CARLSON SCHOOL OF MANAGEMENT
UNIVERSITY OF MINNESOTA

Spark Streaming: Part 2

MSBA 6330 Prof Liu

Learning Objectives

- Understand stateful multi-batch DStream operations and applications
- Understand streaming sources Kafka and Kinesis
- Understand fault tolerance in Spark Streaming
- Concept and applications of Structured Streaming

Introduction to Spark Streaming

STATEFUL MULTI-BATCH DSTREAM OPERATIONS

Multi-batch DStream Operations

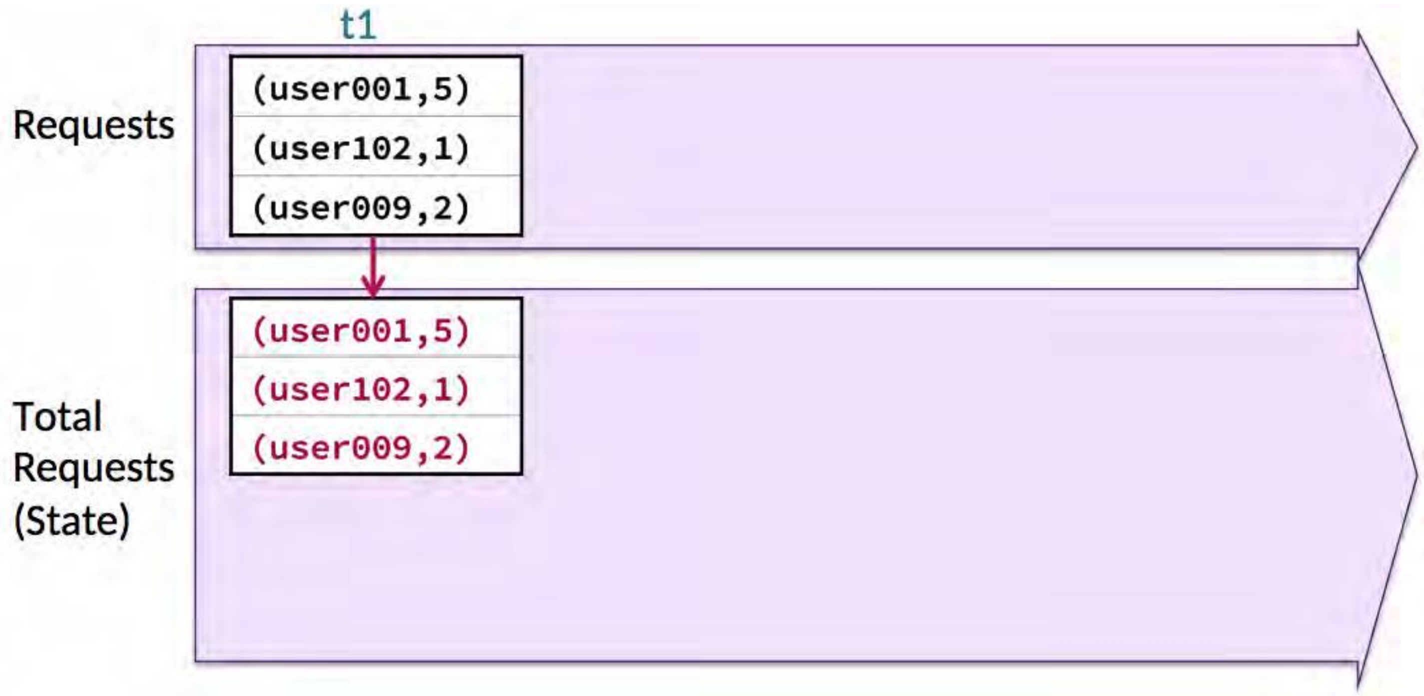
- Basic DStream operations analyze each batch individually
- Advanced operations allow you to analyze data collected across batches
  - **Slice**: allows you to operate on a collection of batches
  - **State**: allows you to perform cumulative operations
  - **Windows**: allows you to aggregate data across a sliding time period

