

IEMS5730/ IERG4330/ESTR4316

Spring 2022



Stream Processing

Prof. Wing C. Lau

Department of Information Engineering

wclau@ie.cuhk.edu.hk

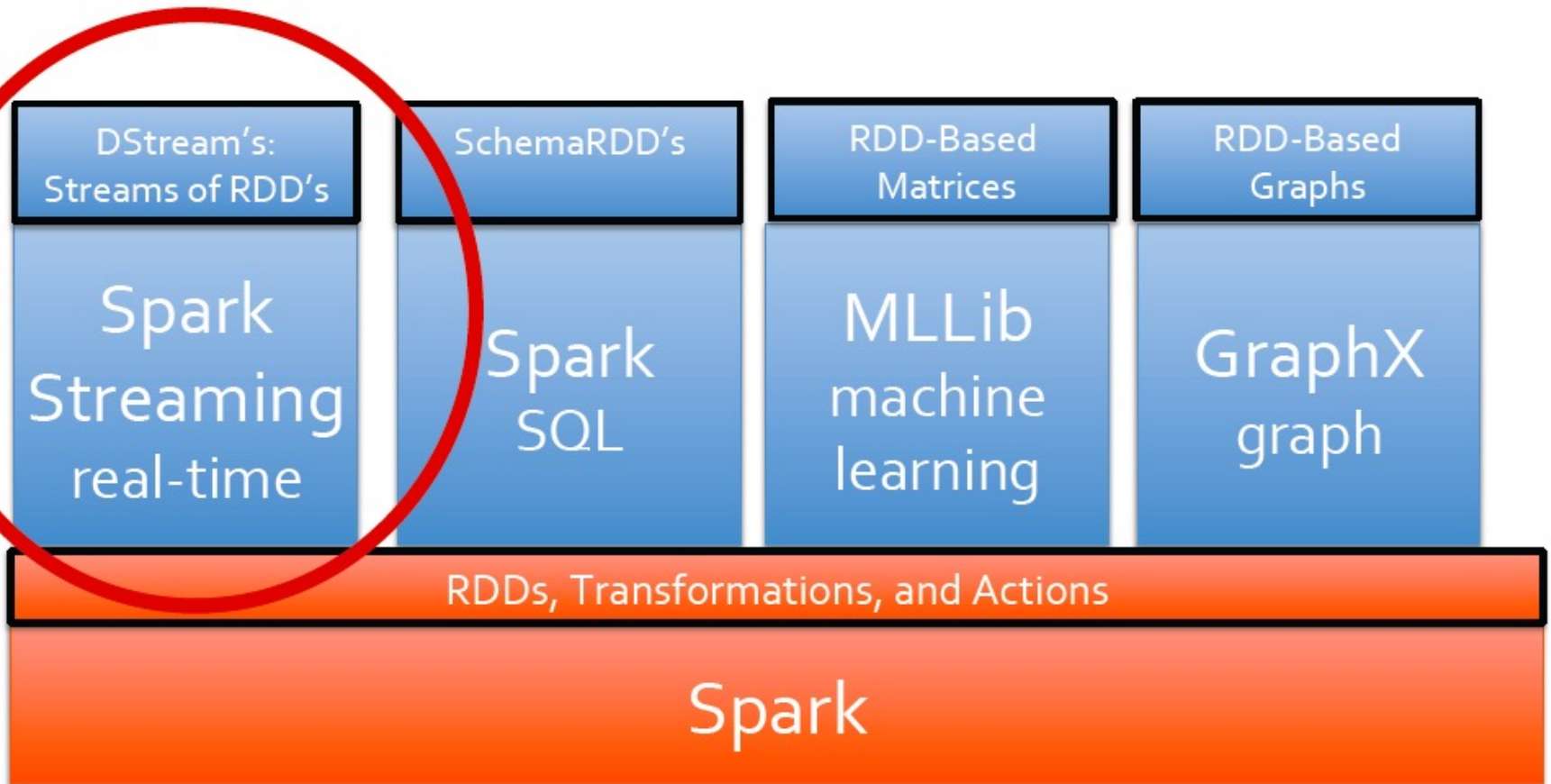
Acknowledgements

■ These slides are adapted from the following sources:

- Matei Zaharia, “Spark 2.0,” Spark Summit East Keynote, Feb 2016.
- Reynold Xin, “The Future of Real-Time in Spark,” Spark Summit East Keynote, Feb 2016.
- Michael Armbrust, “Structuring Spark: SQL, DataFrames, DataSets, and Streaming,” Spark Summit East Keynote, Feb 2016.
- Ankur Dave, “GraphFrames: Graph Queries in Spark SQL,” Spark Summit East, Feb 2016.
- Michael Armbrust, “Spark DataFrames: Simple and Fast Analytics on Structured Data,” Spark Summit Amsterdam, Oct 2015.
- Michael Armbrust et al, “Spark SQL: Relational Data Processing in Spark,” SIGMOD 2015.
- Michael Armbrust, “Spark SQL Deep Dive,” Melbourne Spark Meetup, June 2015.
- Reynold Xin, “Spark,” Stanford CS347 Guest Lecture, May 2015.
- Joseph K. Bradley, “Apache Spark MLlib’s past trajectory and new directions,” Spark Summit Jun 2017.
- Joseph K. Bradley, “Distributed ML in Apache Spark,” NYC Spark MeetUp, June 2016.
- Ankur Dave, “GraphFrames: Graph Queries in Apache Spark SQL,” Spark Summit, June 2016.
- Joseph K. Bradley, “GraphFrames: DataFrame-based graphs for Apache Spark,” NYC Spark MeetUp, April 2016.
- Joseph K. Bradley, “Practical Machine Learning Pipelines with MLlib,” Spark Summit East, March 2015.
- Joseph K. Bradley, “Spark DataFrames and ML Pipelines,” MLconf Seattle, May 2015.
- Ameet Talwalkar, “MLlib: Spark’s Machine Learning Library,” AMP Camps 5, Nov. 2014.
- Shivaram Venkataraman, Zongheng Yang, “SparkR: Enabling Interactive Data Science at Scale,” AMP Camps 5, Nov. 2014.
- Tathagata Das, “Spark Streaming: Large-scale near-real-time stream processing,” O’Reilly Strata Conference, 2013.
- Joseph Gonzalez et al, “GraphX: Graph Analytics on Spark,” AMPCAMP 3, 2013.
- Jules Damji, “Jumpstart on Apache Spark 2.X with Databricks,” Spark Sat. Meetup Workshop, Jul 2017.
- Sameer Agarwal, “What’s new in Apache Spark 2.3,” Spark+AI Summit, June 2018.
- Reynold Xin, Spark+AI Summit Europe, 2018.
- Hyukjin Kwon of Hortonworks, “What’s New in Spark 2.3 and Spark 2.4,” Oct 2018.
- Matei Zaharia, “MLflow: Accelerating the End-to-End ML Lifecycle,” Nov. 2018.
- Jules Damji, “MLflow: Platform for Complete Machine Learning Lifecycle,” PyData, Jan 2019.

■ All copyrights belong to the original authors of the materials.

Major Modules in Spark



Motivation for Spark Streaming

- Many Important Applications must process Large Data Streams at second-scale latencies
 - Site Statistics, Intrusion Detection, Online ML, Fraud Detection
- To build and scale these applications require:
 - Integration: with Offline Analytic Stack
 - Fault-tolerance: to handle Crashes and Stragglers
 - Efficiency: low cost beyond base processing
 - Work with distributed collections as you would with local ones

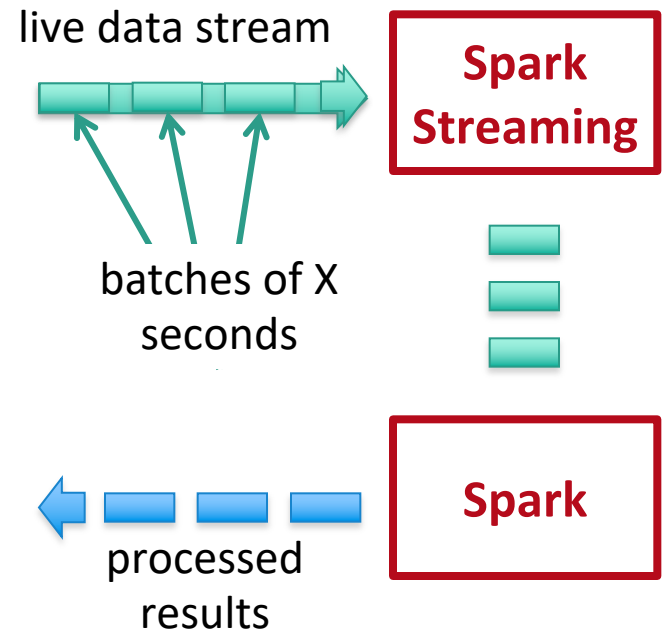
Spark Streaming Overview

- “Low Latency”, High-throughput and Fault Tolerant
- Discretized Stream (DStream): Micro-batches of RDDs
 - Operations are similar to RDD
 - Lineage for Fault-Tolerance
- Leverage Core Components from Spark
 - RDD data model and API
 - Data Partitioning and Shuffles
 - Task Scheduling
 - Monitoring/ Instrumentation
 - Scheduling and Resource Allocation
- Support Flume, Kafka, Twitter, Kinesis, etc for Data Ingestion
- Long-running Spark Applications

Discretized Stream Processing

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

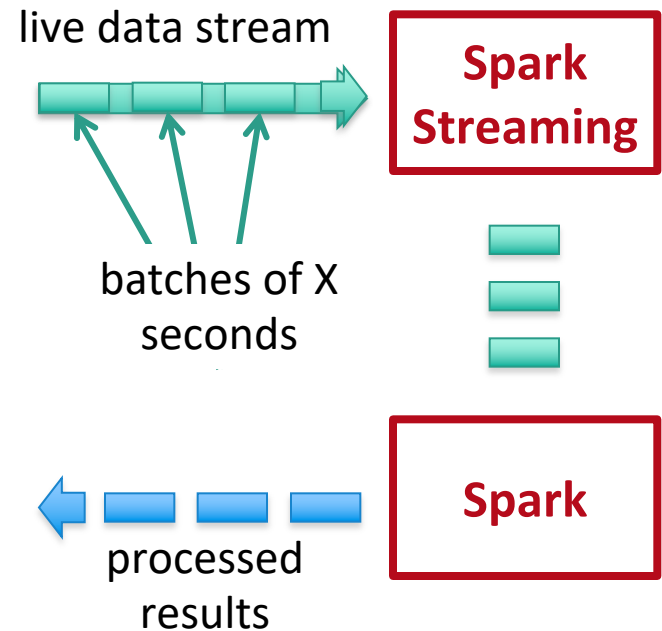
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



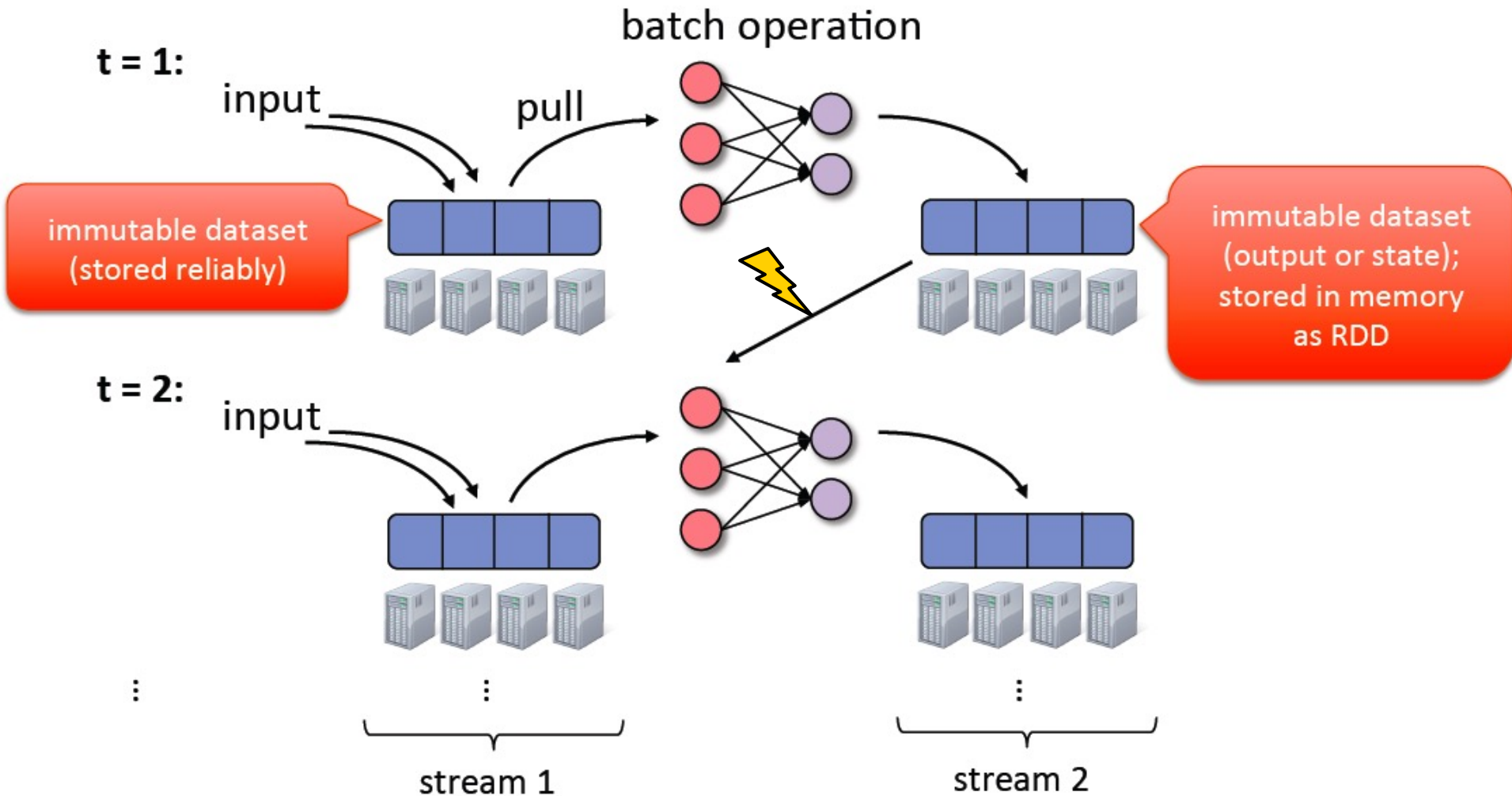
Discretized Stream Processing

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

- Batch sizes as low as $\frac{1}{2}$ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



Discretized Stream Processing (Micro-Batching)



Programming Interface

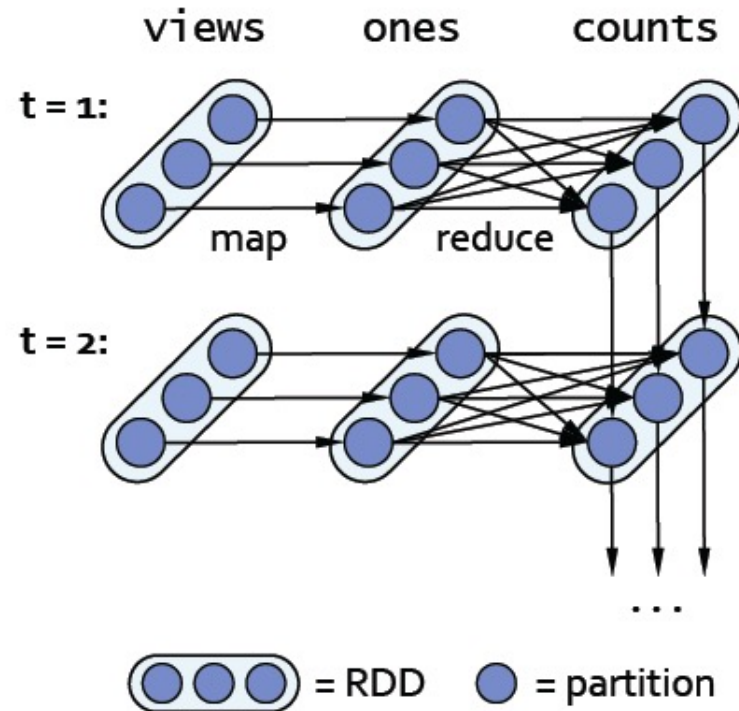
Simple functional API

```
views = readStream("http:...", "1s")
ones = views.map(ev => (ev.url, 1))
counts = ones.runningReduce(_ + _)
```

Interoperates with RDDs

```
// Join stream with static RDD
counts.join(historicCounts).map(...)
```

```
// Ad-hoc queries on stream state
counts.slice("21:00", "21:05").topK(10)
```



`runningReduce()` is merely a concept, actually not implemented by Spark ;
Use `updateStateByKey()`, `mapStateByKey()` etc instead ; more details on
arbitrary Stateful operations with Spark Streaming later.

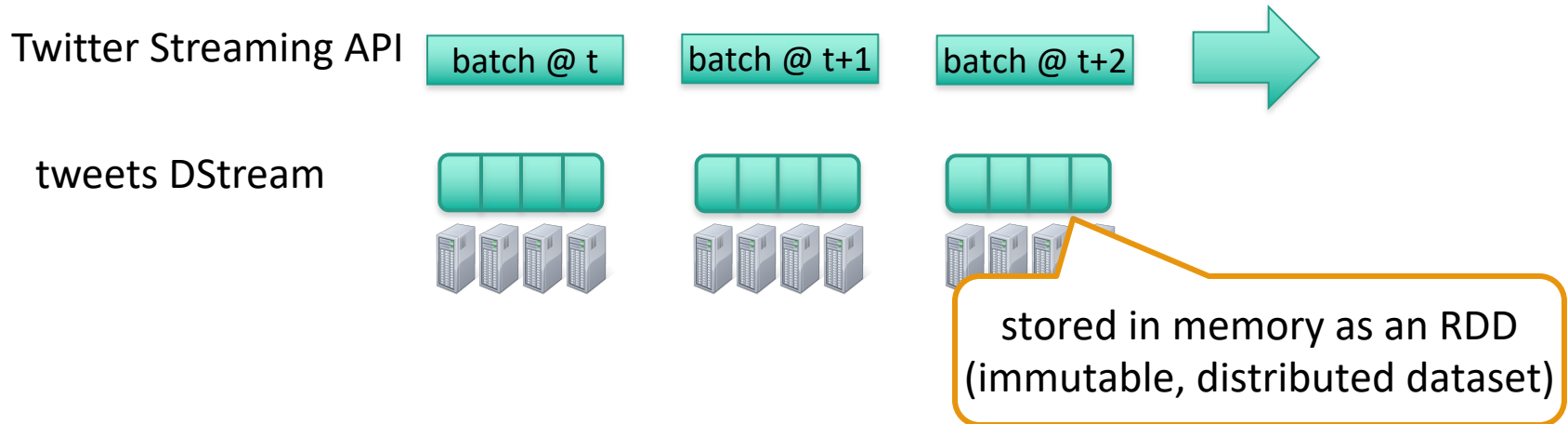
Spark Streaming API

- **Transformations** – modify data from one DStream to another
 - Standard RDD operations – map, filter, distinct, countByValue, reduceByKey, join, ...
 - Stateful, Sliding Window-based Operations
 - window, updateStateByKey, countByValueAndWindow
 - Window Size & Slide Interval
- **Output Operations** – send data to external entity
 - saveAsHadoopFiles – saves to HDFS
 - foreach – do anything with each batch of results
- Checkpointing
- Register DStream as a SQL table

Example 1: Get HashTags from Twitter

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

DStream: a sequence of distributed datasets (RDDs) representing a distributed stream of data



