.lastProgress
progress.json
```

```
{
  "lastProgress": {
    "totalOperations": 114,
    "unifiedTable": 640000,
    "memoryAllocated": 138897387
  }
}
```

Get it asynchronously through StreamingQueryListener API

```
new StreamingQueryListener {
  ...
  def onQueryProgress(
    event: QueryProgressEvent)
  ...
}
```

● databricks

SPARK+AI SUMMIT 2018
ORGANIZED BY ● databricks



Monitoring the state of Query State

Databricks Notebooks integrated with Structured Streaming
Shows size of state along with other processing metrics



● databricks

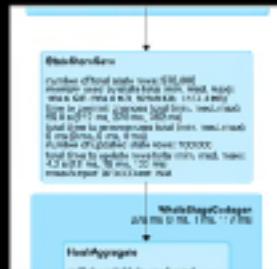
SPARK+AI SUMMIT 2018
ORGANIZED BY ● databricks

Debugging Query State

SQL metrics in the Spark UI (SQL tab, DAG view) expose more operator-specific stats

Answer questions like

- Is the memory usage skewed?
 - Is removing rows slow?
 - Is writing checkpoints slow?

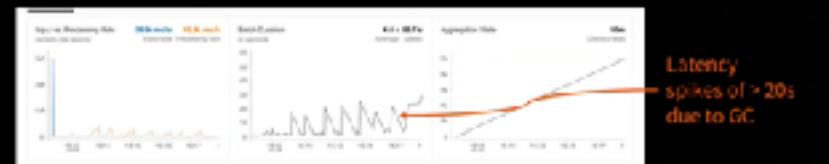


Managing Very Large State

State data kept on JVM heap

Can have GC issues with millions of state keys per worker

Limits depend on the size and complexity of state data structures





Managing Very Large State with RocksDB

In Databricks Runtime, you can store state locally in RocksDB
Avoids JVM heap, no GC issues with 100 millions state keys per worker
Local RocksDB snapshot files automatically checkpointed to HDFS
Same exactly-once fault-tolerant guarantees



(More info in [Databricks Docs](#))



 SPARK+AI
SUMMIT 2018
ORGANIZED BY 



Continuous Processing

Millisecond-level latencies with [Continuous Processing engine](#)
Experimental release in Apache Spark 2.3
Shuffle and stateful operator support coming in future releases

More info:

Talk today!

TUESDAY, JUNE 5 - CONFERENCE

2:00pm **Continuous Processing in Structured Streaming**
Data Trends [Contributed]
Continuous Processing 101 [Contributed]

Blog post

[Introducing Low-Latency Continuous Processing Model in Structured Streaming in Apache Spark 2.3](#)
Available on [Databricks Runtime 4.0](#)



 SPARK+AI
SUMMIT 2018
ORGANIZED BY 





Data Pipeline @ Fortune 100 Company

Trillions of Records

- Cloud infrastructure and serverless
- Cloud databases and time series
- Serverless
- Network traffic

Messy data not ready for analytics

Extract ETL

Databricks

DATAWAREHOUSE

Complex ETL

Separate warehouses for each type of analytics

- DW1: Incident Response
- DW2: Billing
- DW3: Reports

Hours of delay in accessing data

Very expensive to scale

Proprietary formats

No advanced analytics (ML)

#databricks



New Pipeline @ Fortune 100 Company

Cloud infrastructure and serverless

Cloud databases and time series

Serverless

Network traffic

Spark STRUCTURED STREAMING → databricks DELTA → Spark SQL, ML, STREAMING

Incident Response

Auditing

Reports

Data usable in minutes/seconds

Easy to scale

Open formats

Enables advanced analytics

#databricks



DataFrame/Dataset

SQL	DataFrame	Dataset
<pre>spark.sql("SELECT type, user_id FROM ...") # SQL type, user_id</pre>	<pre>val df = DataFrame(spark.read("...")) # DataFrame type</pre>	<pre>val ds = Dataset.create(DataType.String, DatasetType.Dataset) # Dataset type</pre>
Most familiar to BI Analysts Supports SQL, DataFrame	Great for Data Scientists familiar with Pandas/R DataFrames	Created for Data Engineers who want compile-time type safety

Same semantics, same performance
Choose your hammer for whatever nail you have!





Anatomy of a Streaming Query

```
spark.readStream.format("kafka")  
    .option("kafka.bootstrap.servers", "...")  
    .option("subscribe", "topic")  
    .load()  
    .selectExpr("value as stream")  
    .select(from_json("stream", schema).as("data"))
```

Transformations

- Cast bytes from Kafka records to a string, parse it as a json, and generate nested columns
- 100s of built-in, optimized SQL functions like `from_json`
- User-defined functions, lambdas, function literals with `map`, `flatMap`...





Anatomy of a Streaming Query

```
spark.readStream().format("kafka")
    .option("kafka.bootstrap.servers", ...)
    .options("subscribe", "mytopic")
    .load()
    .selectExpr("value as string") as json
    .selectFrom_json("json", schema).as("data")
    .writeStream
        .format("console")
        .options("checkpointInterval", "10minutes")
        .start()
```

Sink

- write transformed output to external storage systems
- Built-in support for Files / Kafka
- use foreach to execute arbitrary code with the output data
- Some sinks are transactional and exactly once (e.g. Files)

© databricks



Anatomy of a Streaming Query

```
spark.readStream().format("kafka")
    .option("kafka.bootstrap.servers", ...)
    .options("subscribe", "mytopic")
    .load()
    .selectExpr("value as string") as json
    .selectFrom_json("json", schema).as("data")
    .writeStream
        .format("console")
        .options("checkpointInterval", "10minutes")
        .trigger("5 minutes")
        .options("checkpointsLocation", "/tmp/checkpoints")
        .start()
```

Processing Details

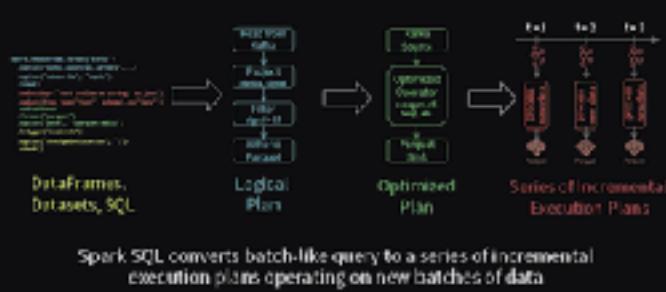
- Trigger: when to process data
 - Fixed interval micro-batches
 - As fast as possible micro-batches
 - Continuously (new in Spark 2.3)

The checkpoint location: for tracking the progress of the query

© databricks



Spark automatically streamifies!