Time Slicing & Remember

- `DStream.slice(fromTime,toTime)`
  - Returns a collection of batch RDDs based on data from the stream
- `StreamingContext.remember(duration)`
  - By default, input data is automatically cleared when no RDD's lineage depends on it
  - slice will return no data for time periods for which data has already been cleared
  - Use `remember` to keep data around longer

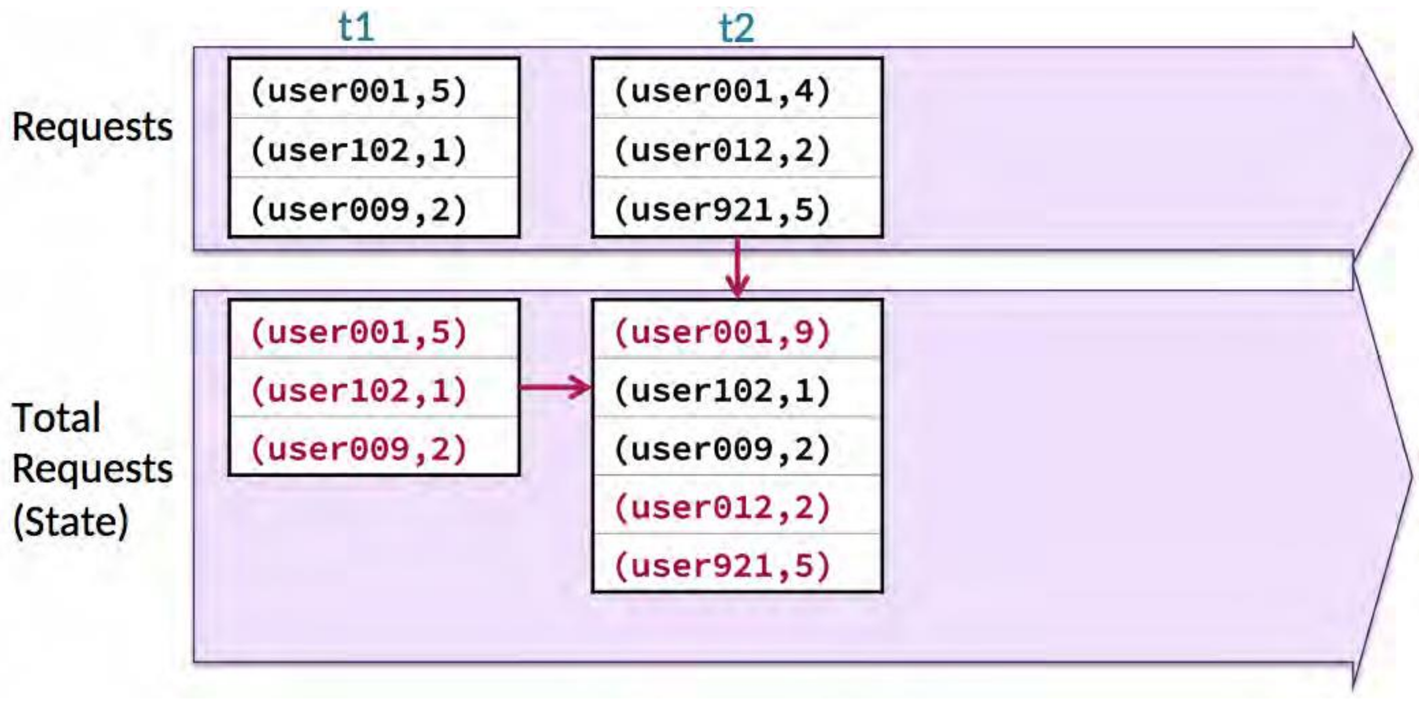
Use `updateStateByKey` to do stateful accumulations (1)

- Use the `updateStateByKey(updateFunc)` to do calculate cumulative states
  - Returns a new “state” DStream where the state for *each key* is updated by applying the given function on the previous state of the key and the new values of the key.
- Example: Total request count by User ID