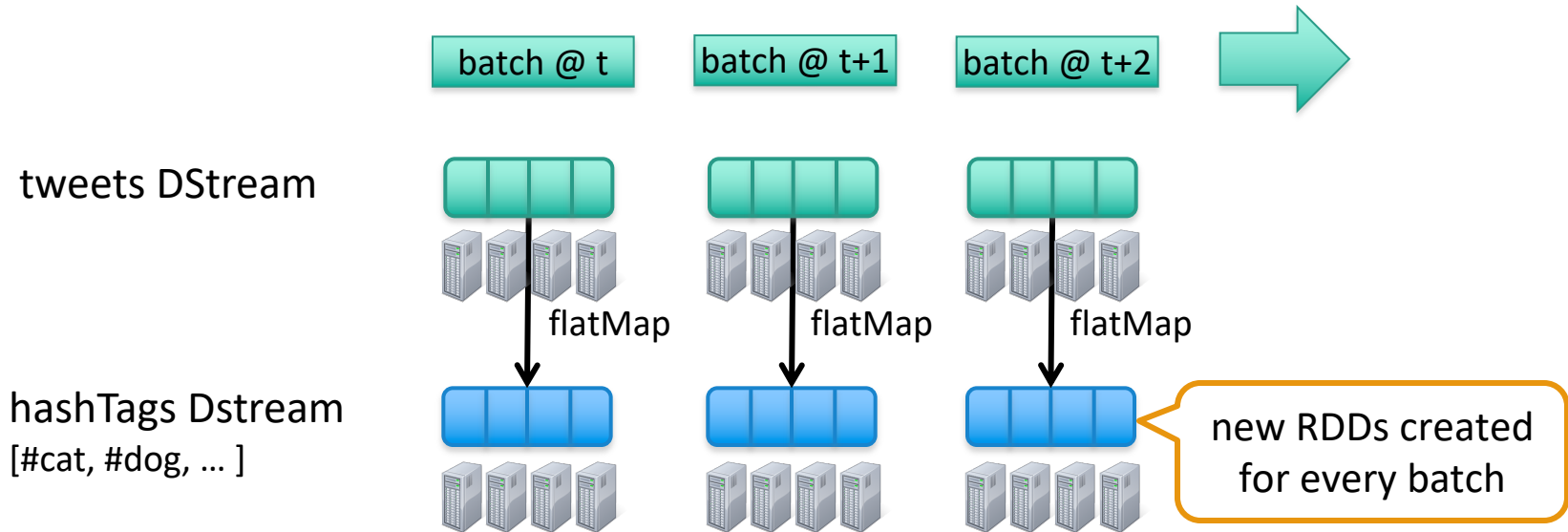
Example 1: Get HashTags from Twitter

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

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

new DStream

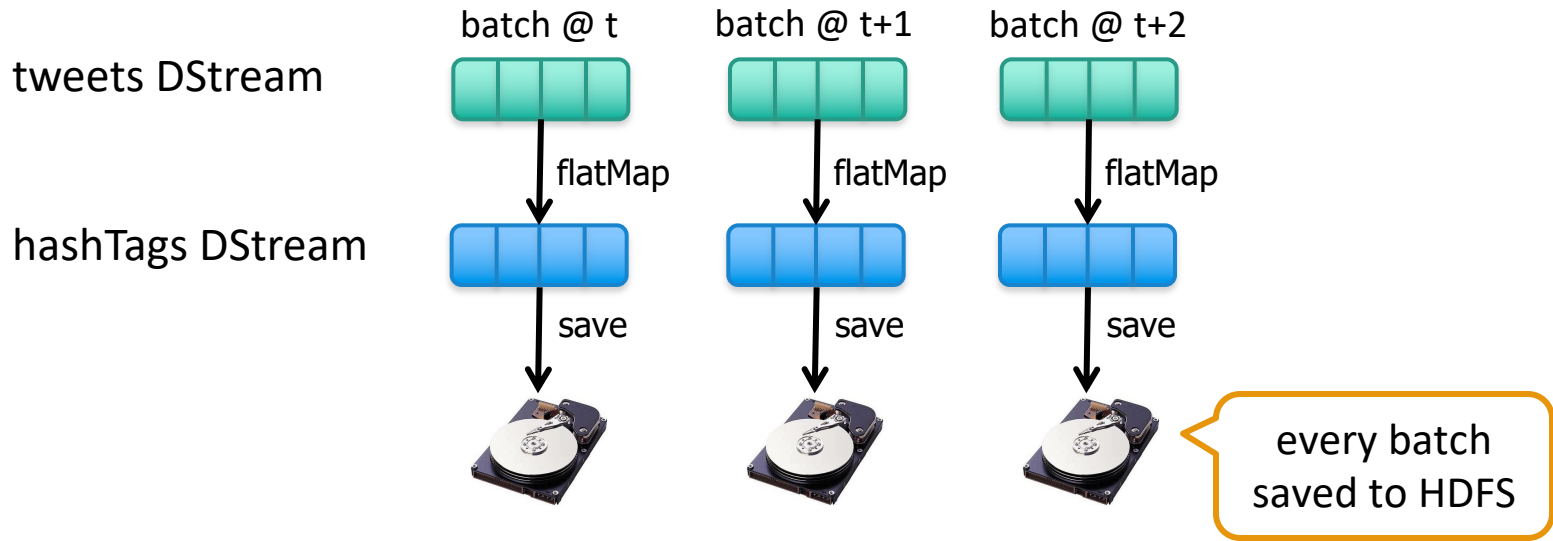
transformation: modify data in one DStream to create another DStream



Example 1: Get HashTags from Twitter

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

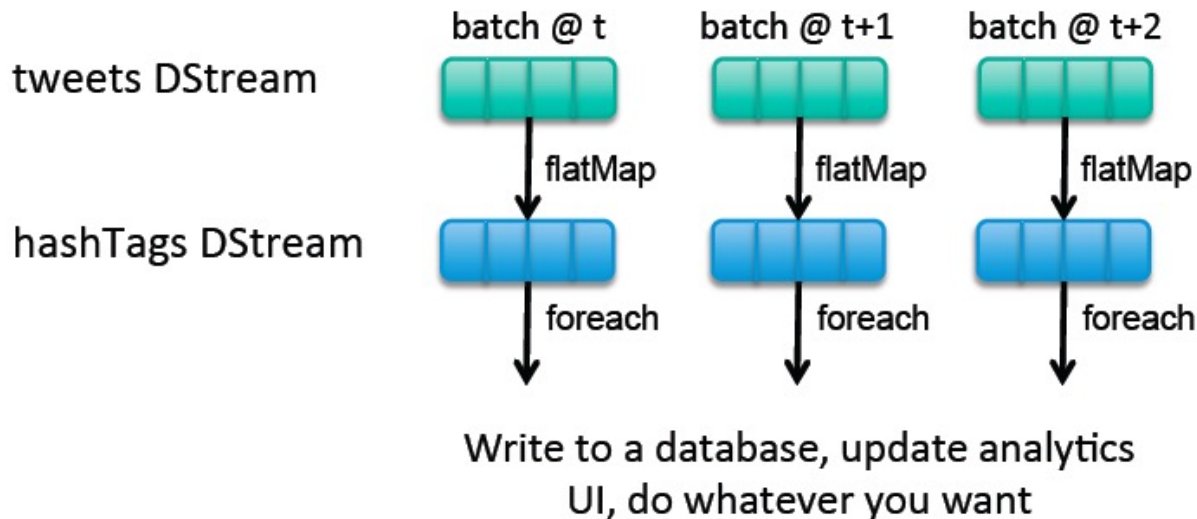
output operation: to push data to external storage



Example 1: Get HashTags from Twitter

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

foreach: do whatever you want with the processed data




Example 1 in Java vs. Scala

Scala

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

Java

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



Spark program vs Spark Streaming program

Spark Streaming program on Twitter stream

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

Spark program on Twitter log file

```
val tweets = sc.hadoopFile("hdfs://...")  
val hashTags = tweets.flatMap(status => getTags(status))  
hashTags.saveAsHadoopFile("hdfs://...")
```


Vision - one stack to rule them all

- Explore data interactively using Spark Shell / PySpark to identify problems
- Use same code in Spark stand-alone programs to identify problems in production logs
- Use similar code in Spark Streaming to identify problems in live log streams

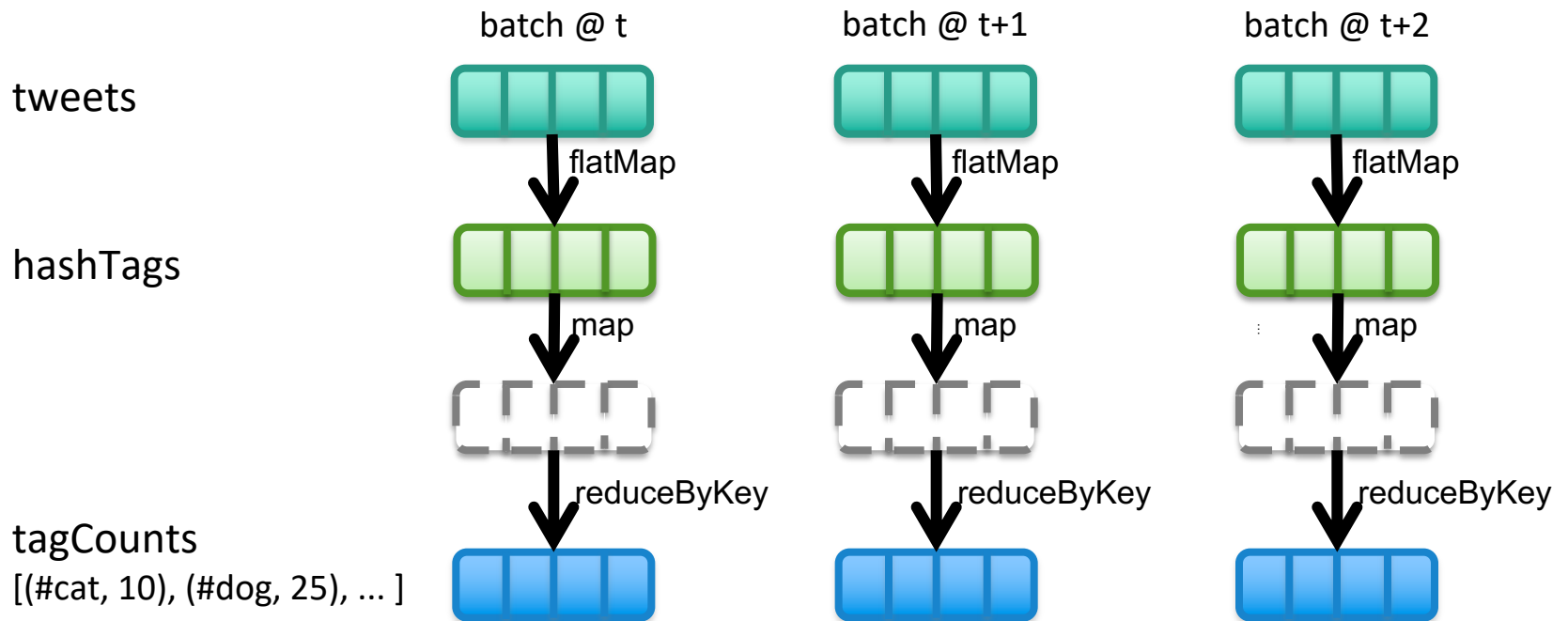
```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = file.map(...)
```

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

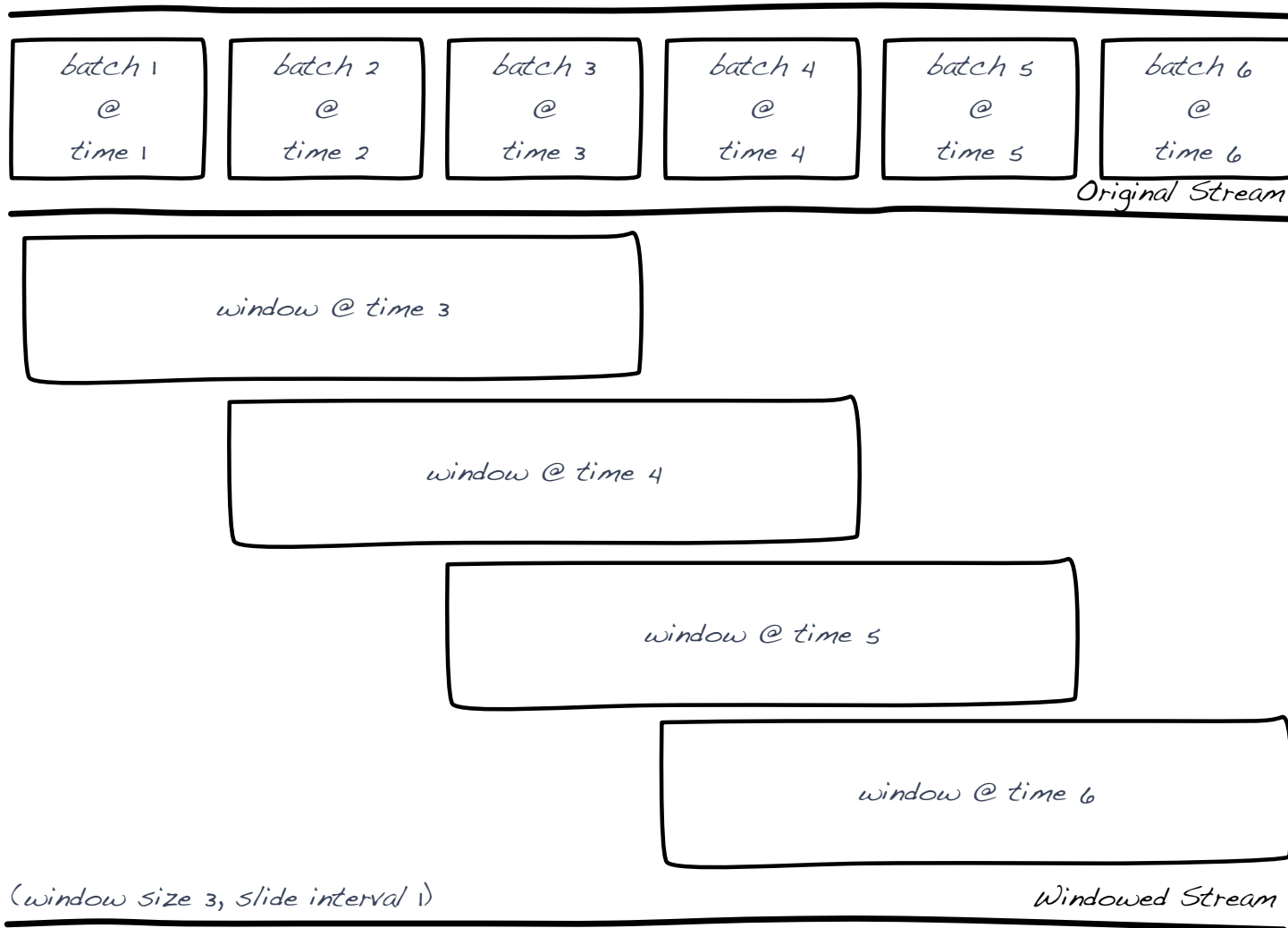
```
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered =
file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
    ...
  }
}
```

Example 2: Count the HashTags

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



Window-based Operations on DStreams



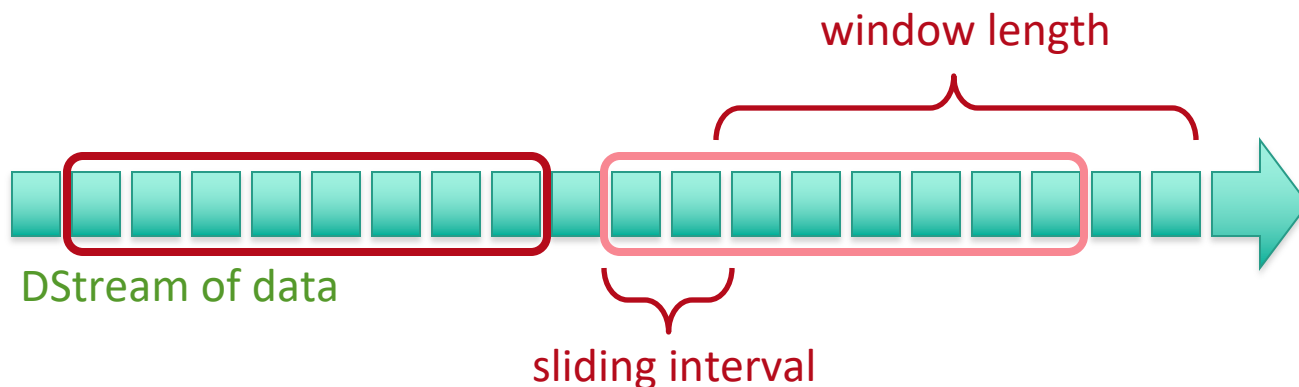
Example 3: Count the HashTags over last 1 min

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

sliding window
operation

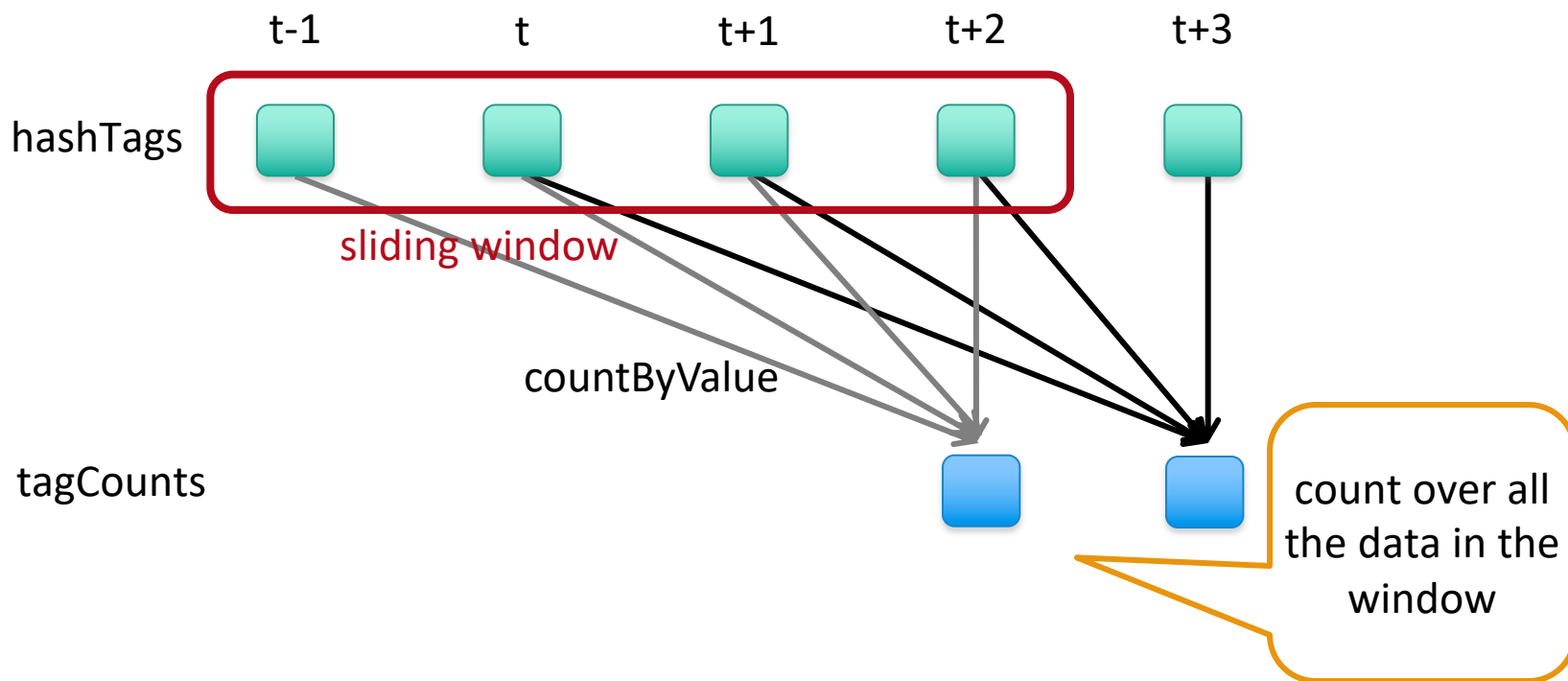
window length

sliding interval



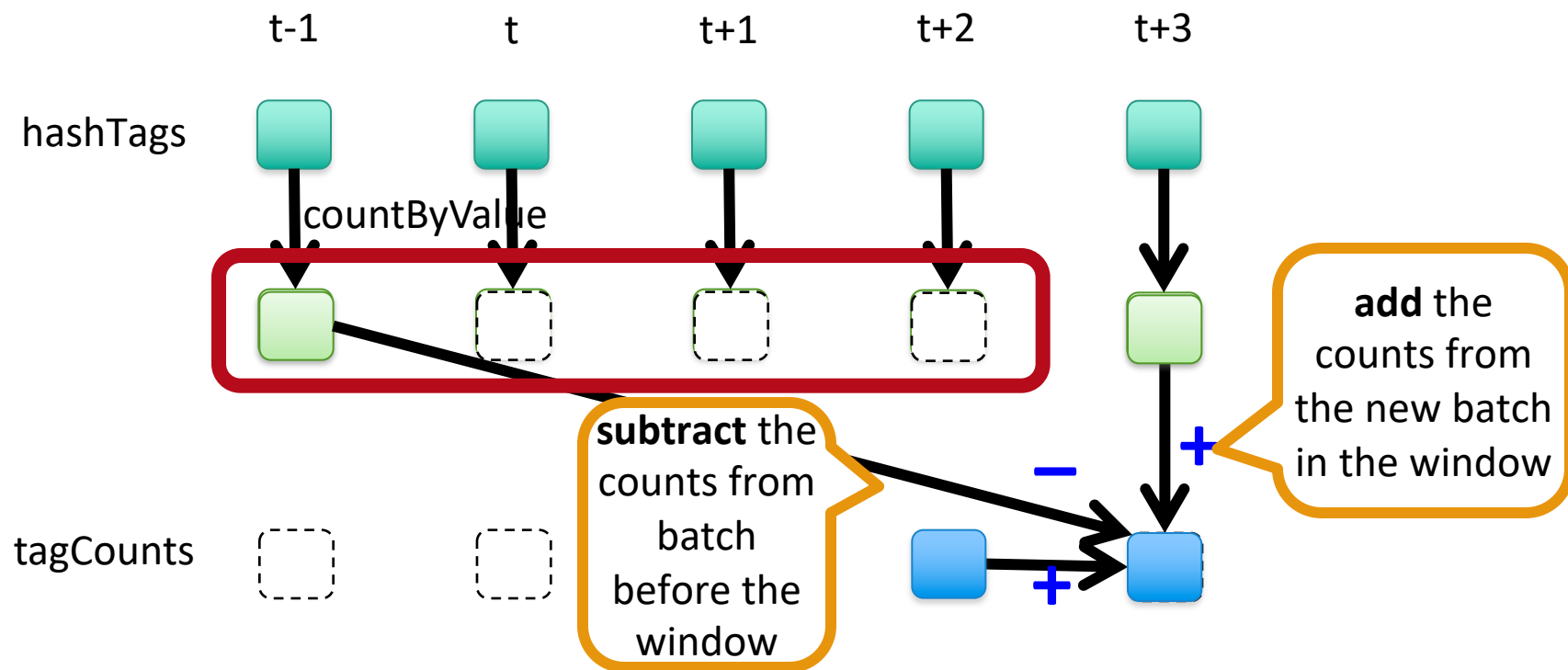
Example 3: Count the HashTags over last 1 min

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



Example 3: Smart Window-based countByValue

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



Smart window-based *reduce*

- Technique to incrementally compute count generalizes to many reduce operations
 - Need a function to “inverse reduce” (“subtract” for counting)

- Could have implemented counting as:

```
hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), ...)
```