Spark SQL converts batch-like query to a series of incremental execution plans operating on new batches of data

© databricks



Business Logic independent of Execution Mode



```
spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", ...)
    .option("subscribe", "topic1")
    .load()
    .selectExpr("value as stream")
    .selectExpr("from_json(stream, schema).as('data')")
    .writeStream
    .format("parquet")
    .option("path", "/tmp/quarterly")
    .trigger("1 minute")
    .option("checkpointLocation", "/tmp")
    .start()
```

Business logic

© databricks



Business Logic independent of Execution Mode

```
spark.read.format("kafka")
  .option("topic", "streaming_servers")
  .option("subscribe", "topic")
  .load()
  .selectExpr("value as string as json")
  .selectFrom_json("json", "value).as("data"))
  .write
  .format("parquet")
  .option("path", "/tmp/questions.parquet")
  .mode("append")
```

Business logic remains unchanged

Peripheral code decides whether it's a batch or a streaming query

© databricks



Business Logic independent of Execution Mode

```
SELECT * FROM STREAMING_SOURCES
```

Batch Micro-batch streaming Continuous** Streaming

high latency (minutes) low latency (seconds) ultra-low latency (milliseconds)

execute on-demand efficient resource allocation static resource allocation

high throughput high throughput high throughput

** incremental refresh (append and update) © databricks



Event time Aggregations

Windowing is just another type of grouping in Struct. Streaming

number of records every hour

```
parameters  
    -> window("hour", "1 hour")  
    -> count()
```

avg signal strength of each device every 10 mins

```
parameters  
    -> groupByKey()  
    -> window("10 mins", "10 mins")  
    -> avg("signal")
```

Support UDAFs!

© databricks

B



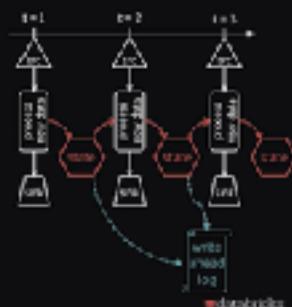
Stateful Processing for Aggregations

Aggregates has to be saved as distributed state between triggers

Each trigger reads previous state and writes updated state

State stored in memory, backed by write-ahead log in HDFS

Fault-tolerant, exactly-once guaranteed



B



Evolution of a Cutting-Edge Data Pipeline

Events → kafka ?

Streaming Analytics

Data Lake

Reporting

datastrada



Evolution of a Cutting-Edge Data Pipeline

Events → kafka → Spark → Streaming Analytics

Streaming Analytics

Data Lake

Reporting

datastrada



Challenge #1: Historical Queries?

Events → **kafka** → **Spark** → **Streaming Analytics** → **X-arch**

Events → **kafka** → **Spark** → **Data Lake** → **Spark** → **Reporting**

① Search
② Validation

© datadrifts



Challenge #2: Messy Data?

Events → **kafka** → **Spark** → **Streaming Analytics** → **X-arch**

Events → **kafka** → **Spark** → **Data Lake** → **Spark** → **Validation** → **Streaming Analytics** → **X-arch**

① Search
② Validation

© datadrifts



Challenge #3: Mistakes and Failures?

Events → **kafka** → **Spark** → **Streaming Analytics**

Spark → **Validation** → **Reporting**

Spark → **Data Lake** → **Reprocessing**

Legend:

- 1. X-search
- 2. Validation
- 3. Reprocessing

© datadrifts



Challenge #4: Query Performance?

Events → **kafka** → **Spark** → **Streaming Analytics**

Spark → **Validation** → **Reporting**

Spark → **Data Lake** → **Reprocessing**

Spark → **Compaction** → **Reporting**

Legend:

- 1. X-search
- 2. Validation
- 3. Reprocessing
- 4. Compaction

Scheduled to [watch on YouTube](#)

© datadrifts

Real-Time Data Pipelines Made Easy with Structured Streaming in Apache Spark | DataEngConf SF '10

Press Esc to exit full screen

Challenge #4: Query Performance?

The diagram illustrates a complex real-time data pipeline. It starts with 'Events' entering a Kafka cluster. These events are processed by a first Spark cluster, which performs 'Search' (1) and 'Validation' (2). The results from this cluster are then used for 'Streaming Analytics'. Simultaneously, the data is sent to a second Spark cluster for 'Reprocessing' (3), which then writes to a 'Data Lake'. From the Data Lake, the data is read by a third Spark cluster, which performs 'Compact & Evaluate' (4). Finally, the results are used for 'Reporting'. A legend on the right defines the icons: 1. Search, 2. Validation, 3. Reprocessing, and 4. Compaction.

Real-Time Data Pipelines Made Easy with Structured Streaming in Apache Spark | DataEngConf SF '10

Challenge #4: Query Performance?

The diagram illustrates a complex real-time data pipeline. It starts with 'Events' entering a Kafka cluster. These events are processed by a first Spark cluster, which performs 'Search' (1) and 'Validation' (2). The results from this cluster are then used for 'Streaming Analytics'. Simultaneously, the data is sent to a second Spark cluster for 'Reprocessing' (3), which then writes to a 'Data Lake'. From the Data Lake, the data is read by a third Spark cluster, which performs 'Compact & Evaluate' (4). Finally, the results are used for 'Reporting'. A legend on the right defines the icons: 1. Search, 2. Validation, 3. Reprocessing, and 4. Compaction.