Another Example: Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))
```

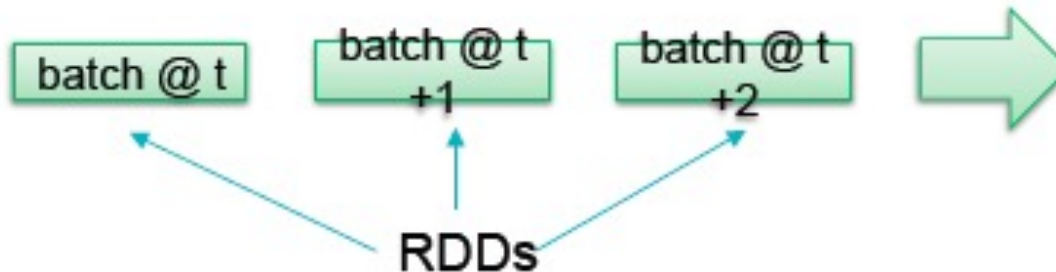
entry point of streaming functionality

```
val lines = KafkaUtils.createStream(context, ...)
```

create **DStream** from Kafka data

Discretized Stream (DStream) basic abstraction of Spark Streaming
series of RDDs representing a stream of data

lines DStream



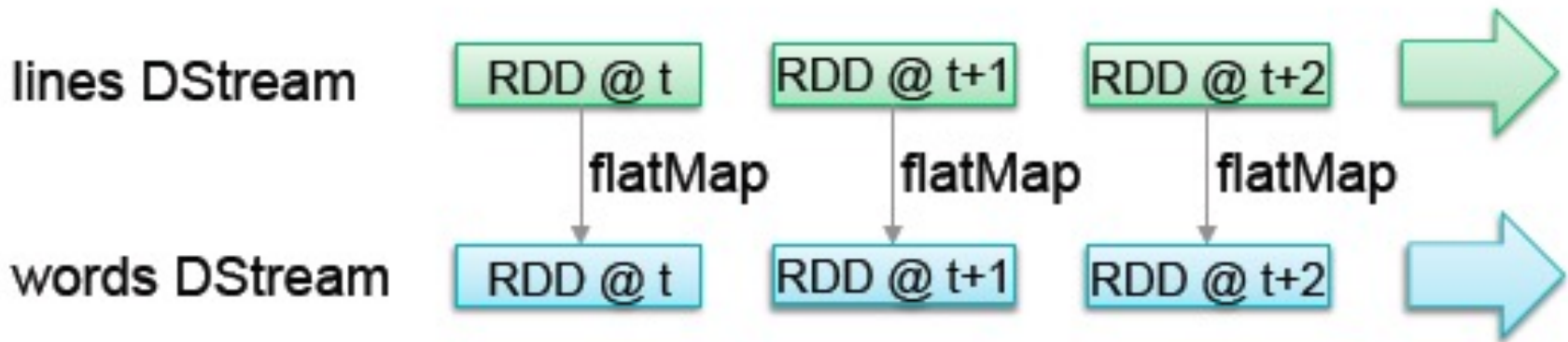
Another Example: Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))
```

```
val lines = KafkaUtils.createStream(context, ...)
```

```
val words = lines.flatMap(_.split(" "))
```

split lines into words



Another Example: Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))
```

```
val lines = KafkaUtils.createStream(context, ...)
```

```
val words = lines.flatMap(_.split(" "))
```

```
val wordCounts = words.map(x => (x, 1))
```

```
.reduceByKey(_ + _)
```

```
wordCounts.print()
```

```
context.start()
```

count the words

print some counts on
screen

start receiving and
transforming the data

Another Example: Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))
```

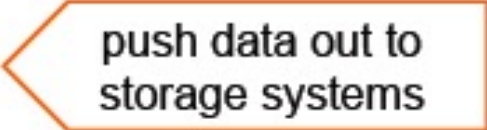
```
val lines = KafkaUtils.createStream(context, ...)
```

```
val words = lines.flatMap(_.split(" "))
```

```
val wordCounts = words.map(x => (x, 1))  
                    .reduceByKey(_ + _)
```

```
wordCounts.foreachRDD(rdd => /* do something */ )
```

```
context.start()
```



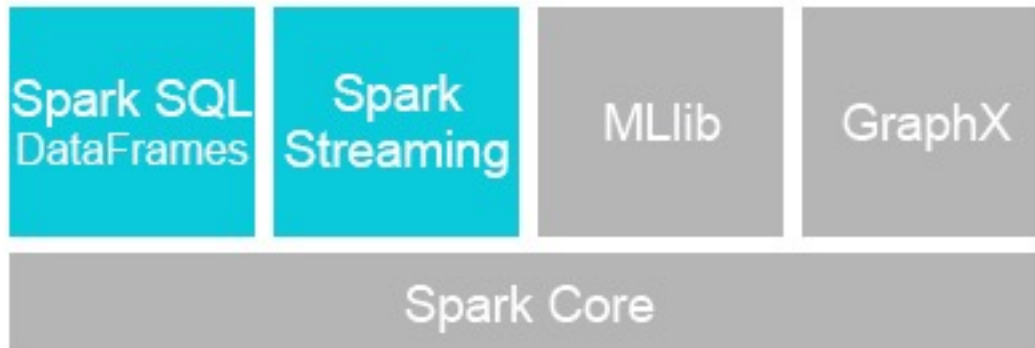
push data out to
storage systems

Combine SQL with Streaming

- Interactively query streaming data with SQL and Dataframes

```
// Register each batch in stream as table  
kafkaStream.foreachRDD { batchRDD =>  
    batchRDD.toDF.registerTempTable("events")  
}
```

```
// Interactively query table  
sqlContext.sql("select * from events")
```



Many Transformations

Window operations

```
words.map(x => (x, 1)).reduceByKeyAndWindow(_ + _, Minutes(1))
```

Arbitrary stateful processing

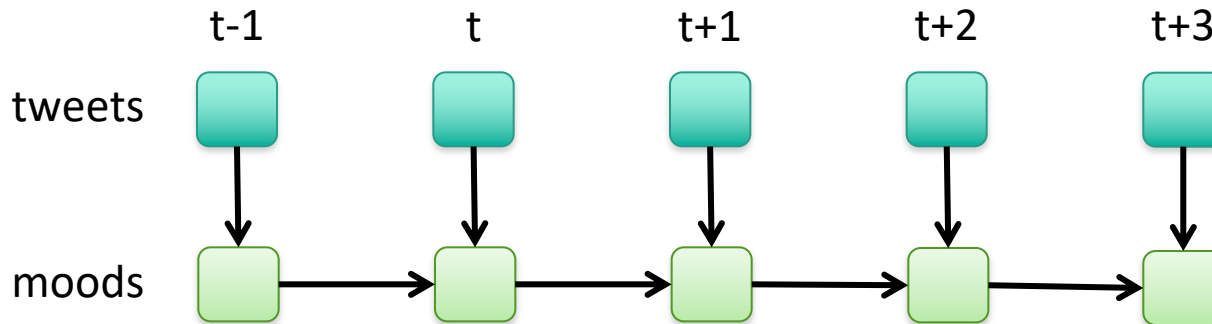
```
def stateUpdateFunc(newData, lastState) => updatedState  
  
val stateStream = keyValueDStream.updateStateByKey(stateUpdateFunc)
```

Arbitrary Stateful Computations

- Maintain arbitrary state, track sessions and specify function to generate new state based on previous state and new data:
 - e.g. Maintain per-user mood as state, and update it with his/her tweets

```
def updateMood(newTweets, lastMood) => newMood
```

```
val moods = tweets.updateStateByKey(tweet => updateMood( _ ))
```



Detail usage example of updateStateByKey at:

https://docs.cloud.databricks.com/docs/latest/databricks_guide/07%20Spark%20Streaming/11%20Global%20Aggregations%20-%20updateStateByKey.html

for the purpose of Historical Reference only !!

New Developments in Spark-Streaming with Arbitrary Stateful Computations

Instead of using `updateStateByKey()`:

- `MapWithState()` was introduced in Spark 1.6 as the preferred way to realize stateful operations in Spark Streaming,
- `MapGroupsWithState` and `FlatMapGroupsWithState` were introduced in Spark 2.2

See detail usage examples at:

<https://databricks.com/blog/2016/02/01/faster-stateful-stream-processing-in-apache-spark-streaming.html>

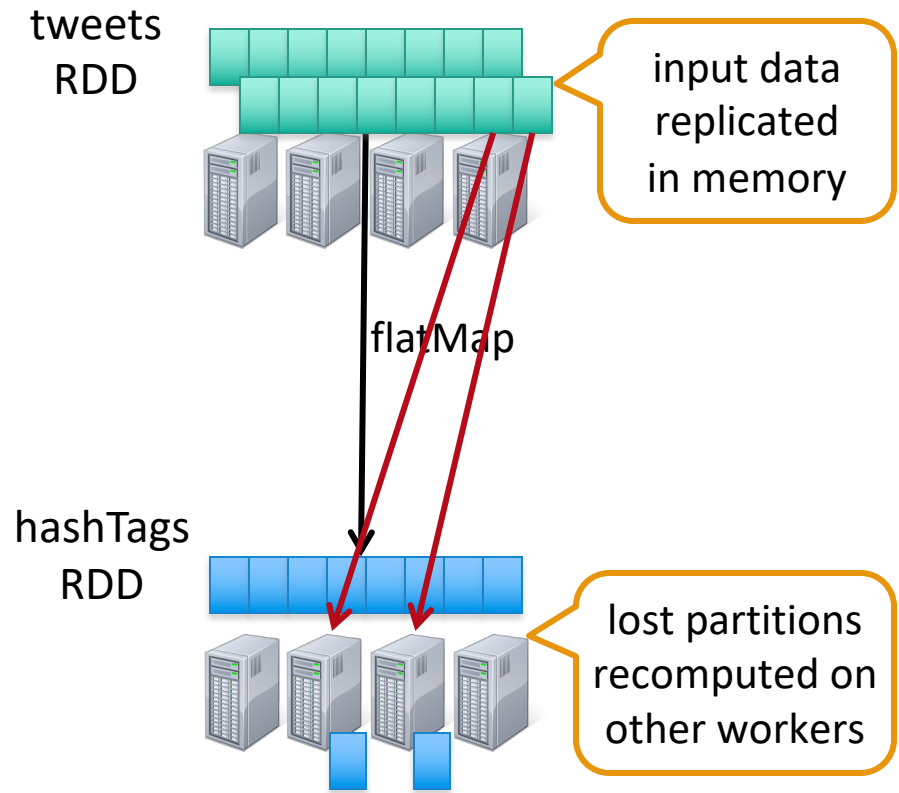
<http://asyncified.io/2016/07/31/exploring-stateful-streaming-with-apache-spark/>

<http://asyncified.io/2017/07/30/exploring-stateful-streaming-with-spark-structured-streaming/>

<https://databricks.com/blog/2017/10/17/arbitrary-stateful-processing-in-apache-sparks-structured-streaming.html>

Fault Tolerance

- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- Therefore, all transformed data is fault-tolerant
- Exactly once semantics
 - No double counting



Input Sources

- Out of the box support for:
 - Kafka, Flume, Akka Actors, Raw TCP sockets, HDFS, etc
- Developers can write additional custom *receiver(s)*
 - Just define what to and when receiver is started and stopped
- Can also generate one's own sequence of RDDs and push them in as a “Stream”

Zero (Input) Data Loss during Streaming

For Non-replayable Sources, i.e. sources that do not support replay from any position (e.g. Flume, etc):

- Solved using Write Ahead Log (WAL) (since Spark 1.3)

For Replayable Sources, i.e. sources that allow data to be replayed from any position (e.g. Kafka, Kinesis, etc):

- Solved with more reliable Kafka and Kinesis Integrations (Spark 1.3-1.5)

Write Ahead Log (WAL) [since Spark 1.3]

- All received data synchronously written to HDFS and replayed when necessary after failure
- WAL can be enabled by setting Spark configuration flag:
`spark.streaming.receiver.writeAheadLog.enabled` to `TRUE`
- Can give end-to-end at least once guarantee for sources that can support acks, but do not support replays

Reliable Kinesis [since Spark 1.5]

- Save record sequence numbers instead of data to WAL
- Replay from Kinesis using sequence numbers
- Higher throughput than using WAL
- Can give at least once guarantee

Reliable Kafka [Spark 1.3, graduated in 1.5]

- New API: **Direct** Kafka stream:
 - Does not use receivers, does not use ZooKeeper to save offsets
 - Offset management (saving, replaying) by Spark Streaming
- Can provide up to 10x higher throughput than earlier receiver
 - <https://spark-summit.org/2015/events/towards-benchmarking-modern-distributed-streaming-systems/>
- Can give exactly-once guarantee (excluding o/p to storage)
- Can run Spark batch jobs directly on Kafka
 - # of RDD partitions = # of Kafka partitions, easy to reason about
 - <https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html>

Innerworkings of a Discretized Stream (DStream)

A sequence of RDDs representing
a stream of data

What does it take to define a DStream?

DStream Interface

The DStream interface primarily defines how to generate an RDD in each batch interval

- List of *dependent* (parent) DStreams
- *Slide Interval*, the interval at which it will compute RDDs
- Function to *compute* RDD at a time *t*

Example: Mapped DStream

- *Dependencies:* Single parent DStream
- *Slide Interval:* Same as the parent DStream
- *Compute function for time t:* Create new RDD by applying map function on parent DStream's RDD of time t

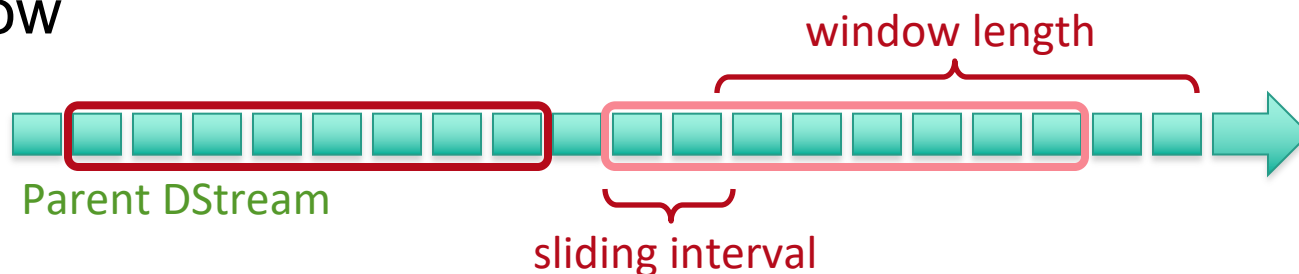
```
override def compute(time: Time): Option[RDD[U]] = {  
    parent.getOrCompute(time).map(_.map[U](mapFunc))  
}
```

Gets RDD of time t if already
computed once, or generates it

Map function applied to
generate new RDD

Example: Windowed DStream

Window operation gather together data over a sliding window



Dependencies: Single parent DStream

Slide Interval: Window sliding interval

Compute function for time t : Apply union over all the RDDs of parent DStream between times t and $(t - \text{window length})$

Example: Network Input DStream

Base class of all input DStreams that receive data from the network

- *Dependencies:* None
- *Slide Interval:* Batch duration in streaming context
- *Compute function for time t :* Create a BlockRDD with all the blocks of data received in the last batch interval
- Associated with a Network Receiver object

Network Receiver

Responsible for receiving data and pushing it into Spark's data management layer (Block Manager)

Base class for all receivers - Kafka, Flume, etc.

Simple Interface:

- What to do *on starting* the receiver
 - Helper object *blockGenerator* to push data into Spark
- What to do *on stopping* the receiver

Example: Socket Receiver

- *On start:*

 - Connect to remote TCP server

 - While socket is connected,

 - Receiving bytes and deserialize

 - Deserialize them into Java objects

 - Add the objects to *blockGenerator*

- *On stop:*

 - Disconnect socket

Other functions in DStream interface

- *parentRememberDuration* – defines how long should
 - Window-based DStreams have *parentRememberDuration = window length*
- *mustCheckpoint* – if set to true, the system will automatically enable periodic checkpointing
 - Set to true for stateful DStreams

DStream Graph

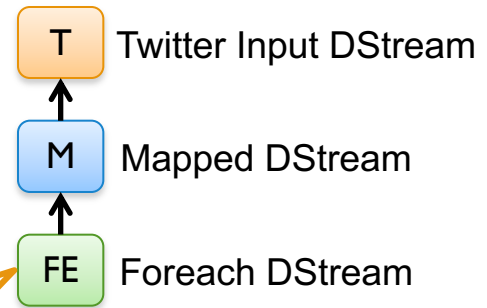
Spark Streaming program

```
t = ssc.twitterStream(...)
    .map(...)
t.foreach(...)
```



Dummy DStream signifying an output operation

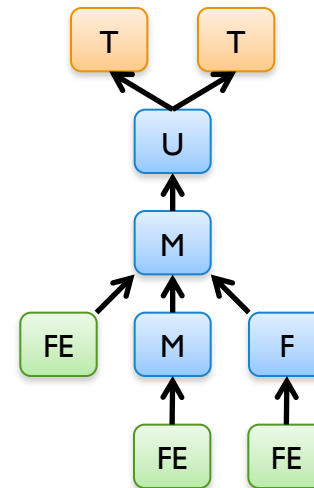
DStream Graph



```
t1 = ssc.twitterStream(...)
t2 = ssc.twitterStream(...)

t = t1.union(t2).map(...)

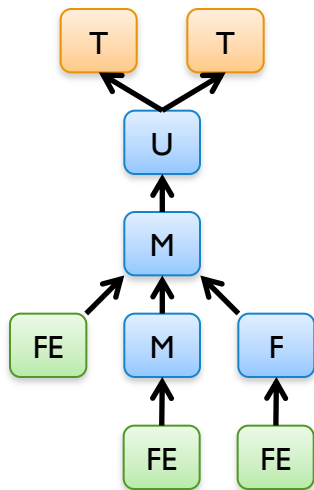
t.saveAsHadoopFiles(...)
t.map(...).foreach(...)
t.filter(...).foreach(...)
```



DStream Graph \rightarrow RDD Graphs \rightarrow Spark jobs

- Every interval, RDD graph is computed from DStream graph
- For each output operation, a Spark action is created
- For each action, a Spark job is created to compute it

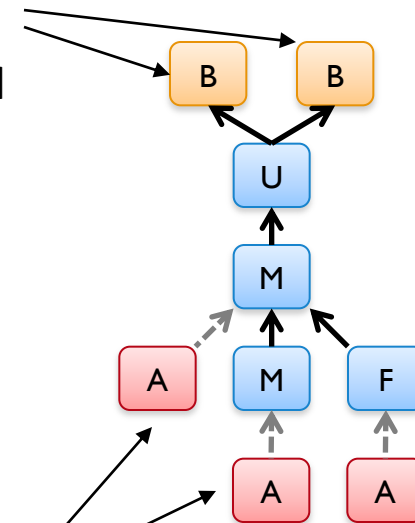
DStream Graph



Block RDDs with data received in last batch interval



RDD Graph

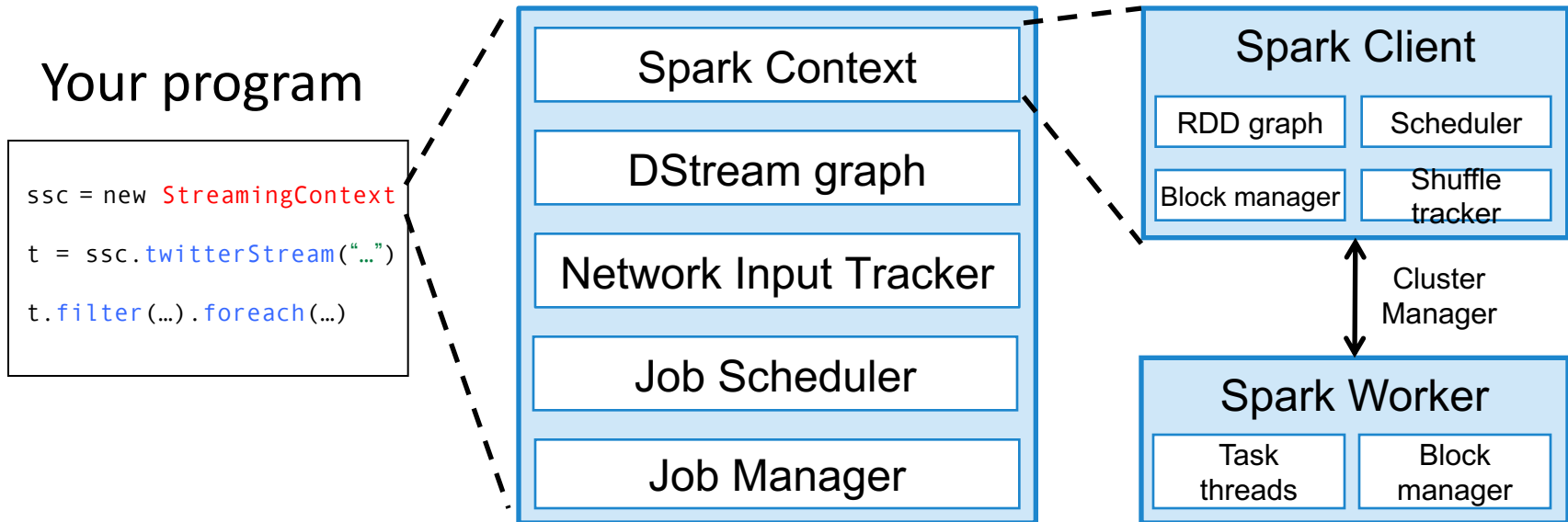


3 Spark jobs

Agenda

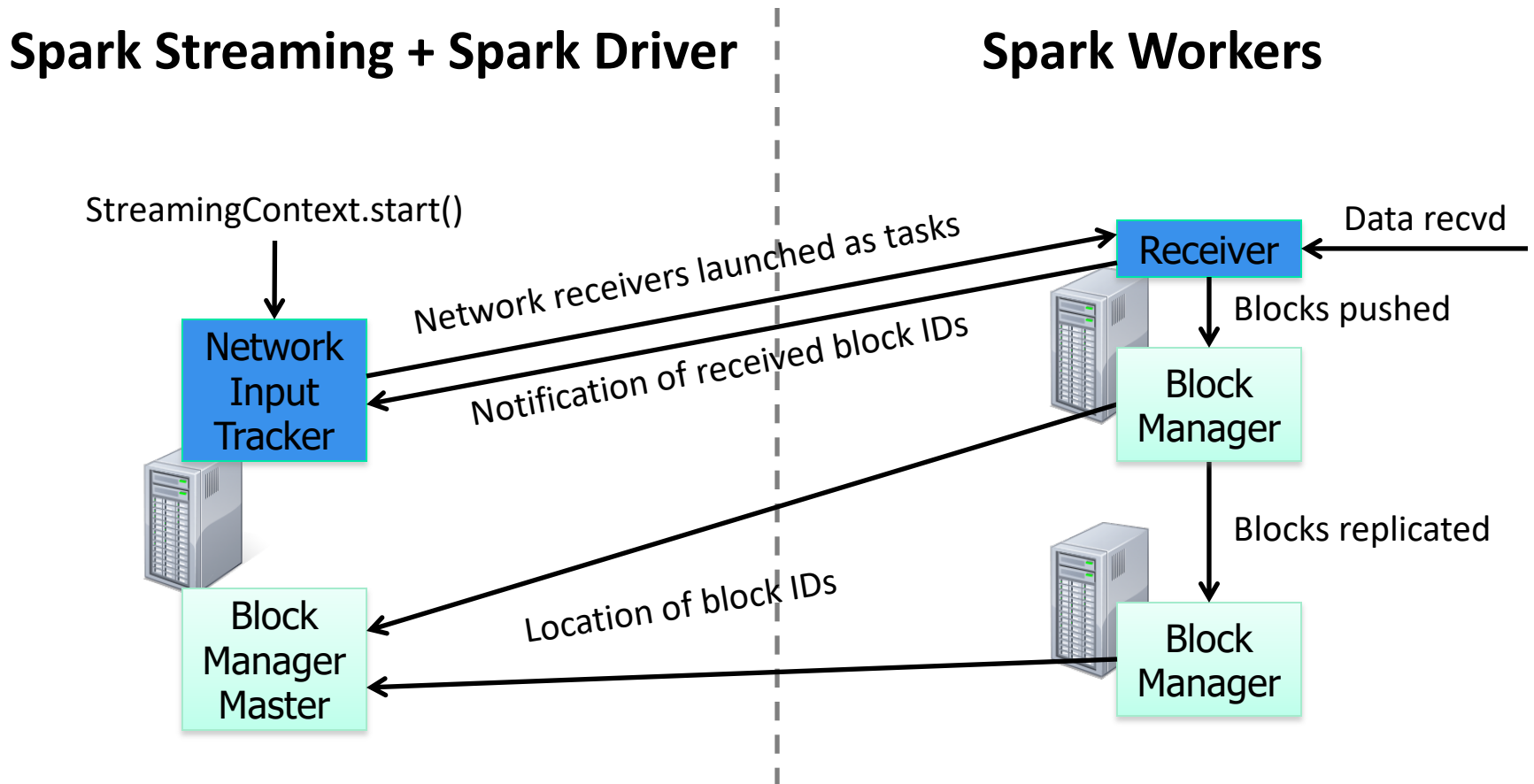
- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

Components

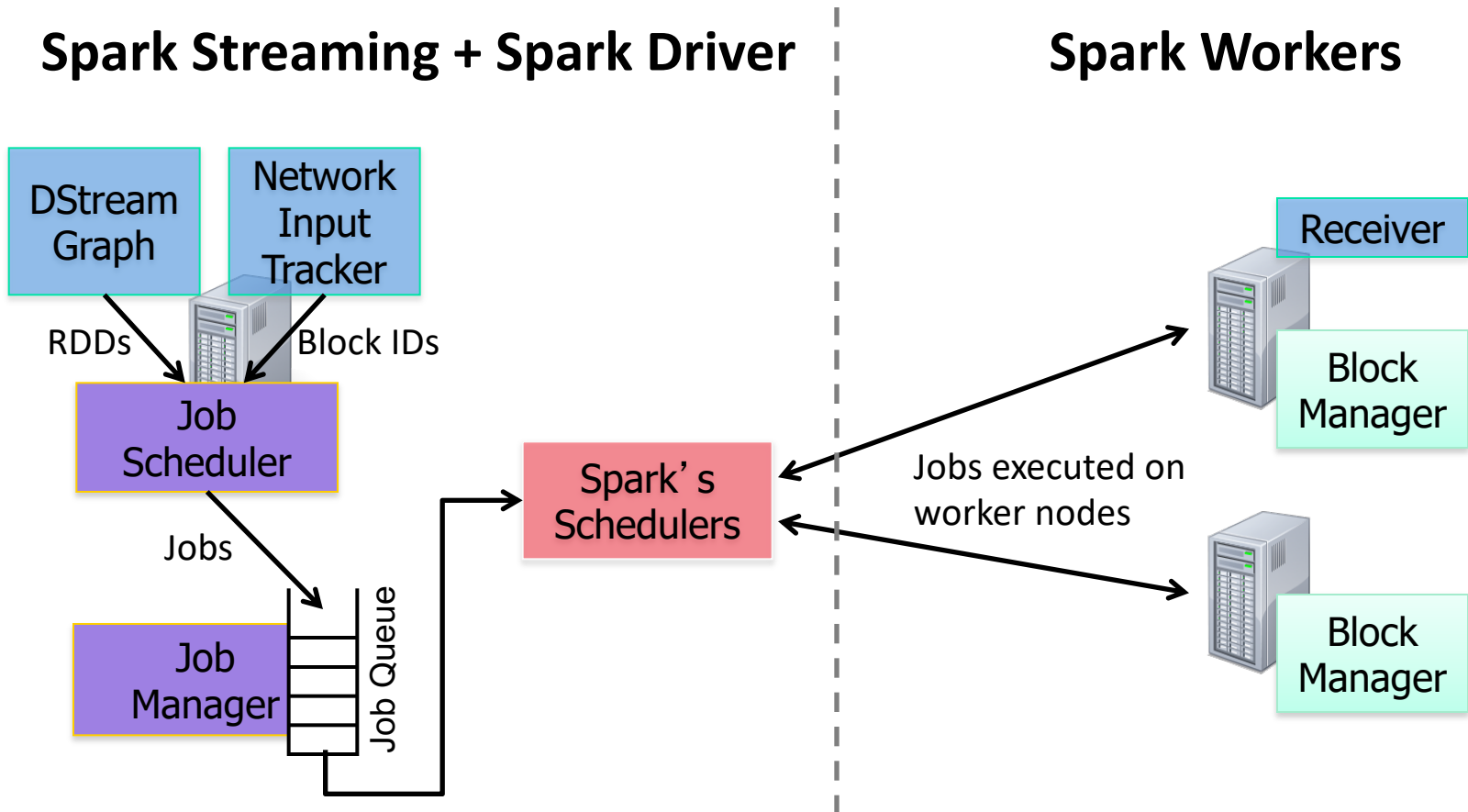


- **Network Input Tracker** – Keeps track of the data received by each network receiver and maps them to the corresponding input DStreams
- **Job Scheduler** – Periodically queries the DStream graph to generate Spark jobs from received data, and hands them to Job Manager for execution
- **Job Manager** – Maintains a job queue and executes the jobs in Spark

Execution Model – Receiving Data



Execution Model – Job Scheduling



Job Scheduling

- Each output operation used generates a job
 - More jobs → more time taken to process batches → higher batch duration
- Job Manager decides how many concurrent Spark jobs to run
 - Default is 1, can be set using Java property `spark.streaming.concurrentJobs`
 - If you have multiple output operations, you can try increasing this property to reduce batch processing times and so reduce batch duration

Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

DStream Persistence

- If a DStream is set to persist at a storage level, then all RDDs generated by it set to the same storage level
- When to persist?
 - If there are multiple transformations / actions on a DStream
 - If RDDs in a DStream is going to be used multiple times
- Window-based DStreams are automatically persisted in memory

DStream Persistence

- Default storage level of DStreams is `StorageLevel.MEMORY_ONLY_SER` (i.e. in memory as serialized bytes)
 - Except for input DStreams which have `StorageLevel.MEMORY_AND_DISK_SER_2`
 - Note the difference from RDD' s default level (no serialization)
 - Serialization reduces random pauses due to GC providing more consistent job processing times

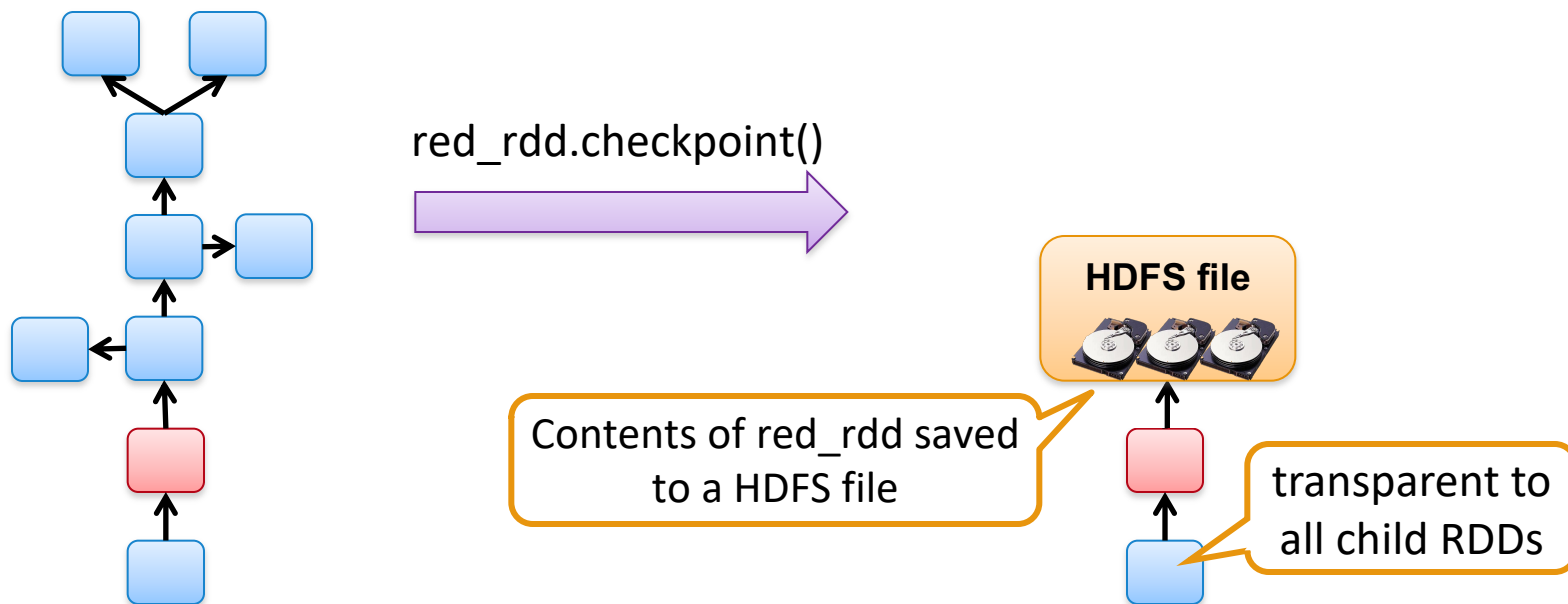
Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

What is RDD checkpointing?

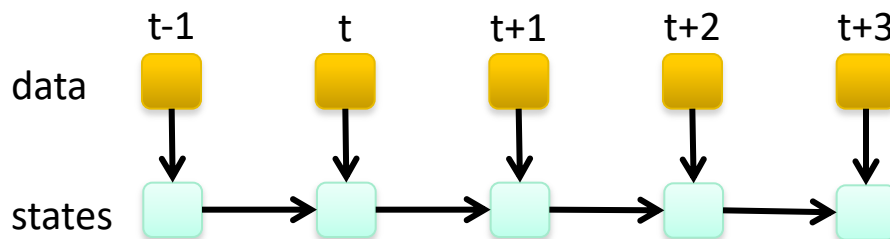
Saving RDD to HDFS to prevent RDD graph from growing too large

- Done internally in Spark transparent to the user program
- Done lazily, saved to HDFS the first time it is computed



Why is RDD checkpointing necessary?

Stateful DStream operators can have infinite lineages

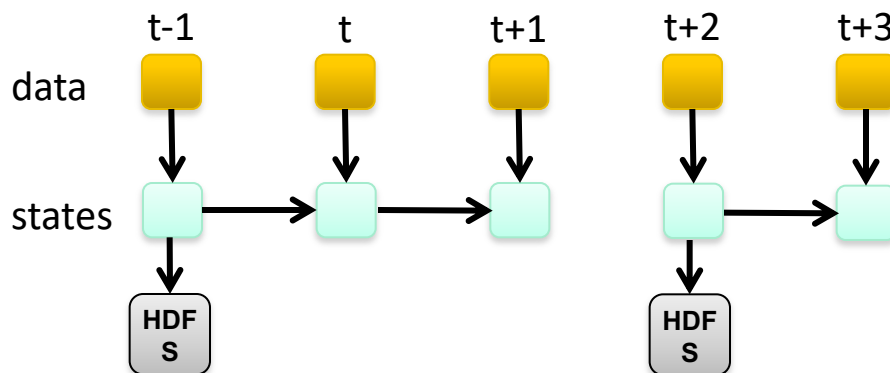


Large lineages lead to ...

- Large closure of the RDD object → large task sizes → high task launch times
- High recovery times under failure

Why is RDD checkpointing necessary?

Stateful DStream operators can have infinite lineages



Periodic RDD checkpointing solves this

Useful for iterative Spark programs as well

RDD Checkpointing

- Periodicity of checkpoint determines a tradeoff
 - Checkpoint too frequent: HDFS writing will slow things down
 - Checkpoint too infrequent: Task launch times may increase
 - Default setting checkpoints at most once in 10 seconds
 - Try to checkpoint once in about 10 batches

Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

Performance Tuning

Step 1

Achieve a stable configuration that can sustain the streaming workload

Step 2

Optimize for lower latency

Step 1: Achieving Stable Configuration

How to figure out a good stable configuration?

- Start with a low data rate, small number of nodes, reasonably large batch duration (5 – 10 seconds)
- Increase the data rate, number of nodes, etc.
- Find the bottleneck in the job processing
 - Jobs are divided into stages
 - Find which stage is taking the most amount of time

System Stability

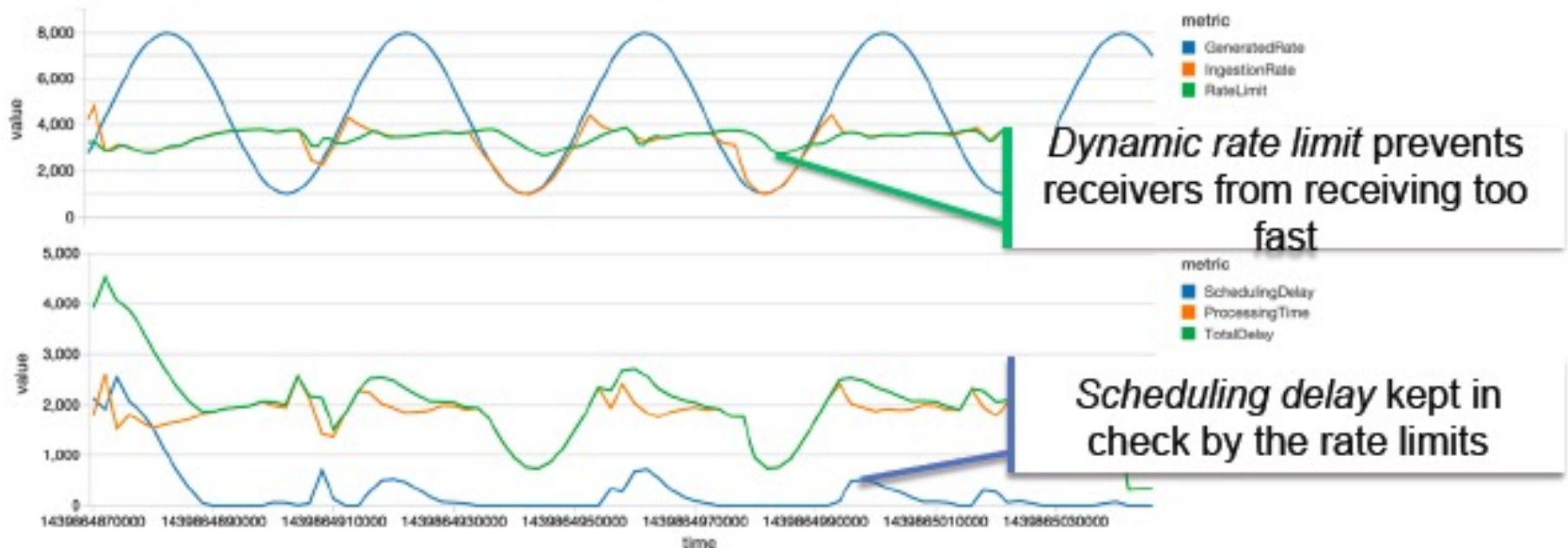
- Streaming applications may have to deal with variations in data rates and processing rates
- For stability, any streaming application must receive data only as fast as it can process
- Static rate limits on receivers [Spark 1.1]
 - But hard to figure out the right rate

Backpressure [Spark 1.5]

- System **automatically** and **dynamically** adapts rate limits to ensure stability under any processing conditions
- If sinks slow down, then the system automatically pushes back on the source to slow down receiving
- System uses batch processing times and scheduling delays experienced to set rate limits
- Well known PID controller theory (used in industrial control systems) is used to calculate appropriate rate limit
 - Contributed by Typesafe
- Enabled by setting Spark configuration flag
 - `spark.streaming.backpressure.enabled` to TRUE

Backpressure [Spark 1.5]

- System automatically and dynamically adapts rate limits

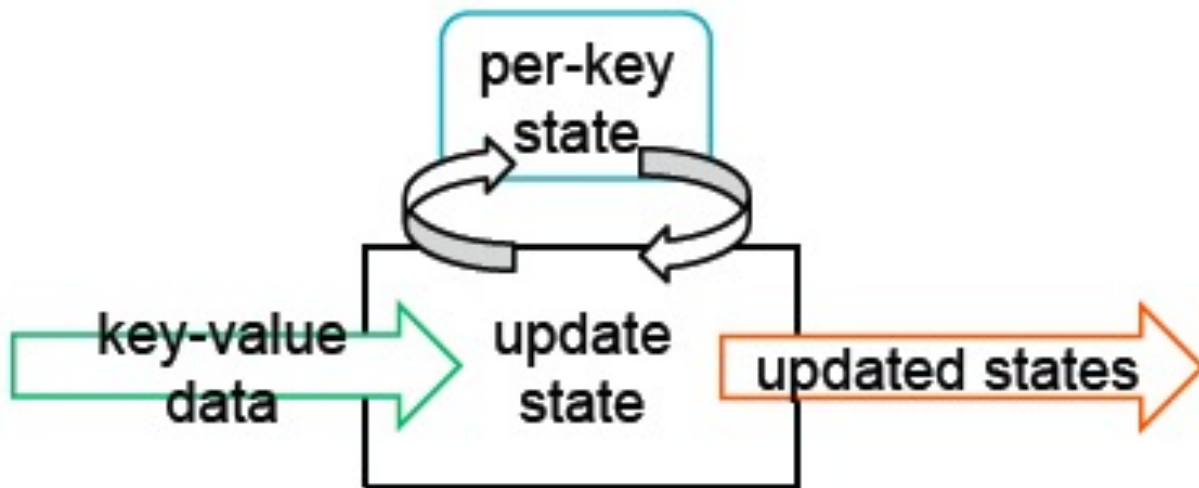


Improved State Management

Earlier stateful stream processing done with `updateStateByKey`

```
def stateUpdateFunc(newData, lastState) => updatedState
```

```
val stateDStream = keyValueDStream.updateStateByKey(stateUpdateFunc)
```



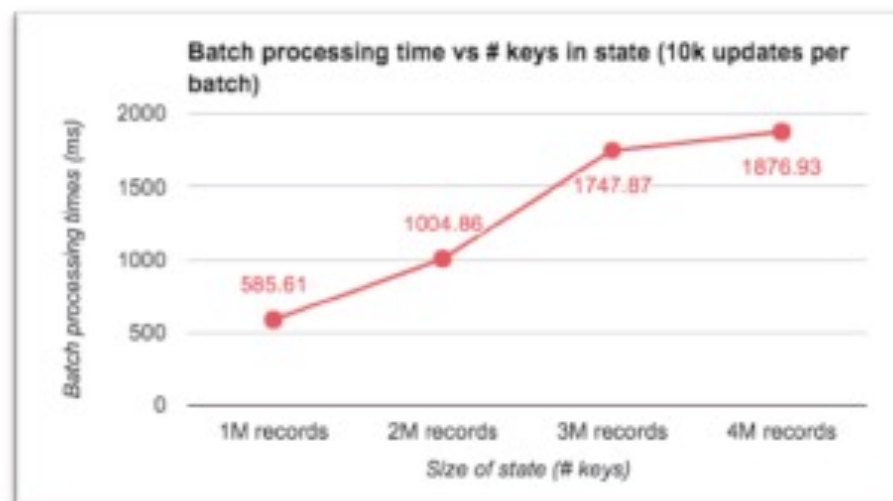
Improved State Management

Feedback from community about `updateStateByKey`

Need to keep much larger state

Processing times of batches increase with the amount state, limits performance

Need to expire keys that have received no data for a while



Improved State Management

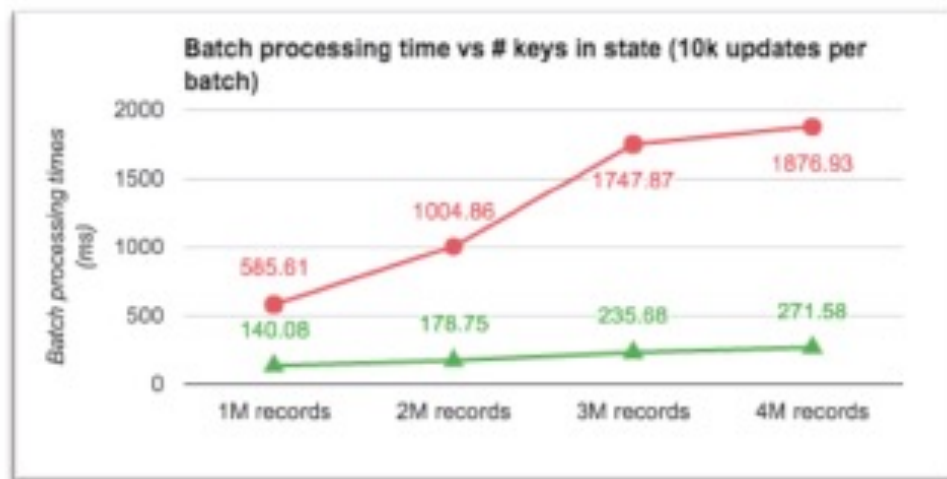
New API with timeouts: `trackStateByKey`

```
def updateFunc(values, state) => emittedData  
  // call state.update(newState) to update state
```

```
keyValueDStream.trackStateByKey(  
  StateSpec.function(updateFunc).timeout(Minutes(10))
```