Let's try it instead with

databricks
DELTA

databricks

Real-Time Data Pipelines Made Easy with Structured Streaming in Apache Spark | DataEngConf SF '10

databricks
DELTA

- The SCALE of data lake
- The RELIABILITY & PERFORMANCE of data warehouses
- The LOW-LATENCY of streaming

databricks



THE GOOD OF DATA WAREHOUSES

- Pristine Data
- Transactional Reliability
- Fast Queries

THE GOOD OF DATA LAKES

- Massive scale on cloud storage
- Open Formats (Parquet, ORC)
- Predictions (ML) & Streaming

databricks



Databricks Delta Combines the Best

MASSIVE SCALE

Decouple Compute & Storage

RELIABILITY

ACID Transactions & Data Validation

PERFORMANCE

Data Indexing & Caching (10-100x)

OPEN

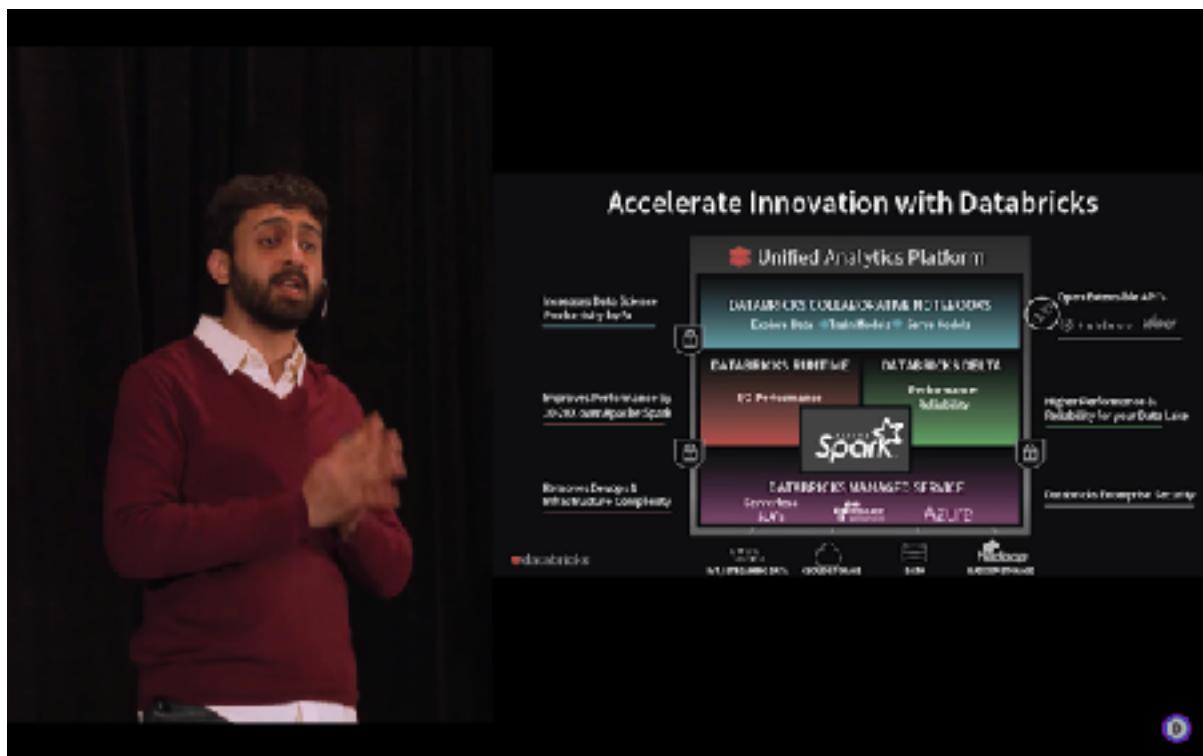
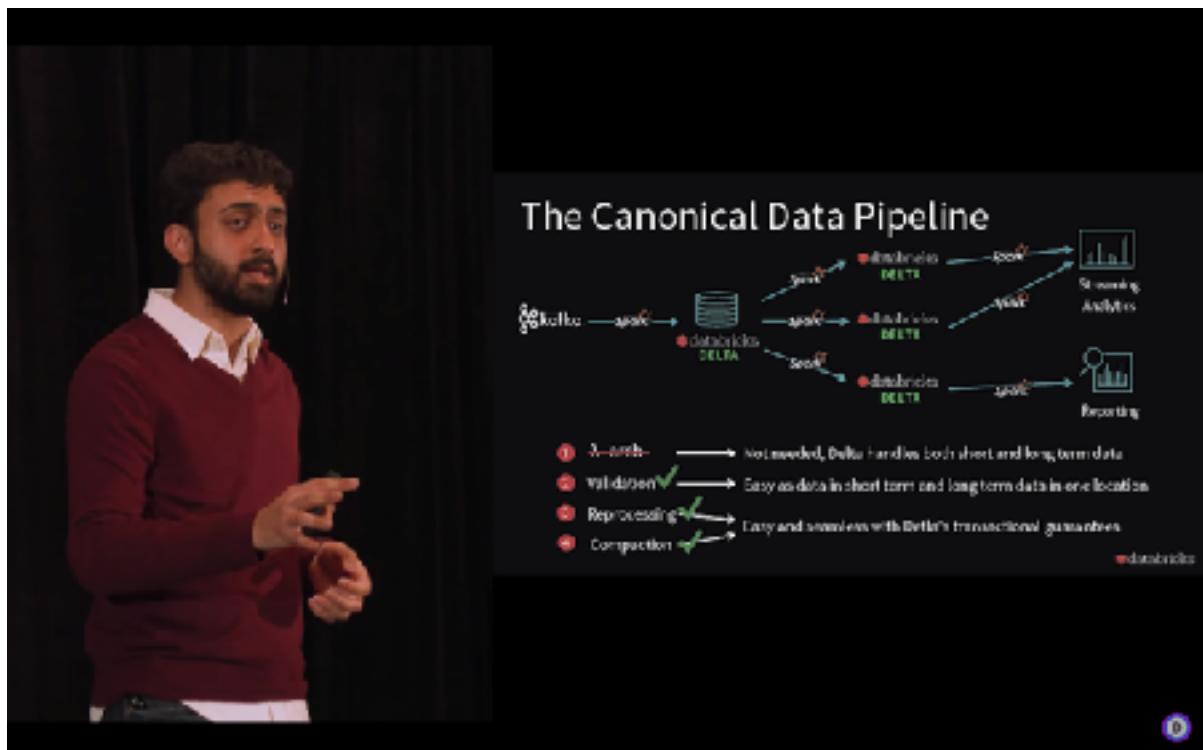
Data stored as Parquet, ORC, etc.