Can provide order of magnitude higher performance than `updateStateByKey`

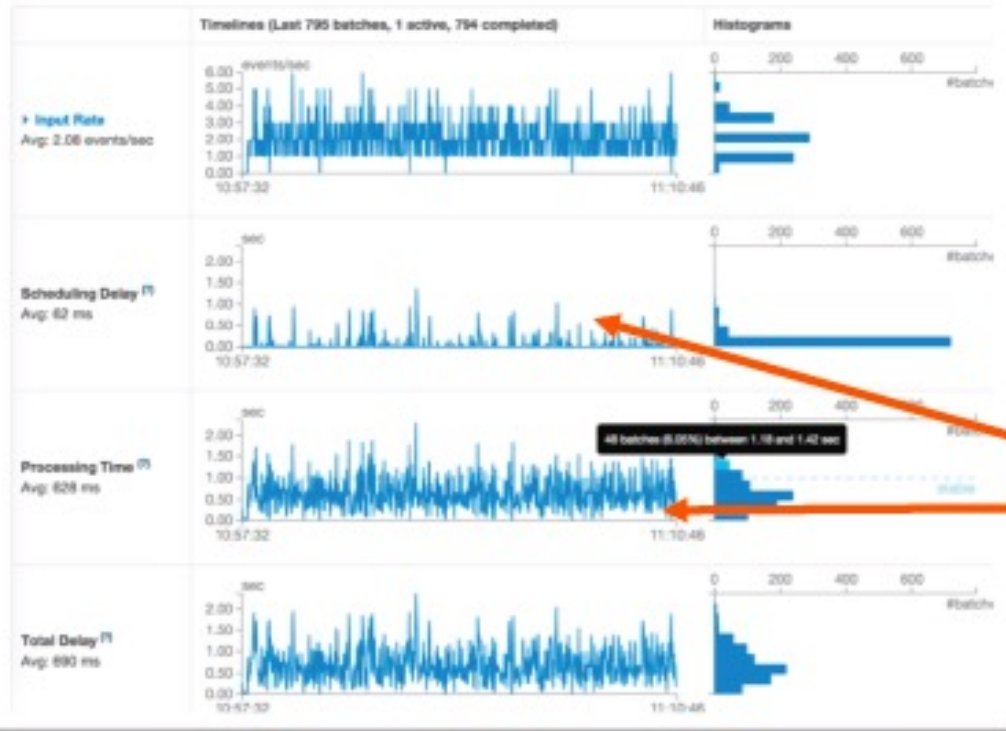
<https://issues.apache.org/jira/browse/SPARK-2629>



Visualizations

Streaming Statistics

Running batches of 1 second for 13 minutes 14 seconds since 2015/06/08 10:57:31 (794 completed batches, 1663 records)



Stats over last 1000 batches

For stability

Scheduling delay should be approx 0

Processing Time approx < batch interval

Visualizations

Details of individual batches

Active Batches (1)

Batch Time	Input Size	Scheduling Delay	Processing Time	Status
2015/06/08 11:10:46				

Completed Batches (last 794)

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 2015/06/29 15:34:00

Batch Duration: 30 s
Input data size: 0 records
Scheduling delay: 0 ms
Processing time: 72 ms
Total delay: 72 ms
Input Metadata:

Input	Metadata
File stream [1]	file:/Users/zsx/streaming/a.txt, file:/Users/zsx/streaming/b.txt, file:/Users/zsx/streaming/c.txt, file:/Users/zsx/
Kafka direct stream [0]	OffsetRange(topic: 'test', partition: 0, range: [1 -> 1]), OffsetRange(topic: 'test2', partition: 0, range: [1 -> 1])

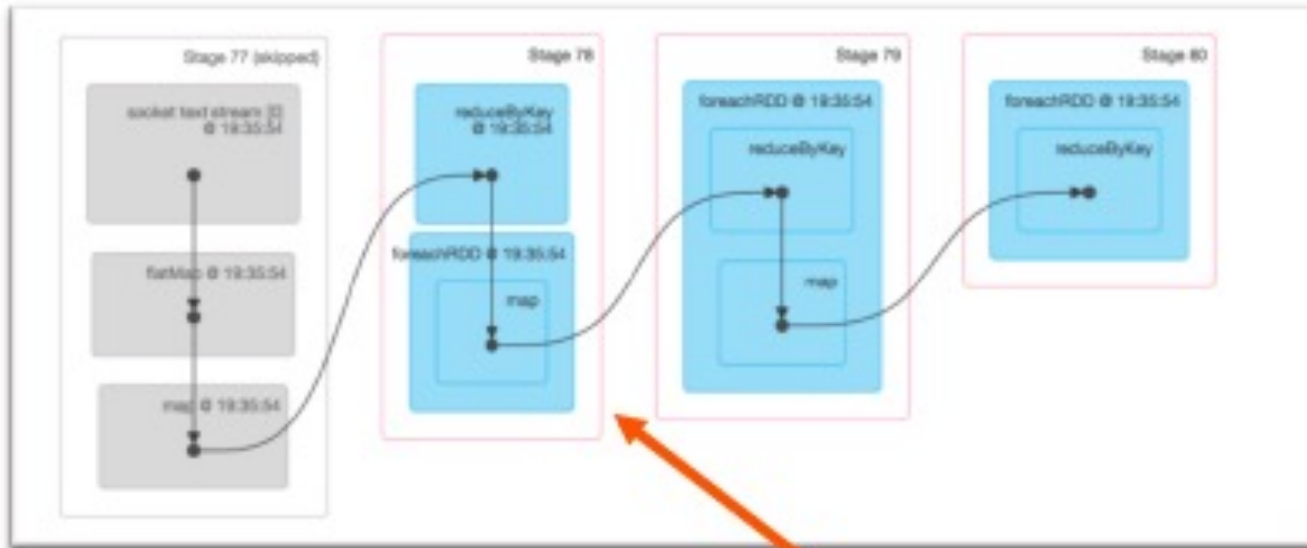
Output Operations

Output Op Id	Description	Duration	Job Id	Duration	Stages: Succeeded/Total	Tar
0	print at DirectKafkaWordCount.scala:69	59 ms	6	48 ms	2/2	
			7	7 ms	1/1 (1 skipped)	
			8	4 ms	1/1 (1 skipped)	

**Kafka offsets processed in each batch,
Can help in debugging bad data**

List of Spark jobs in each batch

Visualizations

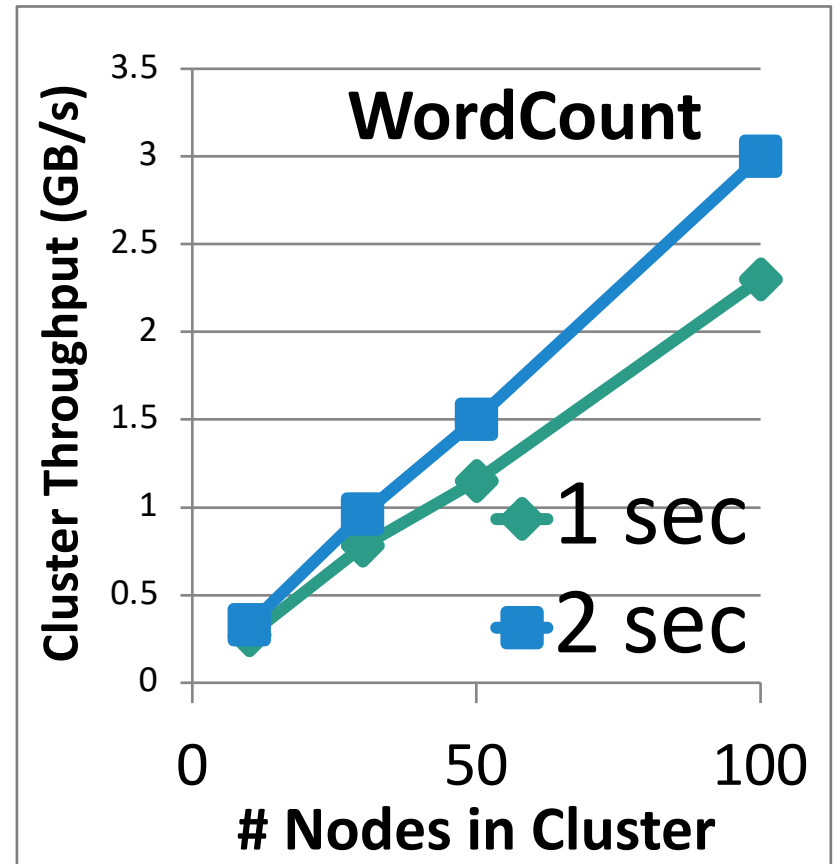
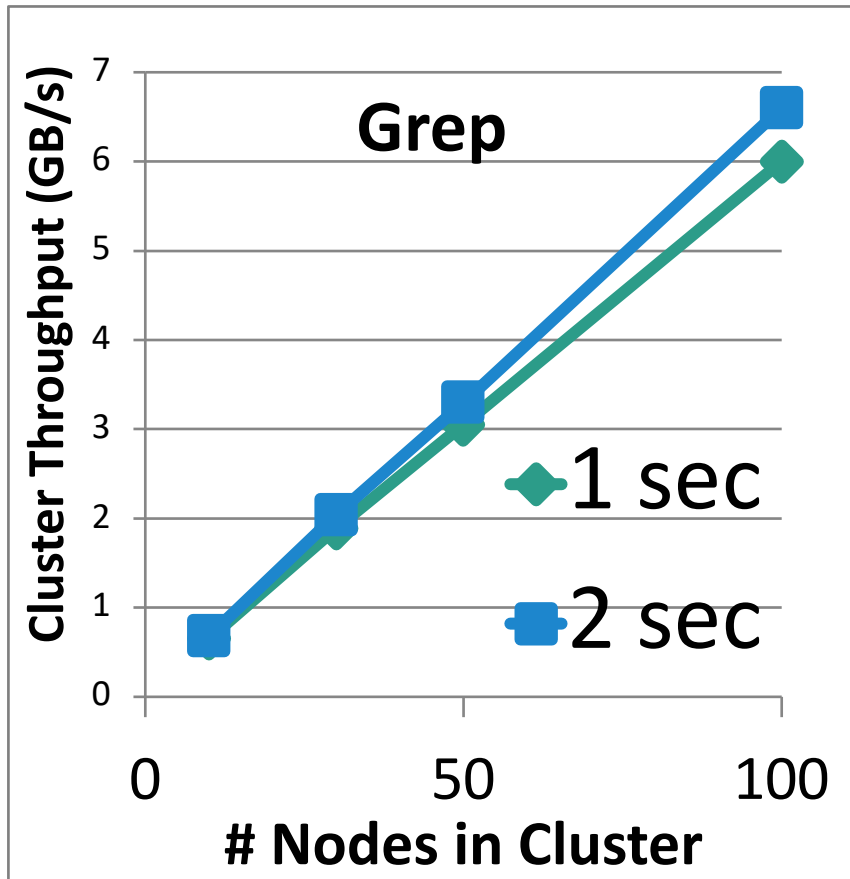


Full DAG of RDDs and stages generated by Spark Streaming

Performance

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

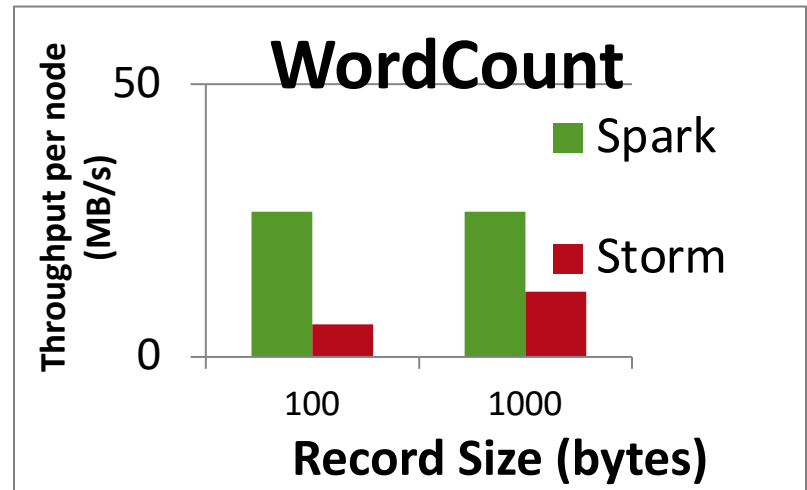
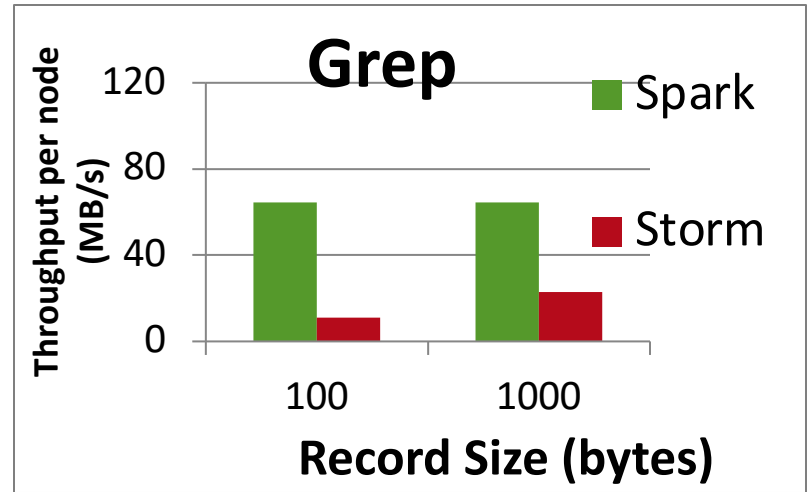
- Tested with 100 streams of data on 100 EC2 instances with 4 cores each



Comparison with Storm and S4

Higher throughput than Storm

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



Small code base

- 5000 LOC for Scala API (+ 1500 LC for Java API)
 - Most DStream code mirrors the RDD code

Subsequent Extensions on Spark Streaming

- API and Libraries Support of Streaming DataFrames
 - Logical-to-physical plan optimizations
 - Tungsten-based binary optimizations
 - Support for event-time based windowing
 - Support for Out-of-Order Data
- Add Native Infrastructure support for Dynamic Allocation for Streaming
 - Dynamically scale the cluster resources based on processing load
 - Need to work with Backpressure to scale up/down while maintaining stability
- Programmable monitoring by exposing more info via Streaming Listener
- Performance Enhancement: higher throughput and lower latency, specially for Stateful Ops, e.g. trackStateByKey

Structured Streaming in Spark 2.0



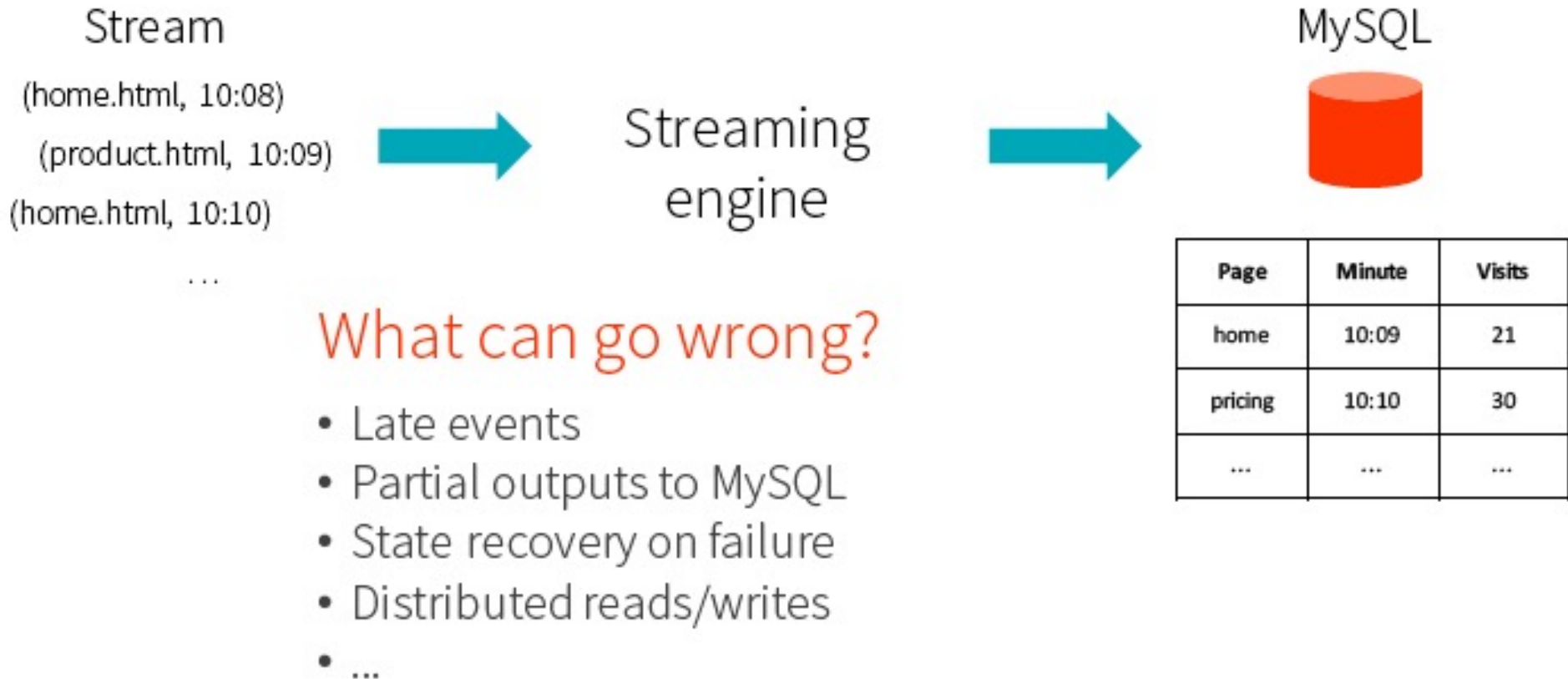
Structured Streaming
real-time engine
on SQL/DataFrames

Motivation for Structured Streaming

- Real-time processing is increasingly important
- Most applications need to combine it with Batch & Interactive queries, e.g.
 - Track state using a stream, then run SQL queries
 - Train an Machine Learning model offline, then update it with new, online data

Not just streaming, but
“continuous applications”

Challenges of Integrating Streaming into a Real-world Application Infrastructure



Complex Programming Models

Data

Late arrival, varying distribution over time, ...

Processing

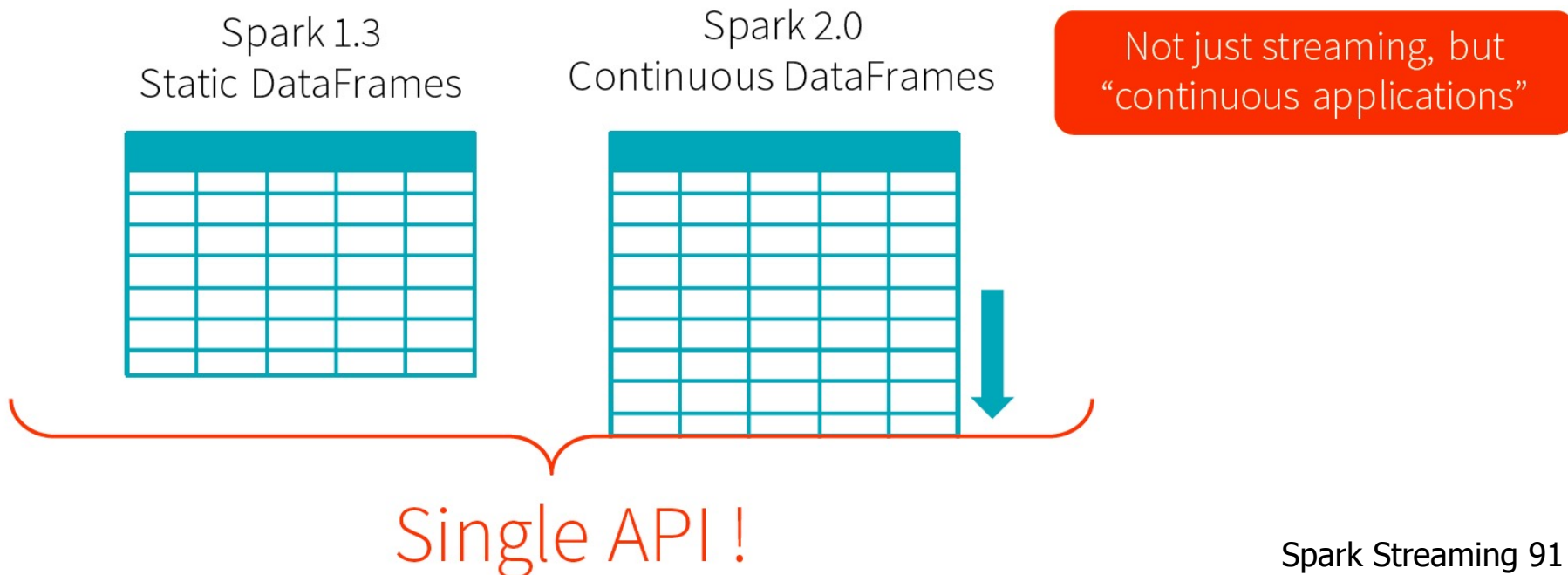
Business logic change & new ops
(windows, sessions)

Output

How do we define
output over time & correctness?