LOW LATENCY

Integrated with Structured Streaming

databricks







Spark - Continuous Processing Engine in Structured Streaming

<https://www.youtube.com/watch?v=tPaOEZewaek>

Continuous Processing in Structured Streaming

Jose Torres, Databricks

#Dev4SAIS

SPARK+AI
SUMMIT 2018

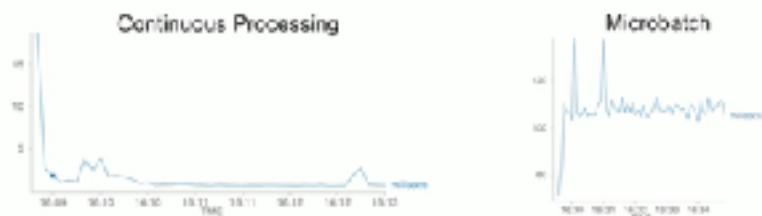
ORGANIZED BY databricks

▶ ⏪ ⏹ 0:24 / 25:49

CC 10 45

Continuous Processing Overview

- No microbatches
- Low (~1ms) latency
- Unified Spark SQL API



SPARK+AI
SUMMIT 2018

ORGANIZED BY databricks

|| ⏪ ⏹ 0:26 / 25:49

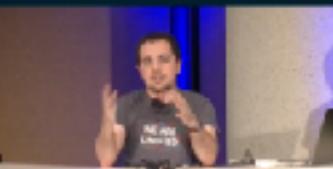
CC 10 45



DStream API

- Non-declarative, similar to RDDs
- Scala/Java only
- Checkpoints only through complete snapshots
- No event time

 **SPARK+AI**
SUMMIT 2018
ORGANIZED BY 



Structured Streaming

- Data abstracted as a virtual append-only table
- Unified Spark SQL query API
- Streaming and batch give the same answer

 **SPARK+AI**
SUMMIT 2018
ORGANIZED BY 

Structured Streaming

Data stream → Unbounded Table

new data in the data stream
≡
new rows appended to a unbounded table

Data stream as an unbounded table

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

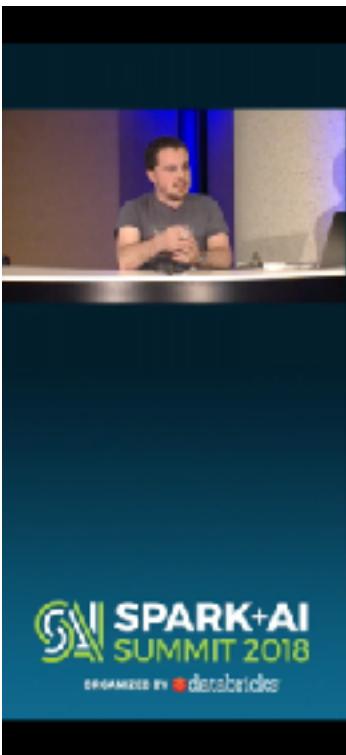
IDEAS

Structured Streaming Features

- Dataframes and Datasets
- SQL, Python, and R language APIs
- Delta-based aggregation state
- Event time and watermarks

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

IDEAS



Microbatches

time when events are available at source

time when processed events are written to sink

micro batch boundaries (thousand of seconds)

second-scale end-to-end latencies

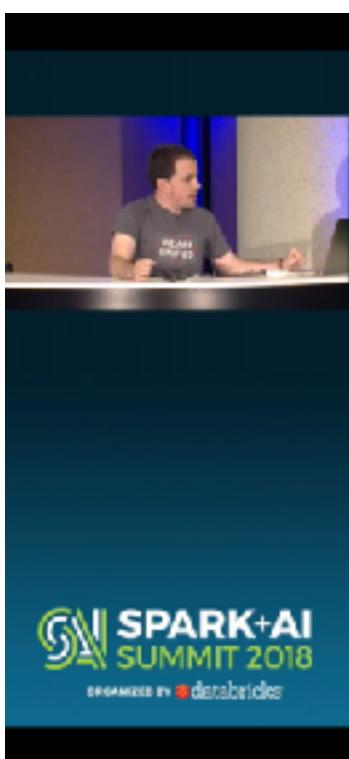
Second-scale end-to-end latencies with Micro-batch Processing

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

IDEAS

3

This slide illustrates microbatch processing. It shows two horizontal timelines: 'time when events are available at source' and 'time when processed events are written to sink'. Events are represented by colored dots (green, orange, red, blue) on the source timeline. On the sink timeline, these events are grouped into discrete batches, indicated by purple dashed boxes labeled 'micro batch boundaries (thousand of seconds)'. Arrows show the flow from source to sink. A red arrow labeled 'second-scale end-to-end latencies' spans the distance between the two timelines. The title 'Second-scale end-to-end latencies with Micro-batch Processing' is centered below the diagrams.



Continuous Processing

time when events are available at source

time when processed events are written to sink

long running Spark tasks continuously processing events

millisecond-scale end-to-end latencies

open markers for checkpointing progress

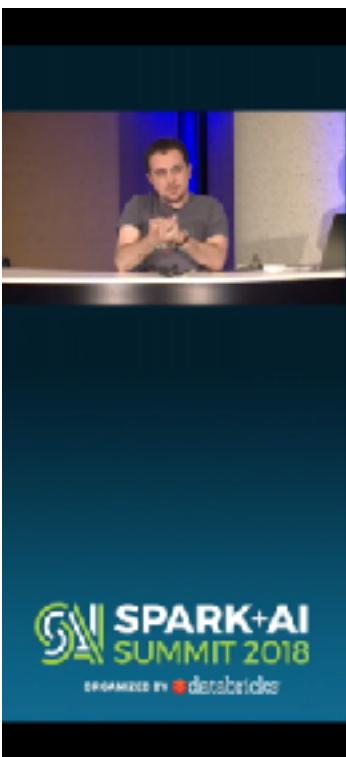
Millisecond-scale end-to-end latencies with Continuous Processing

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

IDEAS

4

This slide illustrates continuous processing. It shows two horizontal timelines: 'time when events are available at source' and 'time when processed events are written to sink'. Events are represented by colored dots (green, orange, red, blue) on the source timeline. On the sink timeline, these events are processed by 'long running Spark tasks' shown as vertical lines with dots. Arrows show the flow from source to sink. A red arrow labeled 'millisecond-scale end-to-end latencies' spans the distance between the two timelines. Open purple markers on the sink timeline are labeled 'open markers for checkpointing progress'. The title 'Millisecond-scale end-to-end latencies with Continuous Processing' is centered below the diagrams.



Chandy-Lamport Checkpoints

- Asynchronous
- Consistent

The diagram illustrates the Chandy-Lamport checkpointing process. A **Driver** node is connected to a **Data Stream**. The Data Stream consists of three sequential components: **Reader**, **Aggregation Task**, and **Writer Task**. The **Driver** sends an **epoch marker** to the **Reader**. The **Reader** sends an **arm checkpoint to state atom** signal to the **Aggregation Task**. The **Aggregation Task** sends a **partition level commit** signal to the **Writer Task**. Finally, the **Writer Task** sends a **checkpoint complete ready for global commit** signal back to the **Driver**.

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

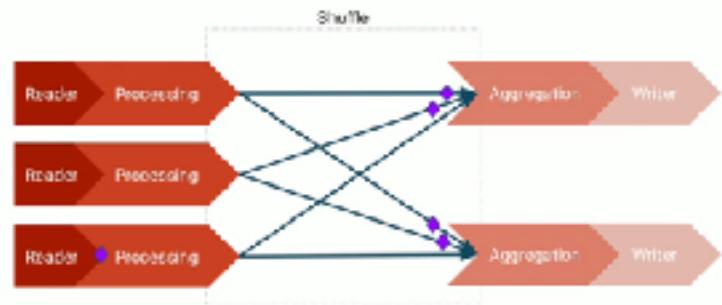


Checkpointing - Detailed

The detailed checkpointing diagram shows a **Driver** node connected to a **Shuffle** block. The **Shuffle** block is connected to three parallel **Reader** nodes, each labeled **Processing**. Each **Reader** node has a blue dot indicating it is active. The output of each **Reader** node is connected to a **Shuffle** block, which then connects to two **Aggregation** blocks. Each **Aggregation** block is followed by a **Writer** block. The **Shuffle** block also receives **epoch markers** from the **Driver**.

SPARK+AI SUMMIT 2018
ORGANIZED BY databricks

Checkpointing - Detailed

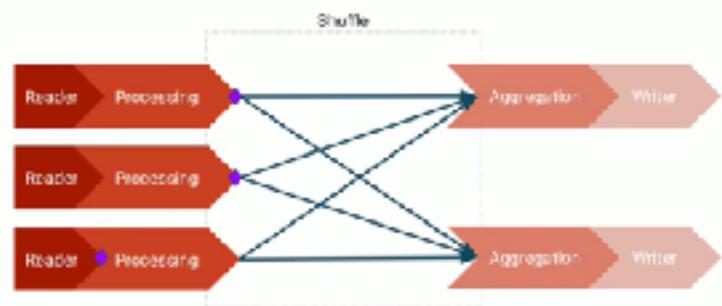


SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks

|| ▶ 🔍 13:16 / 25:49

CC 🔍 ↻

Checkpointing - Detailed



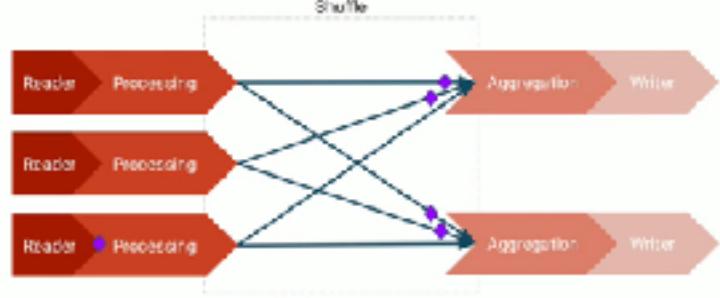
SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks

|| ▶ 🔍 13:14 / 25:49

CC 🔍 ↻



Checkpointing - Detailed

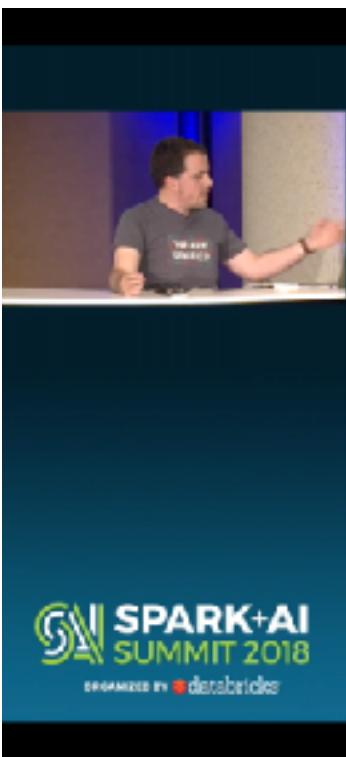


The diagram illustrates a three-stage pipeline for data processing:

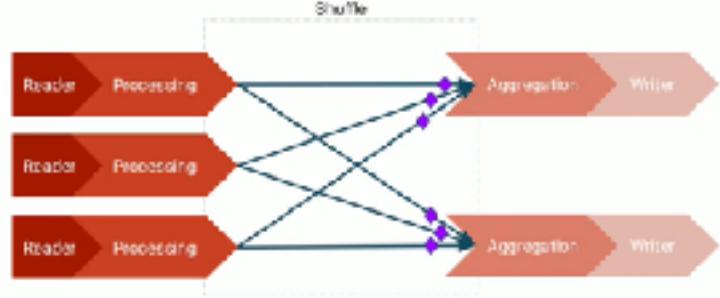
- Reader**: The first stage where data is read.
- Processing**: The second stage where data is processed.
- Shuffle**: A stage where data is shuffled between partitions. It is positioned between the Processing stage and the final stage.
- Aggregation**: The third stage where data is aggregated.
- Writer**: The final stage where data is written.

The process starts with three parallel Readers, each followed by a Processing stage. The output of the Processing stage feeds into the Shuffle stage. From the Shuffle stage, data is distributed to three parallel Aggregation stages, which then feed into a final Writer stage. The entire process is labeled "ID: 1EAS".