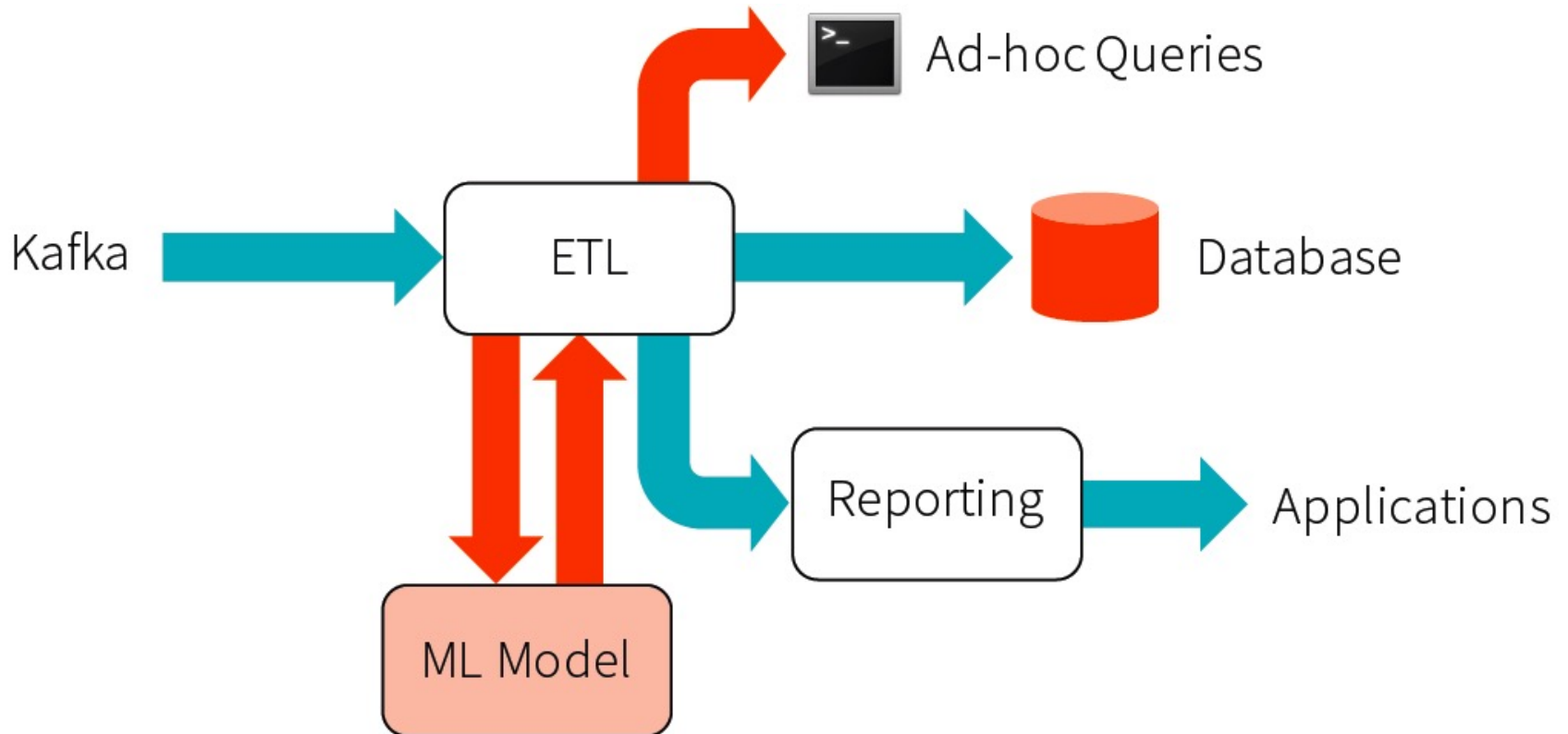
Structured Streaming

- High-level Streaming API built on Spark SQL engine
 - Run the same queries on DataFrames
 - Event-time, windowing, sessions, source and sinks
- Unify Streaming, Interactive and Batch Queries
 - Aggregate data in a stream, then serve using JDBC
 - Change queries at runtime
 - Build and Apply Machine Learning models



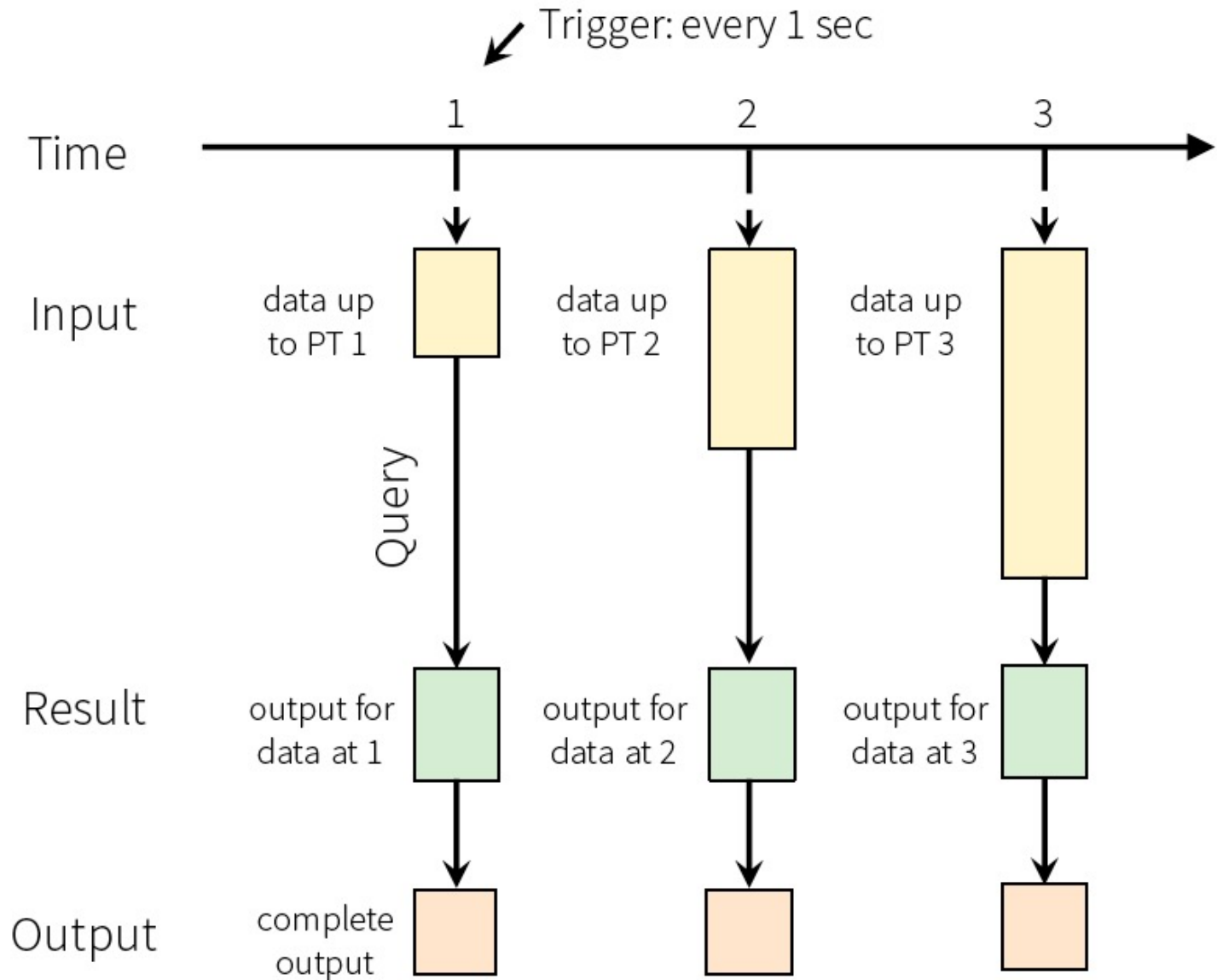
An Example

- Traditional streaming
- Other processing types

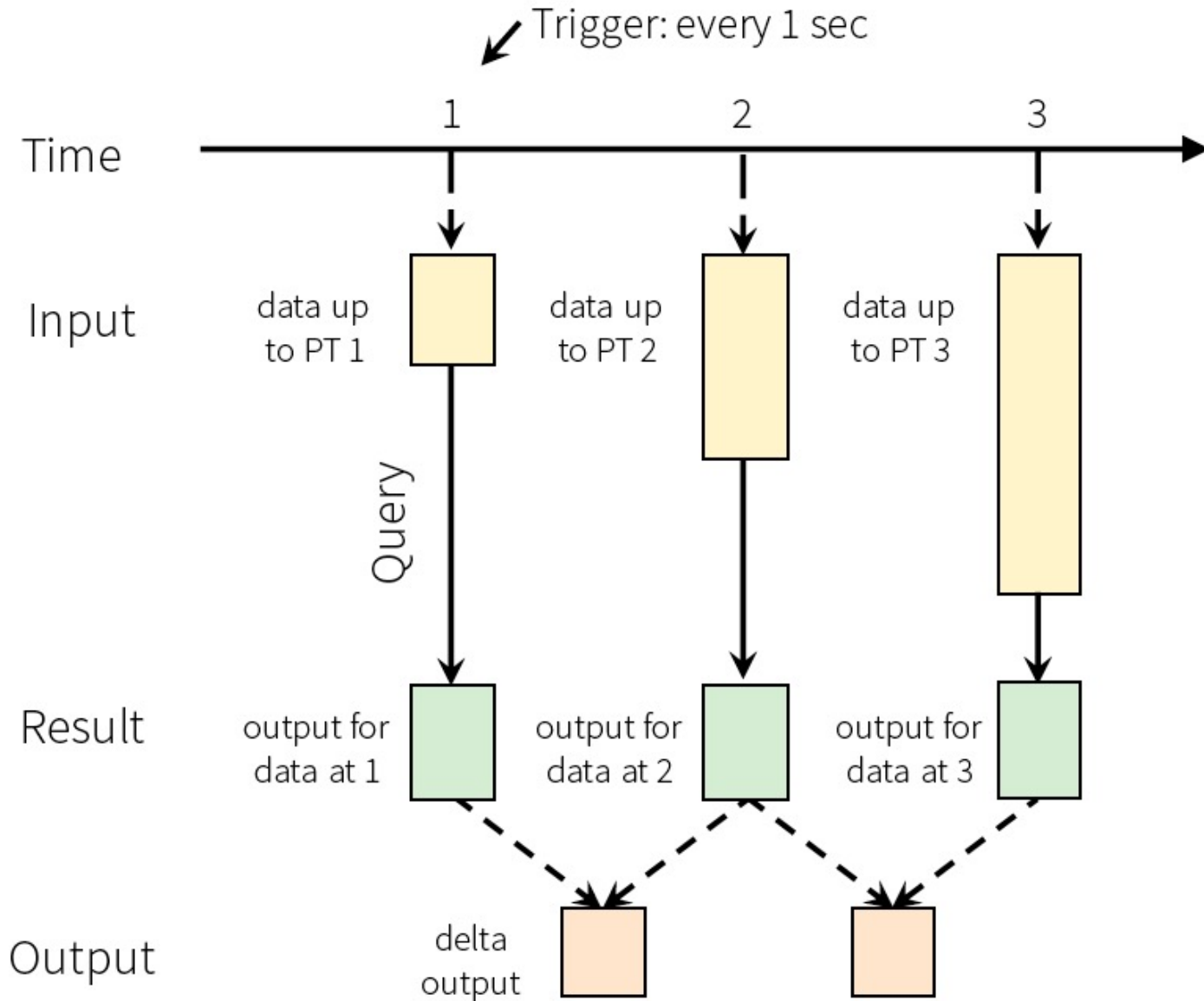


Goal: end-to-end continuous applications

Model for Structured Streaming



Model for Structured Streaming



Model Details for Structured Streaming

- **Input Sources:** Append-Only Tables
- **Queries:** New operators for Windowing, Sessions, etc
- **Triggers:** based on time (e.g. every 1 sec)
- **Output modes:** Complete, Deltas, Update-in-Place

Batch ETL with DataFrames

```
input = spark.read  
    .format("json")  
    .load("source-path")
```

Read from Json file

```
result = input  
    .select("device", "signal")  
    .where("signal > 15")
```

Select some devices

```
result.write  
    .format("parquet")  
    .save("dest-path")
```

Write to parquet file

Streaming ETL with DataFrames

```
input = spark.read  
    .format("json")  
    .stream("source-path")
```

Read from Json **file stream**
Replace **load()** with **stream()**

```
result = input  
    .select("device", "signal")  
    .where("signal > 15")
```

Select some devices
Code does not change

```
result.write  
    .format("parquet")  
    .startStream("dest-path")
```

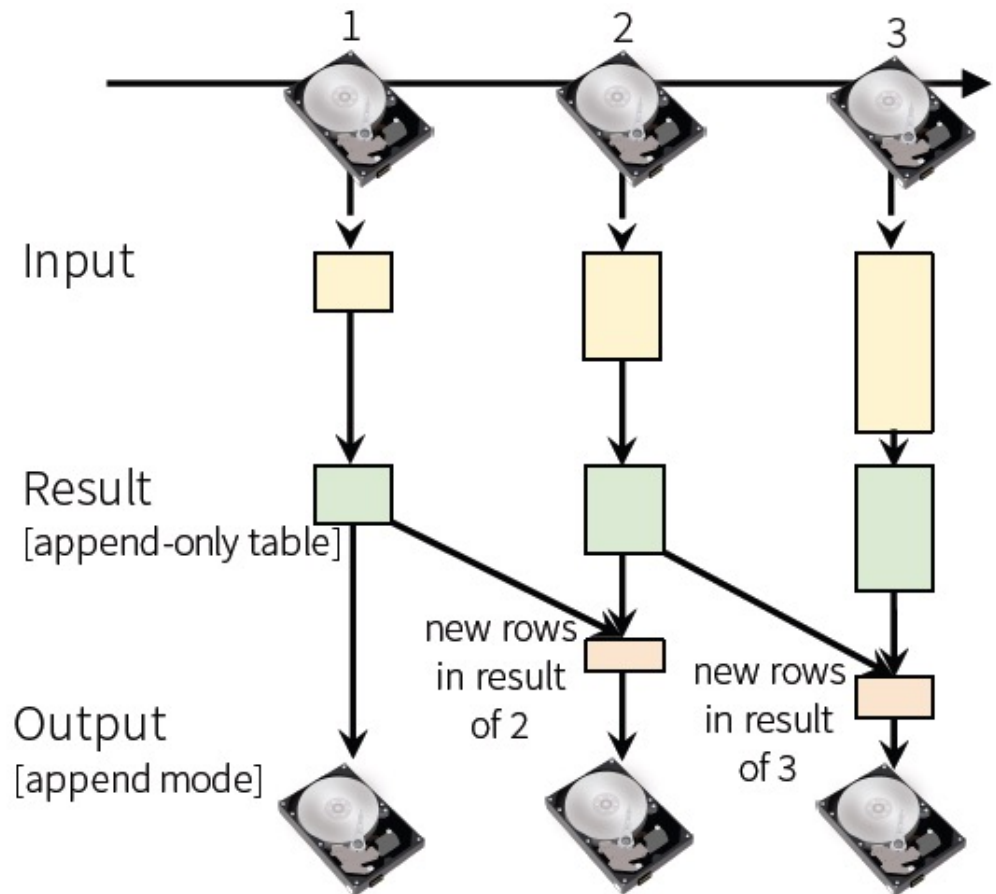
Write to Parquet **file stream**
Replace **save()** with **startStream()**

Streaming ETL with DataFrames

```
input = spark.read
    .format("json")
    .stream("source-path")

result = input
    .select("device", "signal")
    .where("signal > 15")

result.write
    .format("parquet")
    .startStream("dest-path")
```



Continuous Aggregation

```
input.avg("signal")
```

Continuously compute *average* signal *across all devices*

```
input.groupBy("device-type")  
  .avg("signal")
```

Continuously compute *average* signal of *each type of device*

```
input.groupBy(  
  $"device-type",  
  window($"event-time-col", "10 min"))  
  .avg("signal")
```

Continuously compute *average* signal of *each type of device* in last 10 minutes using *event-time*

Joining Streams with Static Data

```
kafkaDataset = spark.read
  .kafka("iot-updates")
  .stream()

staticDataset = ctx.read
  .jdbc("jdbc://", "iot-device-info")

joinedDataset =
  kafkaDataset.join(
    staticDataset, "device-type")
```

Join streaming data from Kafka with static data via JDBC to enrich the streaming data ...

... without having to think that you are joining streaming data

Output Modes for Structured Streaming

Defines what is outputted every time there is a trigger
Different output modes make sense for different queries

Append mode with
non-aggregation queries

```
input.select("device", "signal")  
  .write  
  .outputMode("append")  
  .format("parquet")  
  .startStream("dest-path")
```

Complete mode with
aggregation queries

```
input.agg(count("*"))  
  .write  
  .outputMode("complete")  
  .format("parquet")  
  .startStream("dest-path")
```

Query Management

```
query = result.write  
    .format("parquet")  
    .outputMode("append")  
    .startStream("dest-path")
```

```
query.stop()  
query.awaitTermination()  
query.exception()
```

```
query.sourceStatuses()  
query.sinkStatus()
```

query: a handle to the running streaming computation for managing it

- Stop it, wait for it to terminate
- Get status
- Get error, if terminated

Multiple queries can be active at the same time

Each query has unique name for keeping track

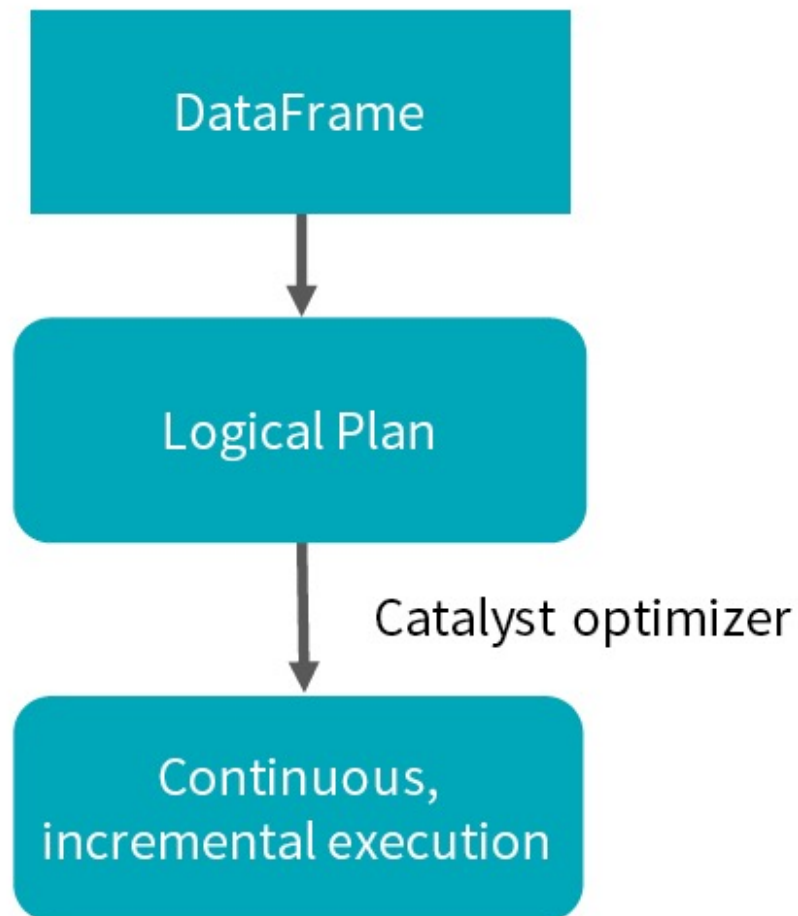
Execution for Structured Streaming

Logically:

- DataFrame operations on static data (i.e. as easy to understand as batch)

Physically:

- Spark automatically runs the query in Streaming fashion (i.e. incrementally and continuously)