SPARK+AI SUMMIT 2018
ORGANIZED BY  databricks



Checkpointing - Detailed

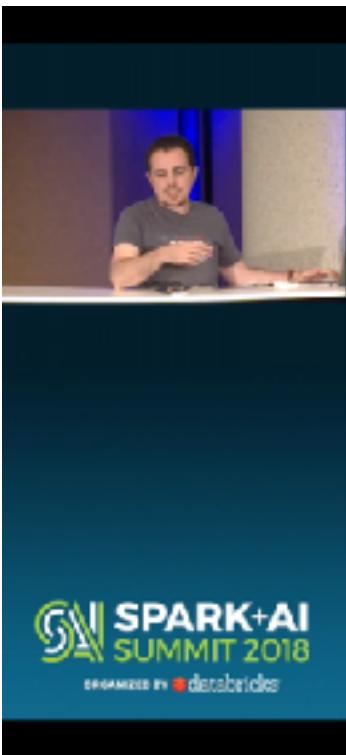


The diagram illustrates a three-stage pipeline for data processing:

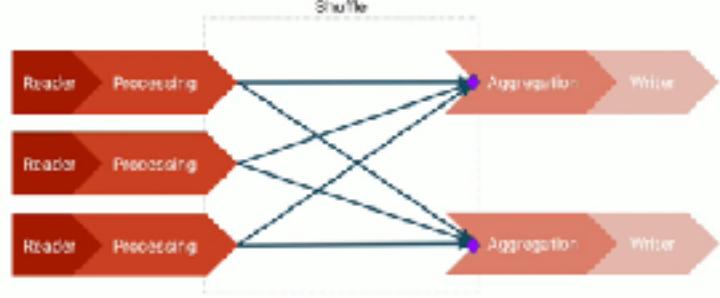
- Reader**: The first stage where data is read.
- Processing**: The second stage where data is processed.
- Shuffle**: A stage where data is shuffled between partitions. It is positioned between the Processing stage and the final stage.
- Aggregation**: The third stage where data is aggregated.
- Writer**: The final stage where data is written.

The process starts with three parallel Readers, each followed by a Processing stage. The output of the Processing stage feeds into the Shuffle stage. From the Shuffle stage, data is distributed to three parallel Aggregation stages, which then feed into a final Writer stage. The entire process is labeled "ID: 1EAS".

SPARK+AI SUMMIT 2018
ORGANIZED BY  databricks

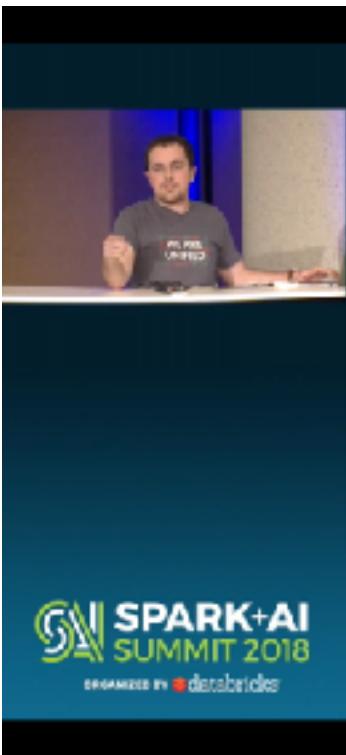


Checkpointing - Detailed

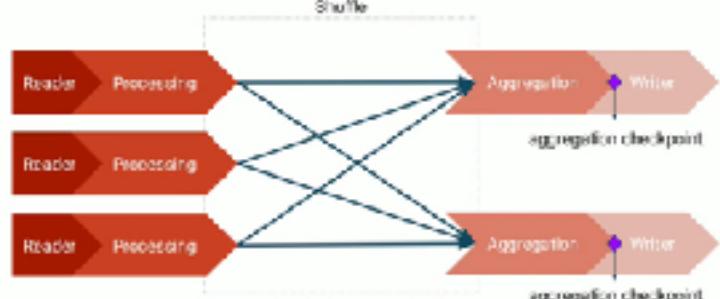


The diagram illustrates a three-stage pipeline: Reader → Processing → Shuffle → Aggregation → Writer. The Reader and Processing stages are shown in red, while the Shuffle, Aggregation, and Writer stages are shown in orange. The Pipeline section is labeled "IDEAS".

SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks



Checkpointing - Detailed



The diagram illustrates a three-stage pipeline: Reader → Processing → Shuffle → Aggregation → Writer. The Reader and Processing stages are shown in red, while the Shuffle, Aggregation, and Writer stages are shown in orange. Two purple dots labeled "aggregation checkpoint" are placed on the Aggregation stage. The Pipeline section is labeled "IDEAS".

SPARK+AI
SUMMIT 2018
ORGANIZED BY databricks

A video frame from the SPARK+AI SUMMIT 2018. On the left, a man in a grey t-shirt with "DATA UNIVERSITY" is speaking at a podium. On the right, a presentation slide titled "Checkpointing - Detailed" is displayed. The slide shows a flow diagram of a data processing pipeline. It starts with three "Reader Processing" boxes on the left, which feed into a central "Shuffle" area. From the shuffle, data flows into two parallel "Aggregation" and "Writer" boxes. A "Driver" box, labeled "ready for global commit", is positioned above the writers. Arrows indicate the flow from readers to shuffle, from shuffle to aggregation, and from aggregation to writer. Labels "commit partition writer" point to the connection between the shuffle and each writer. The slide has a dark background with a teal footer bar containing the SPARK+AI SUMMIT 2018 logo and "ORGANIZED BY databricks".

A video frame from the SPARK+AI SUMMIT 2018. On the left, the same man in the grey t-shirt is speaking. On the right, a presentation slide titled "Continuous Processing API" is shown. It contains a bulleted list:

- It's just Structured Streaming
- Run the same queries in continuous mode
- Only difference is a continuous trigger

The slide has a dark background with a teal footer bar containing the SPARK+AI SUMMIT 2018 logo and "ORGANIZED BY databricks".