Example: Batch Aggregation

```
logs = ctx.read.format("json").open("s3://logs")

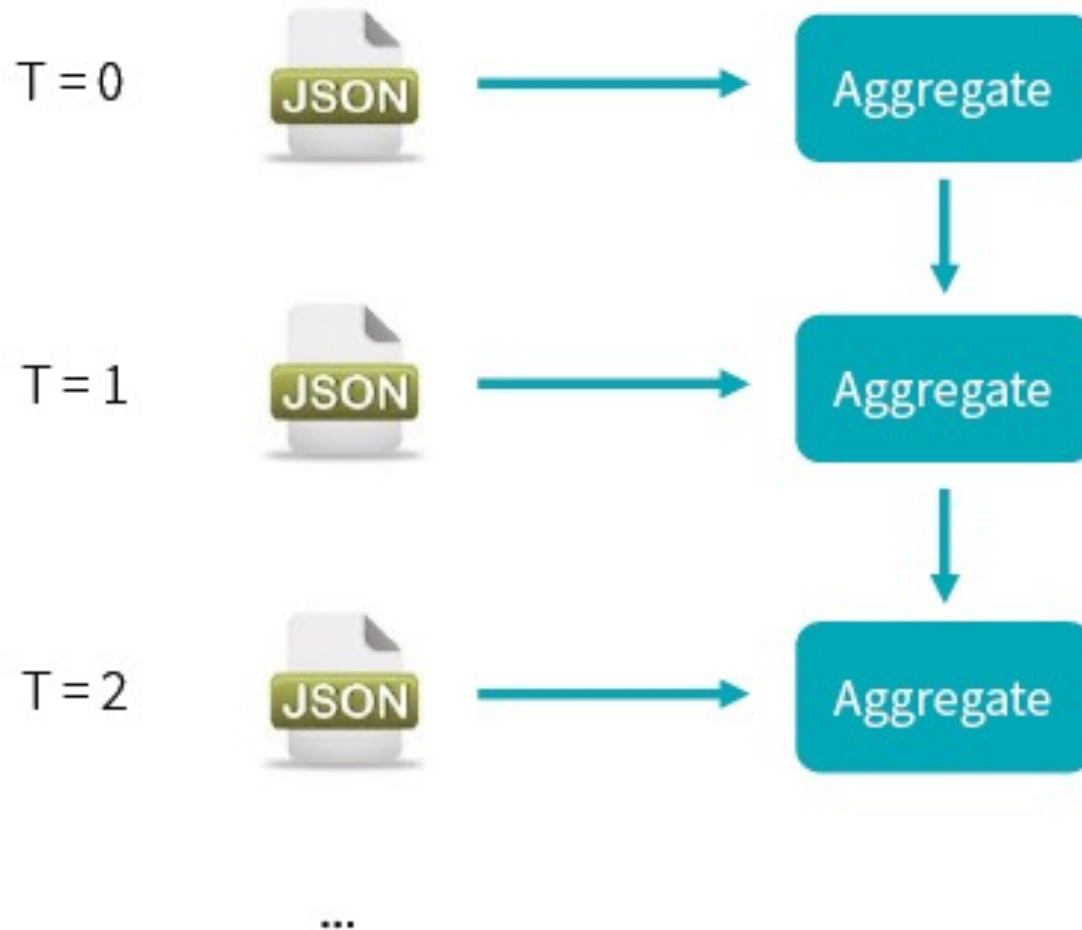
logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .save("jdbc:mysql://...")
```

Example: Continuous Aggregation

```
logs = ctx.read.format("json").stream("s3://logs")

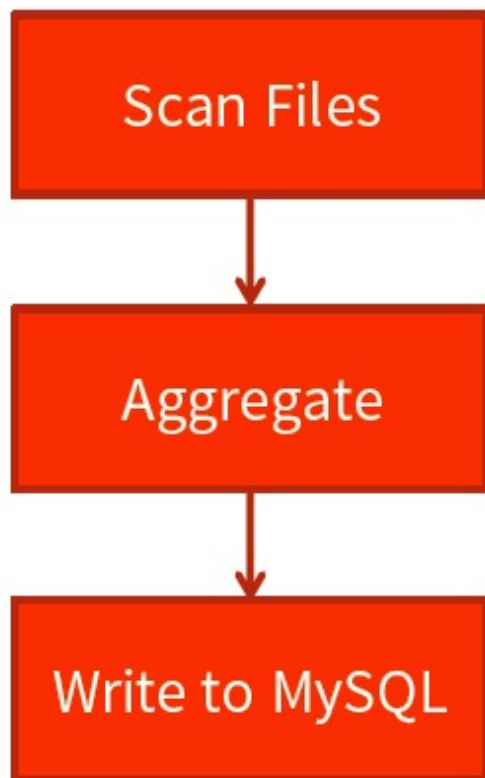
logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .stream("jdbc:mysql://...")
```

Automatic Incremental Execution



Incrementalized by Spark

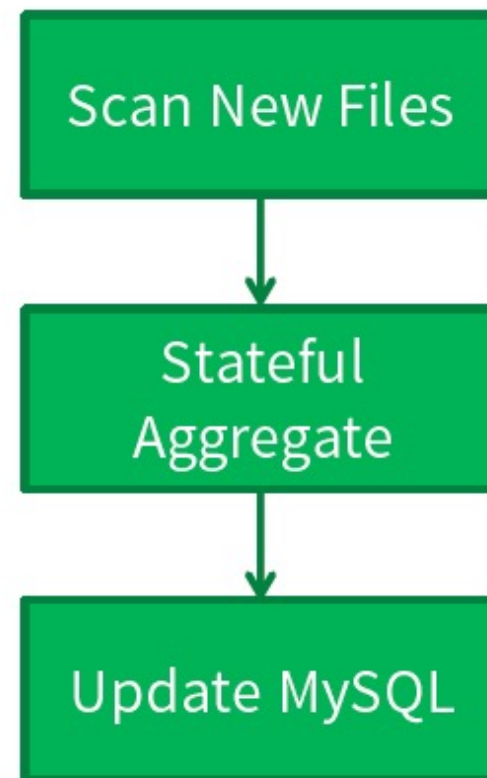
Batch



Transformation
requires
information
about the
structure

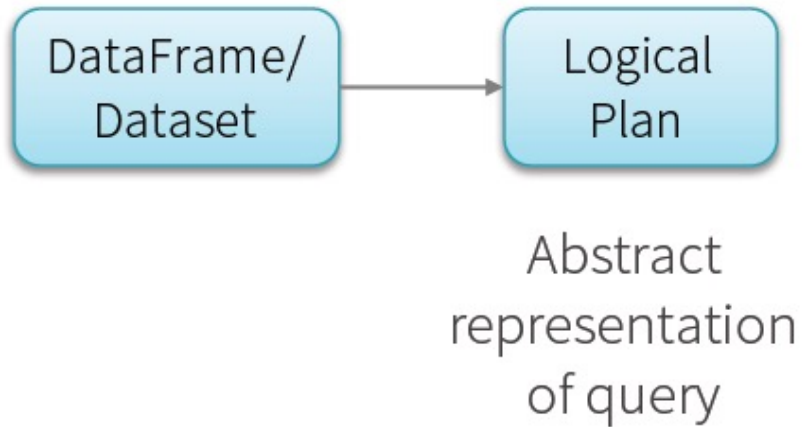


Continuous

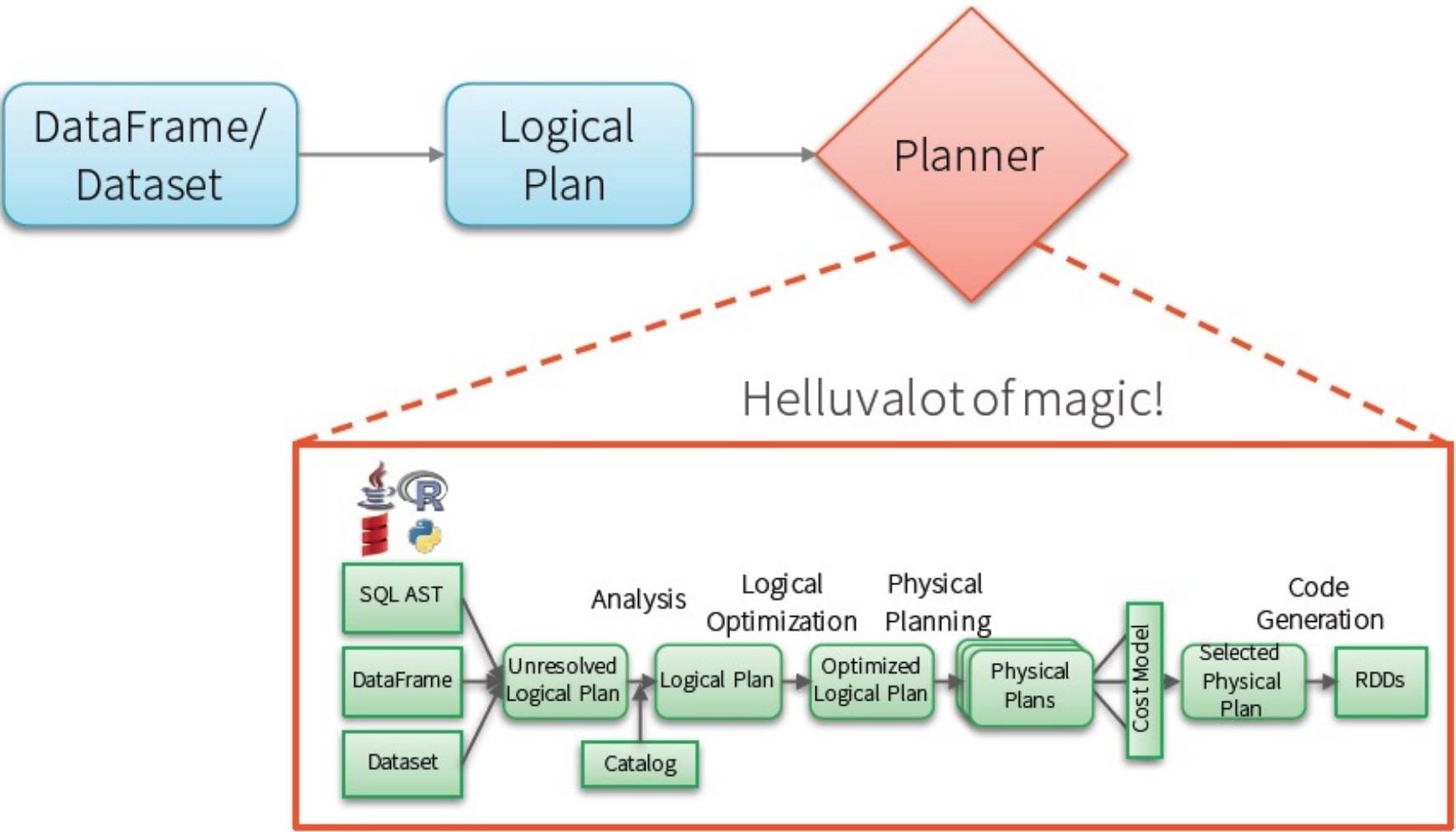


Inner Workings of Structured Streaming

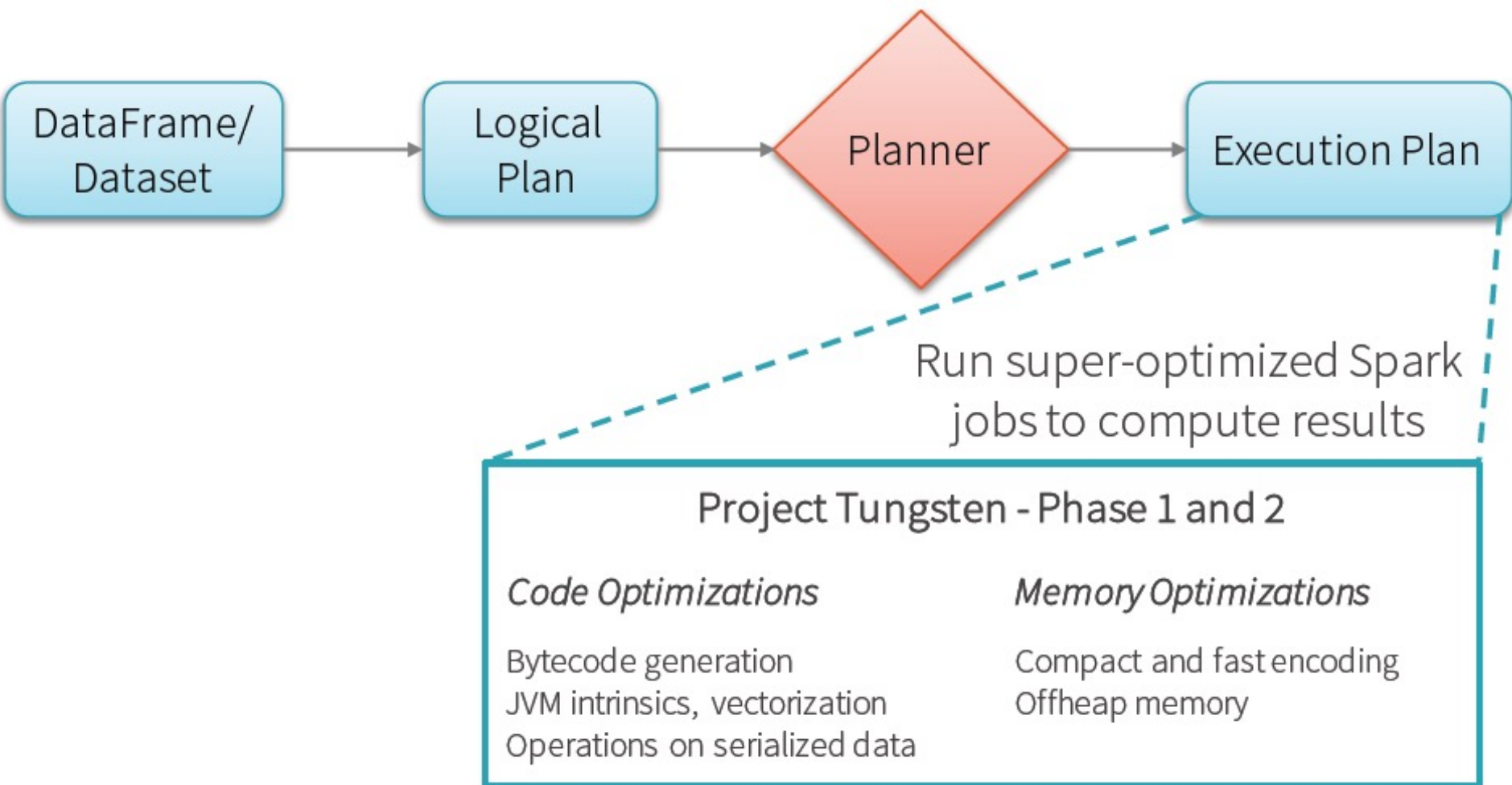
Batch Execution on Spark SQL



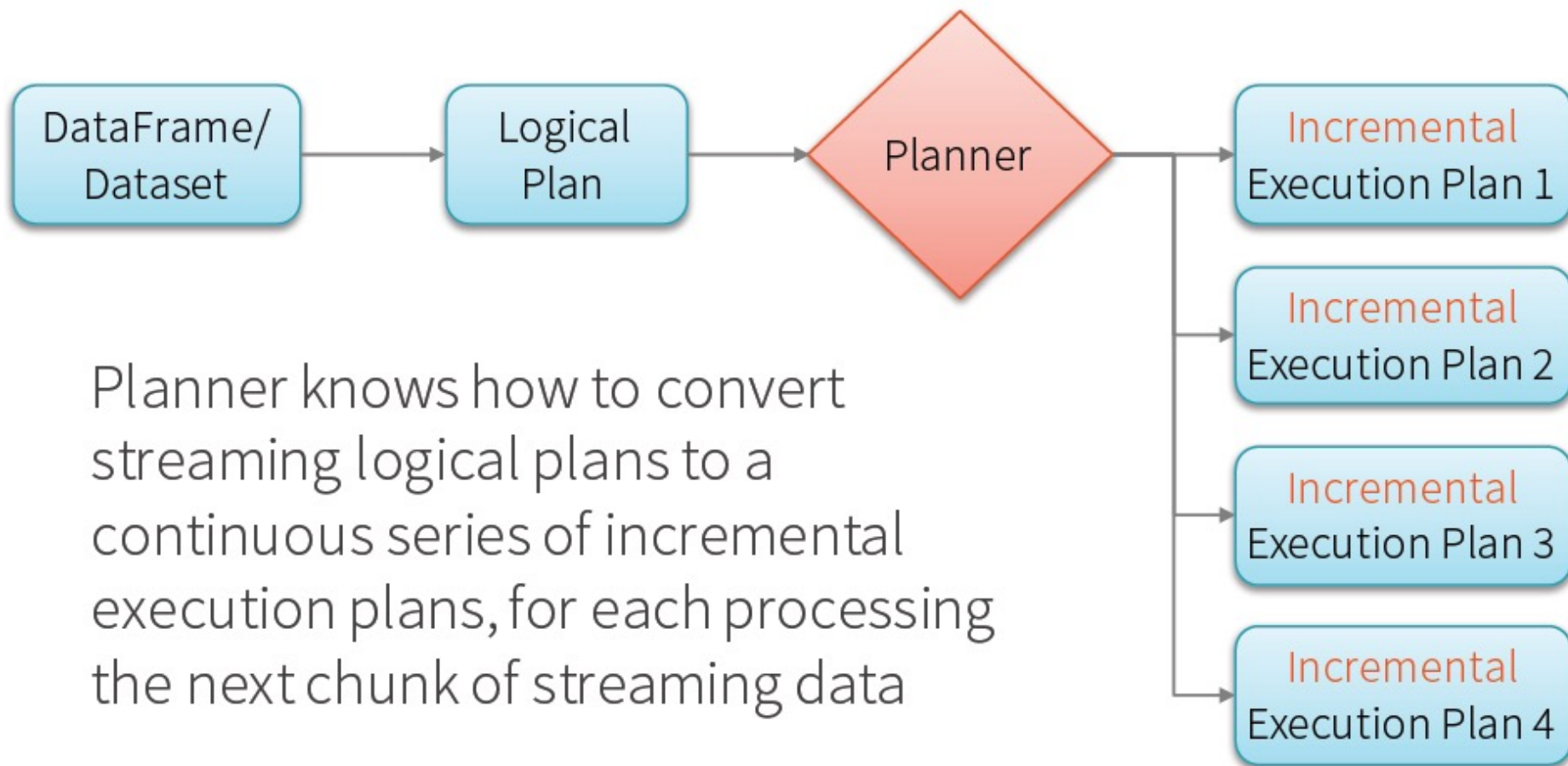
Batch Execution on Spark SQL



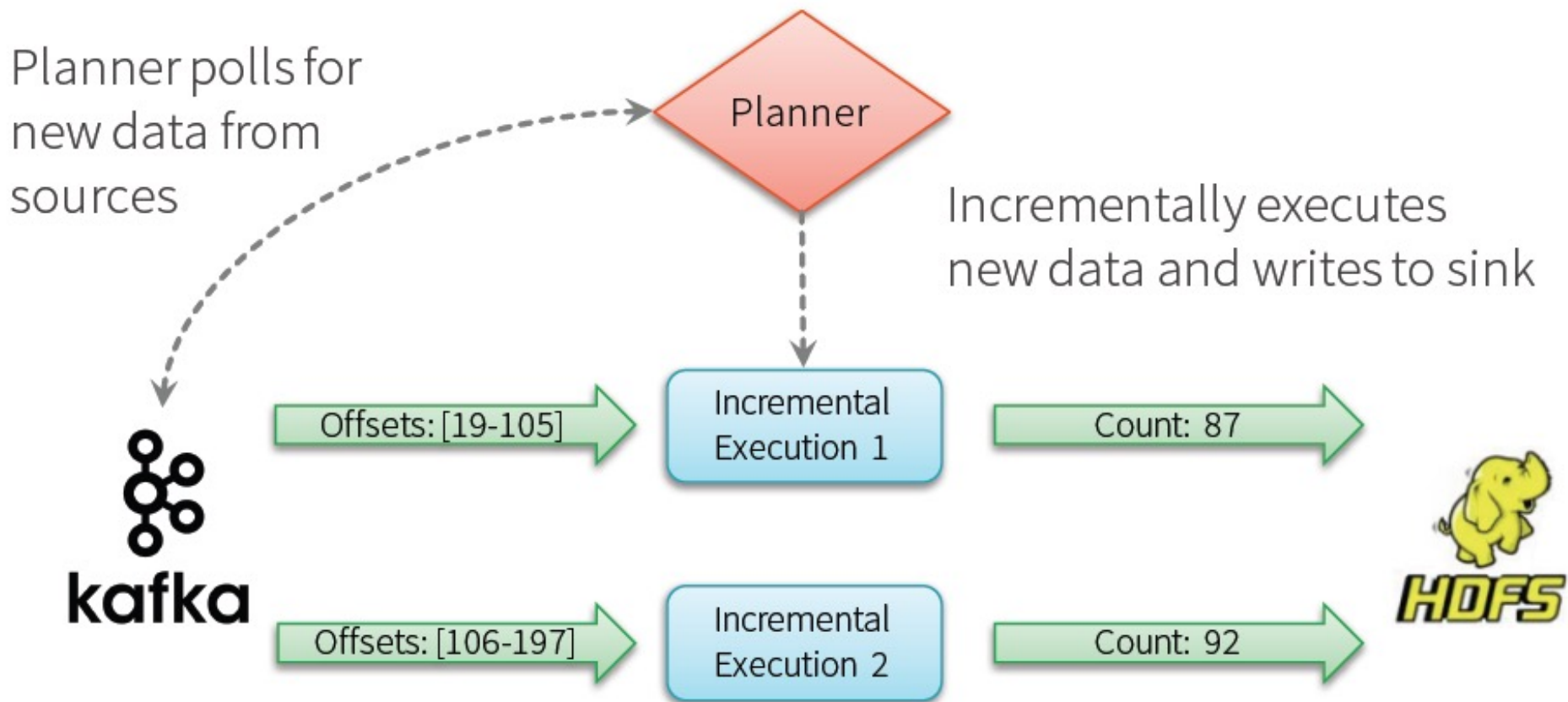
Batch Execution on Spark SQL



Continuous Incremental Execution

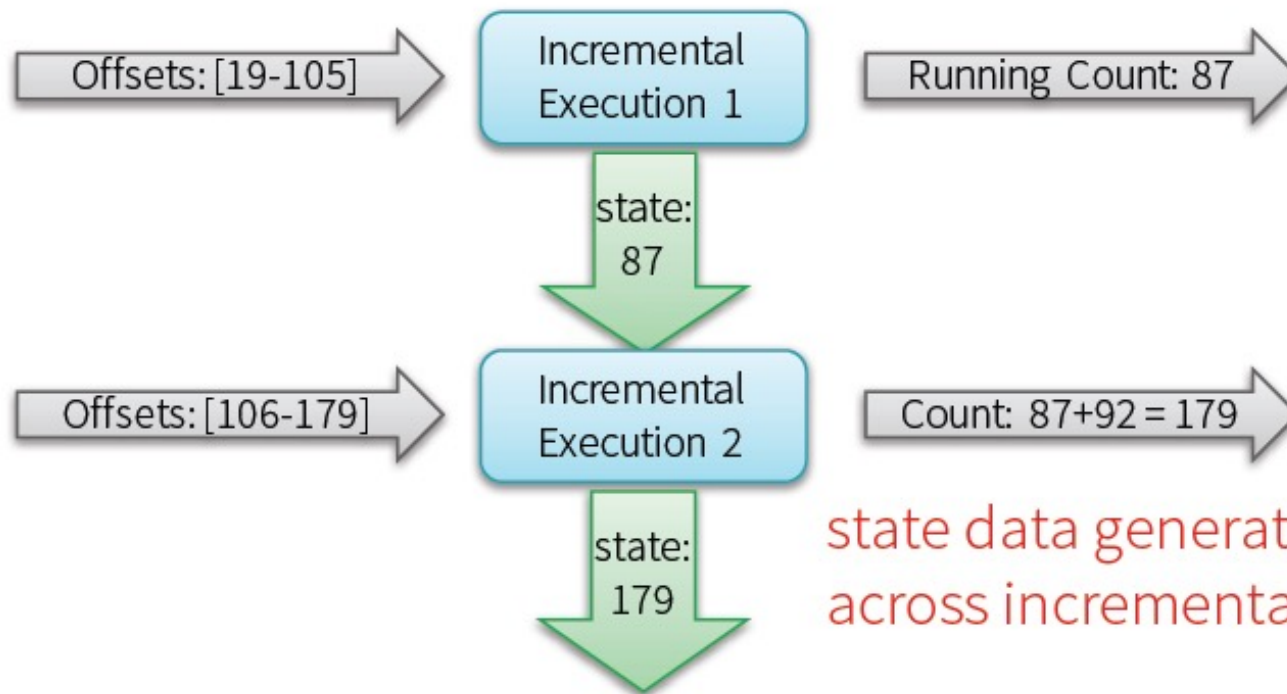


Continuous Incremental Execution



Continuous Aggregation

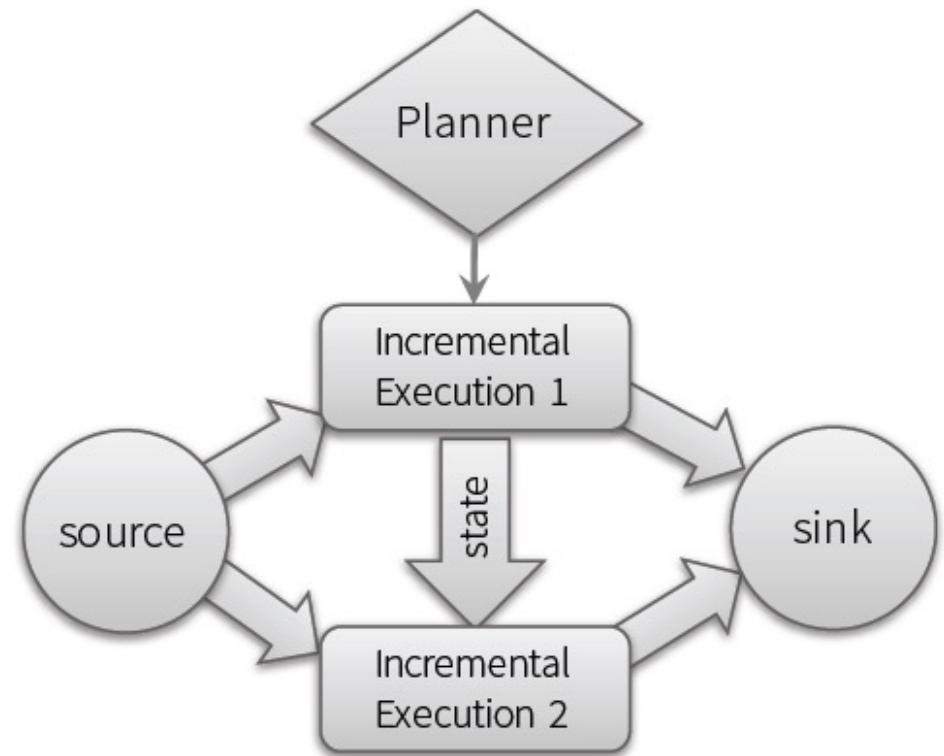
Maintain running aggregate as **in-memory state** backed by **WAL in file system** for fault-tolerance



state data generated and used across incremental executions

Fault-Tolerance

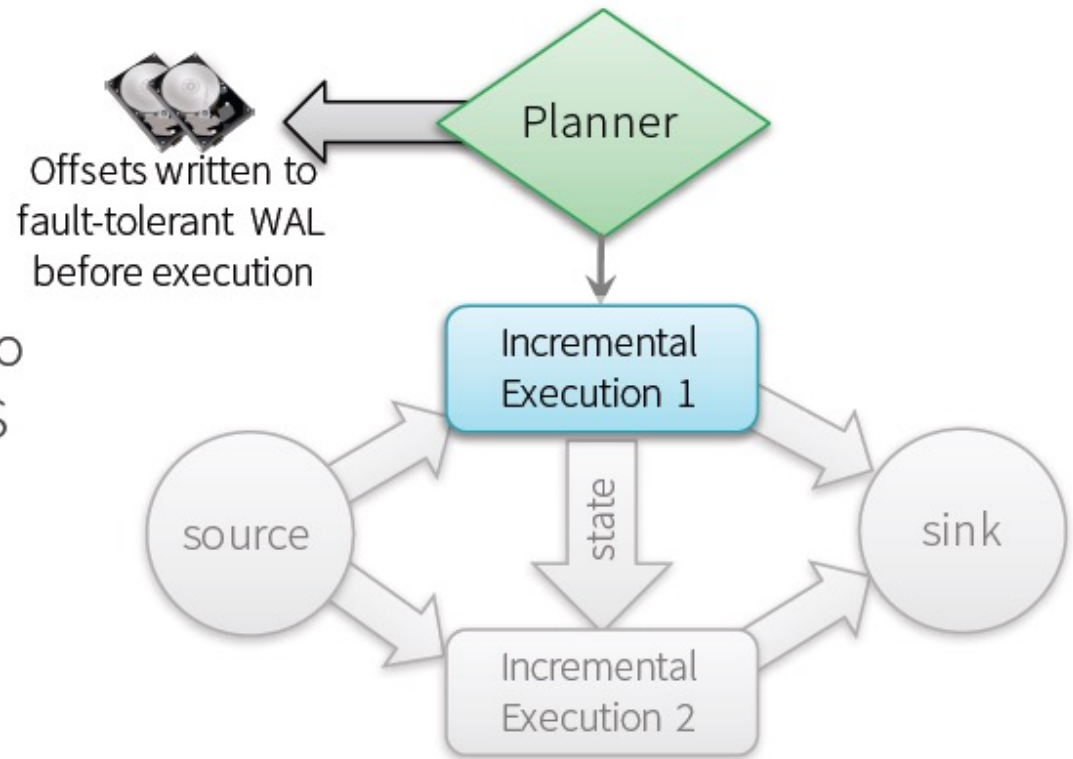
All data and metadata in the system needs to be recoverable / replayable



Fault-Tolerance

Fault-tolerant Planner

Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS

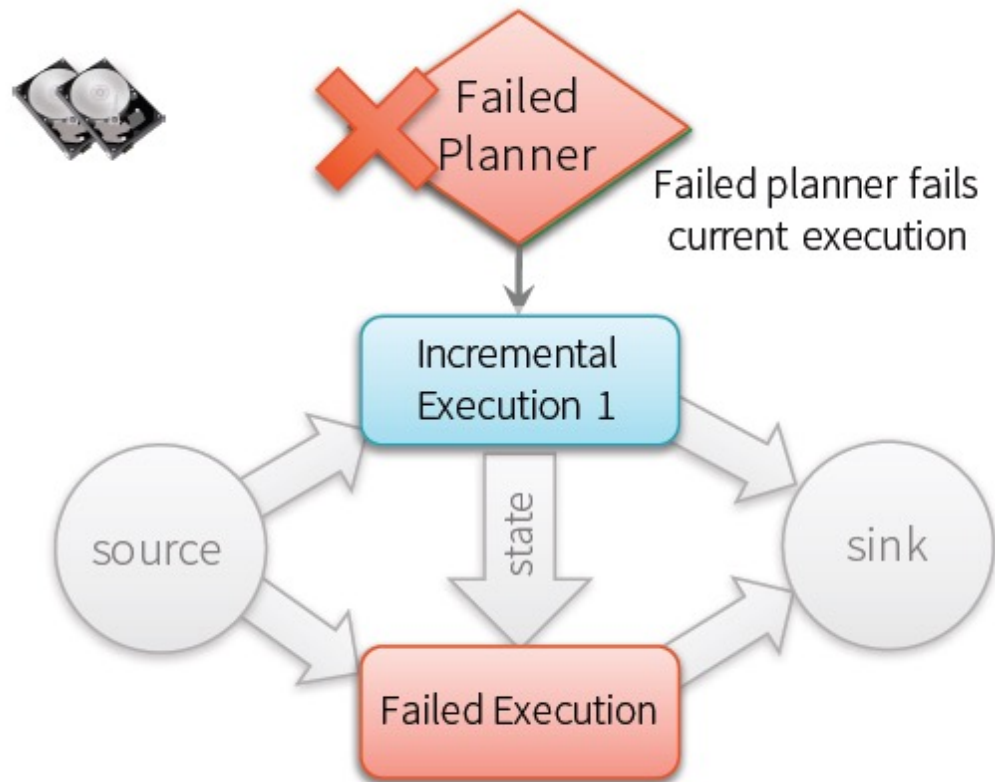


Fault-Tolerance



Fault-tolerant Planner

Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS

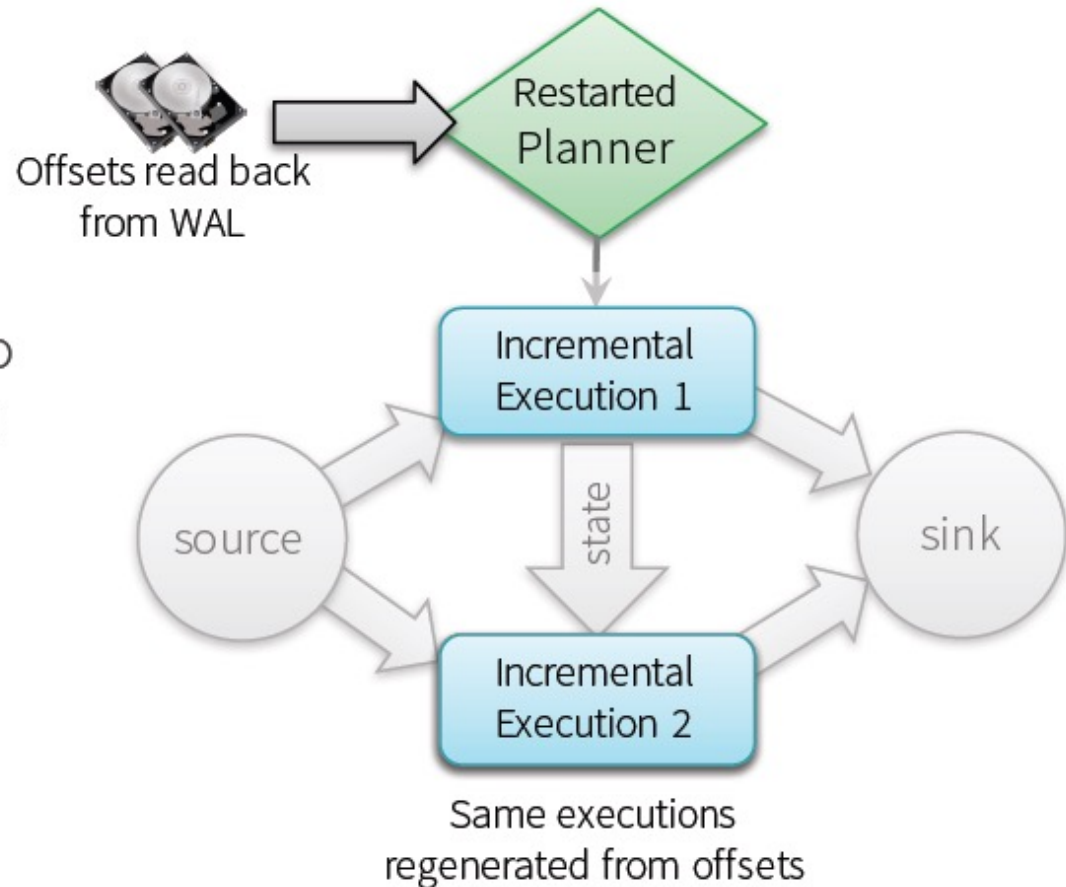


Fault-Tolerance

Fault-tolerant Planner

Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS

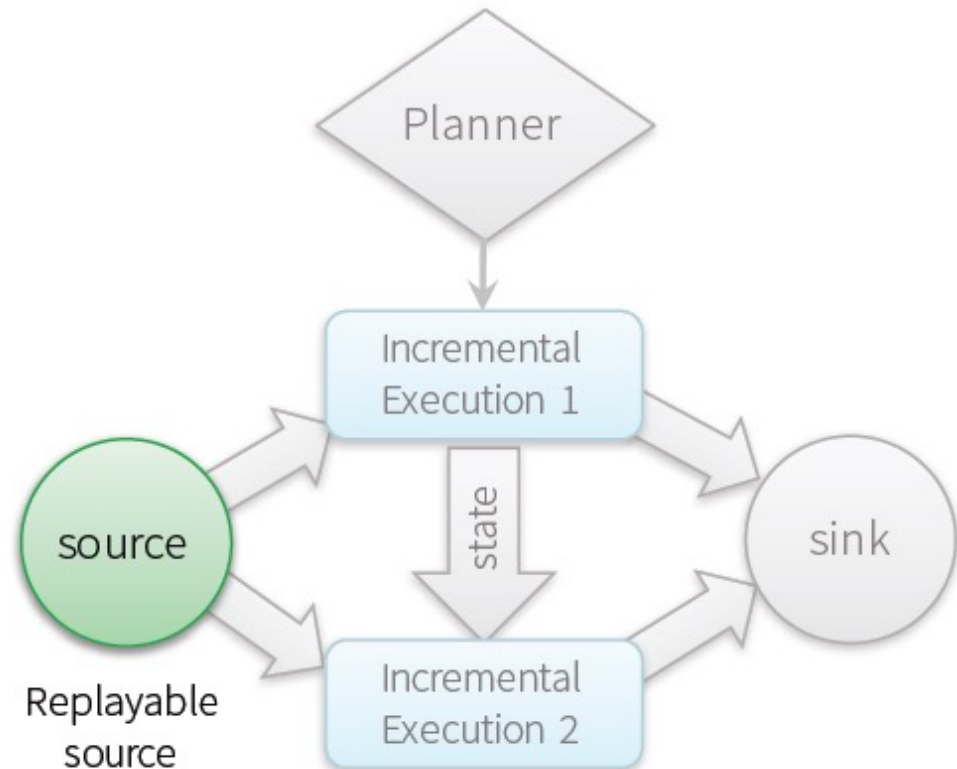
Reads log to recover from failures, and re-execute exact range of offsets



Fault-Tolerance

Fault-tolerant Sources

Structured streaming sources are by design replayable (e.g. Kafka, Kinesis, files) and generate the exactly same data given offsets recovered by planner

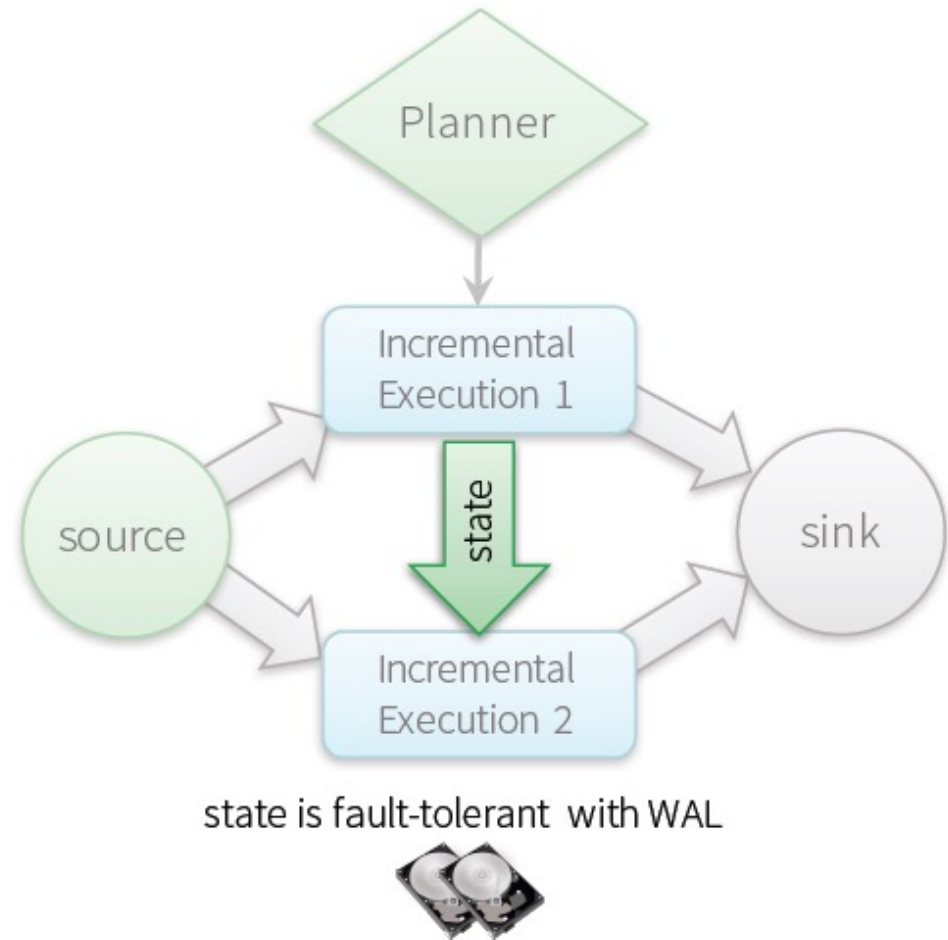


Fault-Tolerance

Fault-tolerant State

Intermediate "state data" is maintained in versioned, key-value maps in Spark workers, backed by HDFS

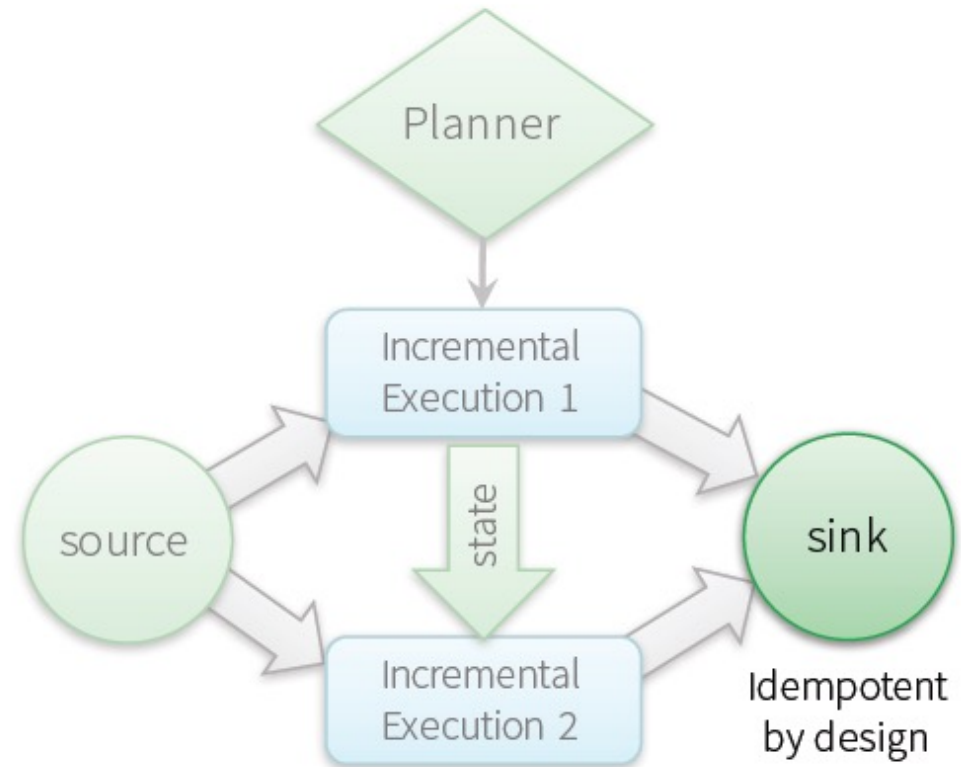
Planner makes sure "correct version" of state used to re-execute after failure



Fault-Tolerance

Fault-tolerant Sink

Sinks are by design idempotent, and handle re-executions to avoid double committing the output



Realizing Fault-Tolerance in Structured Streaming – A Summary

Offset Tracking in WAL

+

State Management

+

Fault-Tolerant Sources and Sinks

= End-to-End Exactly Once Guarantee

Support of Structured Streaming from Other Modules of Spark

- Interactive queries should just work
- Spark's data source API are being updated to support seamless streaming integration
 - Exactly once semantics **end-to-end**
 - Different Output modes (complete, delta, update-in-place)
- Machine Learning algorithms are being updated according to this new model