Continuous Processing in 2.3

- Initial experimental release
- Supports ETL use cases
- Limited data source connectors

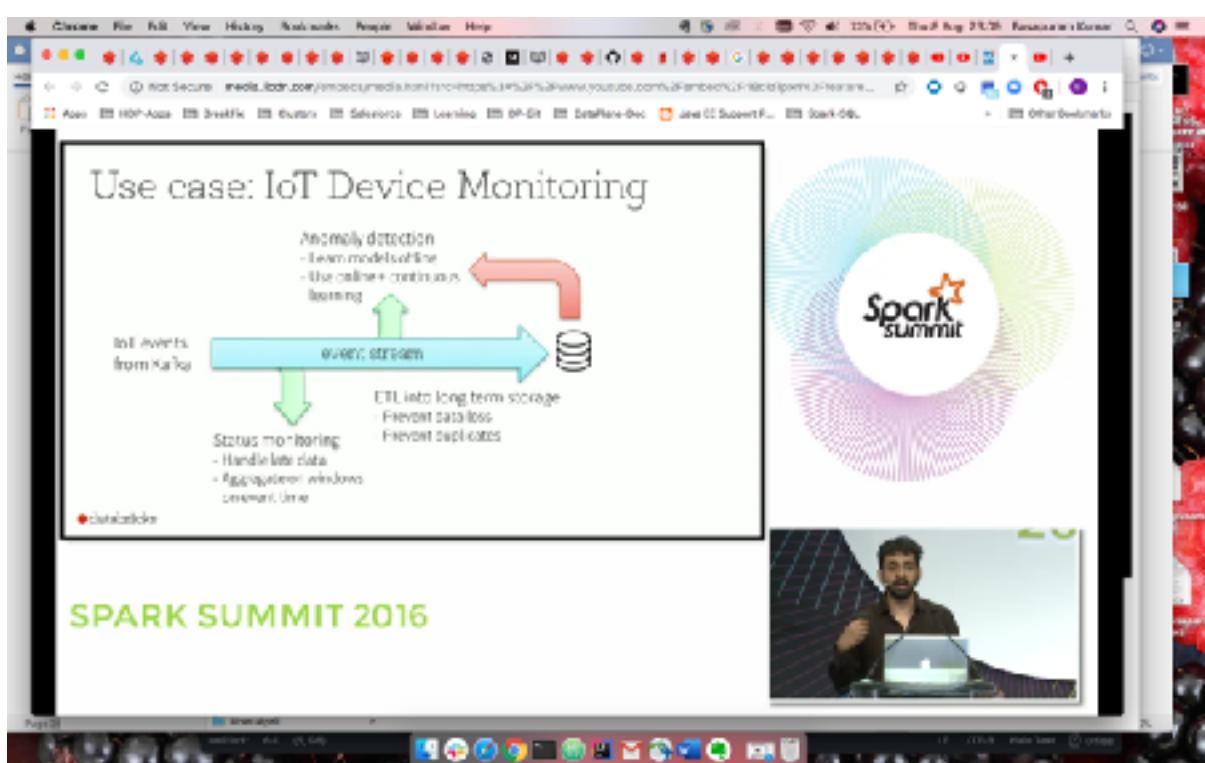
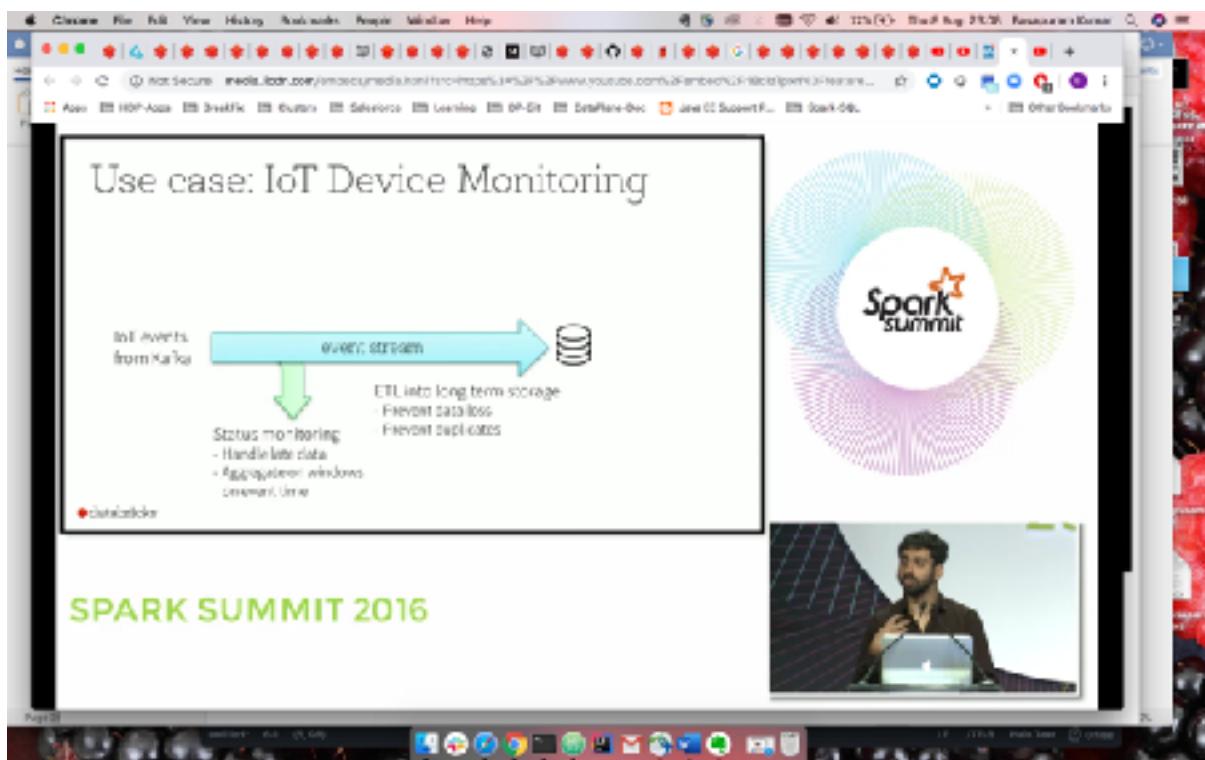
 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks



Ongoing And Future Work

- Shuffles (SPARK-24036)
- Event time (SPARK-24459)
- Metrics (SPARK-23887)
- Exactly-once semantics mode (SPARK-24460)
- Additional data sources (TBD)

 SPARK+AI
SUMMIT 2018
ORGANIZED BY  databricks



Use case: IoT Device Monitoring

The diagram illustrates the flow of data from Kafka to a central system and its subsequent processing. It shows three main paths:

- Anomaly detection:** Involves learning models online and performing continuous learning. This path has a red arrow pointing from the central system back to the Kafka input.
- Status monitoring:** Handles late data and uses adaptive windows (current, time). This path has a green arrow pointing from the central system down to the Kafka input.
- ETL into long term storage:** Prevents data loss and duplicates. This path has a red arrow pointing from the central system down to a database icon.

Finally, a green arrow points from the database icon up to the central system, labeled "Interactive debugging issues - consistency".

SPARK SUMMIT 2016

A video player window shows a man speaking at a podium during a presentation at the Spark Summit 2016.

Use case: IoT Device Monitoring

The diagram illustrates the flow of data from Kafka to a central system and its subsequent processing. It shows three main paths:

- Anomaly detection:** Involves learning models online and performing continuous learning. This path has a red arrow pointing from the central system back to the Kafka input.
- Status monitoring:** Handles late data and uses adaptive windows (current, time). This path has a green arrow pointing from the central system down to the Kafka input.
- ETL into long term storage:** Prevents data loss and duplicates. This path has a red arrow pointing from the central system down to a database icon.

Finally, a green arrow points from the database icon up to the central system, labeled "Interactive debugging issues - consistency".

Continuous Applications

Not just streaming any more

SPARK SUMMIT 2016

A video player window shows a man speaking at a podium during a presentation at the Spark Summit 2016.

Pain points with DStreams

1. Processing with event-time, dealing with late data
 - DStream API exposes batch time, hard to incorporate event-time

● danielnispel

SPARK SUMMIT 2016



What can you do with this that's hard with other engines?

True unification
Same code – same super-optimized engine for everything

Flexible API tightly integrated with the engine
Choose your own tool - Dataset/DataFrame/SQL
Greater debuggability and performance

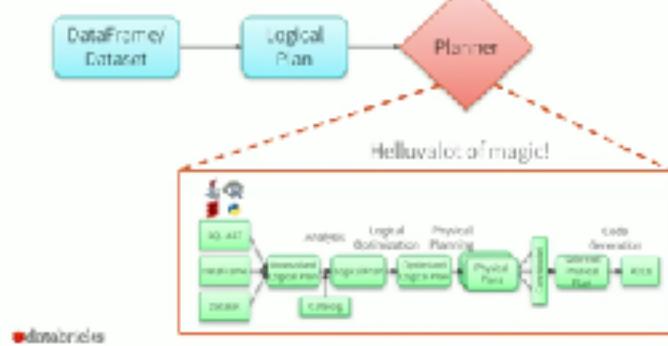
Benefits of Spark
in-memory computing, elastic scaling, fault-tolerance, straggler mitigation, ...

● danielnispel

SPARK SUMMIT 2016

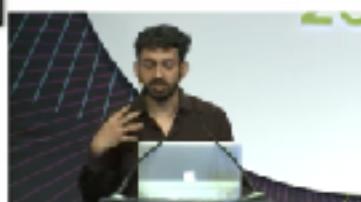
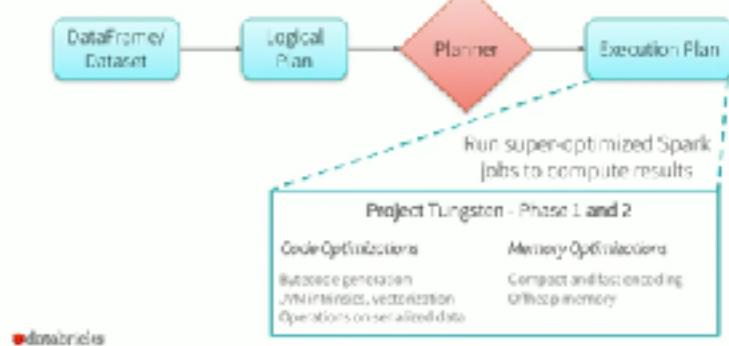


Batch Execution on Spark SQL



SPARK SUMMIT 2016

Batch Execution on Spark SQL



SPARK SUMMIT 2016

Continuous Incremental Execution

```

graph LR
    A[DataFrame/Dataset] --> B[Logical Plan]
    B --> C{Planner}
    C --> D[Incremental Execution Plan 1]
    C --> E[Incremental Execution Plan 2]
    C --> F[Incremental Execution Plan 3]
    C --> G[Incremental Execution Plan 4]
  
```

Planner knows how to convert streaming logical plans to a continuous series of incremental execution plans, for each processing the next chunk of streaming data.

SPARK SUMMIT 2016

Spark Delta lake

[Delta Lake](#) is an [open source storage layer](#) that brings reliability to data lakes.

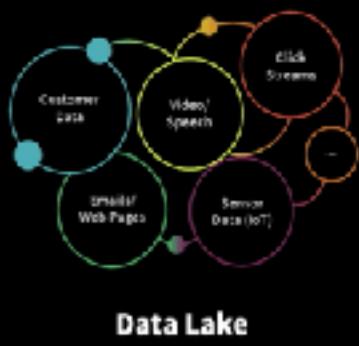
Delta Lake provides

1. ACID transactions,
2. Scalable metadata handling, and
3. Unifies streaming and batch data processing.
4. Delta Lake runs on top of your existing data lake and is fully compatible with Apache Spark APIs.

Challenges faced by current systems:

But the data is not ready for data science & ML

The **majority** of these projects are failing due to
unreliable data!



Data Science & ML



- Recommendation Engines
- Risk, Fraud Detection
- IoT & Predictive Maintenance
- Genomics & DNA Sequencing

©databricks

Data reliability challenges with data lakes



Failed production jobs leave data in corrupt state requiring tedious recovery



Lack of schema enforcement creates inconsistent and low quality data



Lack of consistency makes it almost impossible to mix appends and reads, batch and streaming

©databricks

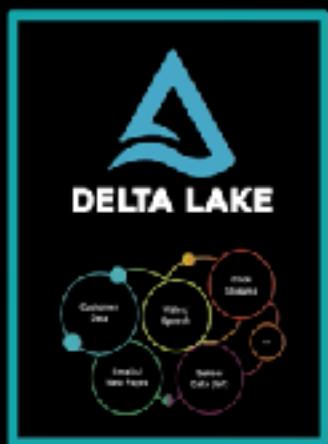
A New Standard for Building Data Lakes



Open Format Based on Parquet
With Transactions
Apache Spark API's

databricks

Delta Lake: makes data ready for Analytics



Reliability →
Performance

Data Science & ML



- Recommendation Engines
- Risk, Fraud Detection
- IoT & Predictive Maintenance
- Genomics & DNA Sequencing

databricks

Delta Lake ensures data reliability



Key Features

- ACID Transactions
- Schema Enforcement
- Unified Batch & Streaming
- Time Travel/Data Snapshots

© databricks

The Delta Architecture



© databricks

The Delta Architecture

kafka → **Spark** → **databricks DELTA** → **Spark**

AWS DATA LAKE

Streaming Analytics

Reporting

The SCALE of data lake

The RELIABILITY & PERFORMANCE of data warehouse

The LOW-LATENCY of streaming

SPARK SUMMIT
powered by **databricks**

databricks