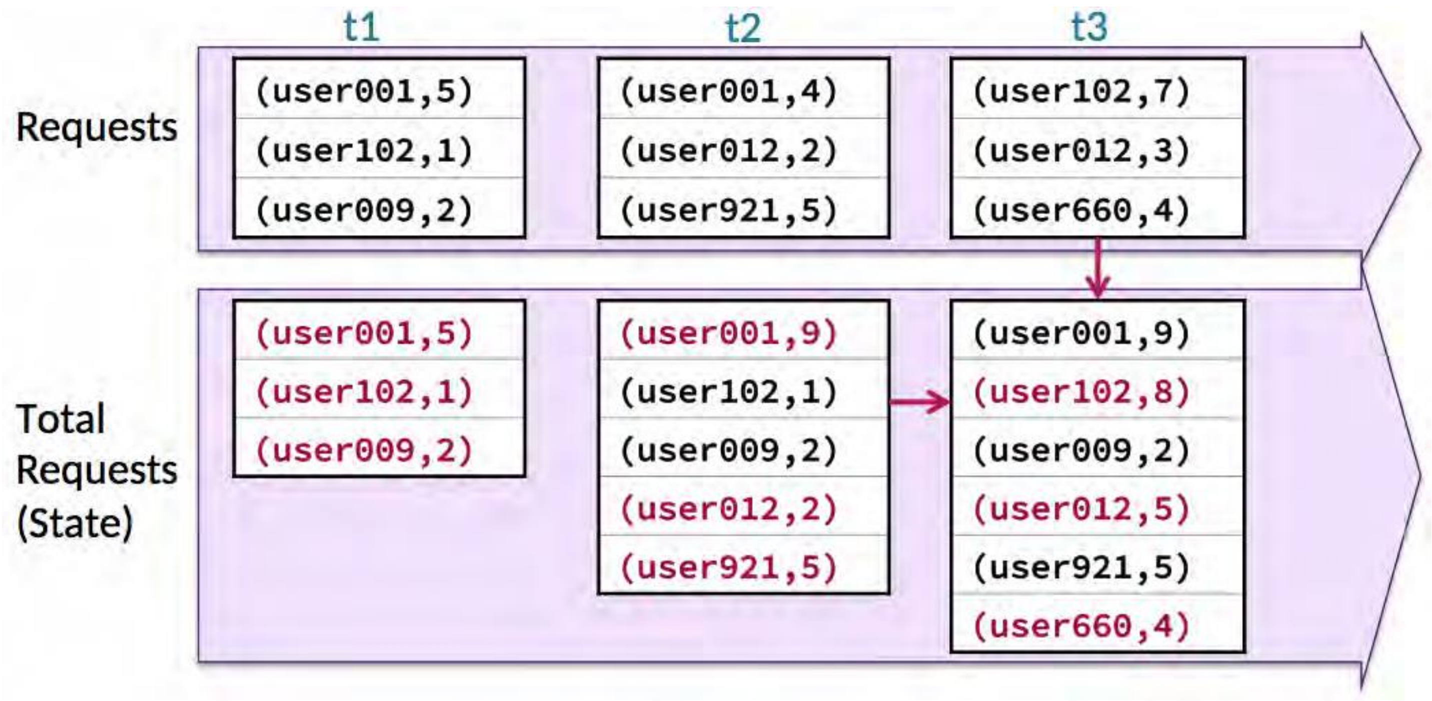
Use `updateStateByKey` to do stateful accumulations (2)

- `updateFunc(new_values, last_sum)`
  - E.g., for key "user001", the `new_values` is [4], the `last_sum` is 5, the new state is 9
  - for key "user012", the `new_values` is 2, the `last_sum` is None, the new state is 2



Use `updateStateByKey` to do stateful accumulations (3)

- The state DStream is a series of RDDs with key-value pairs



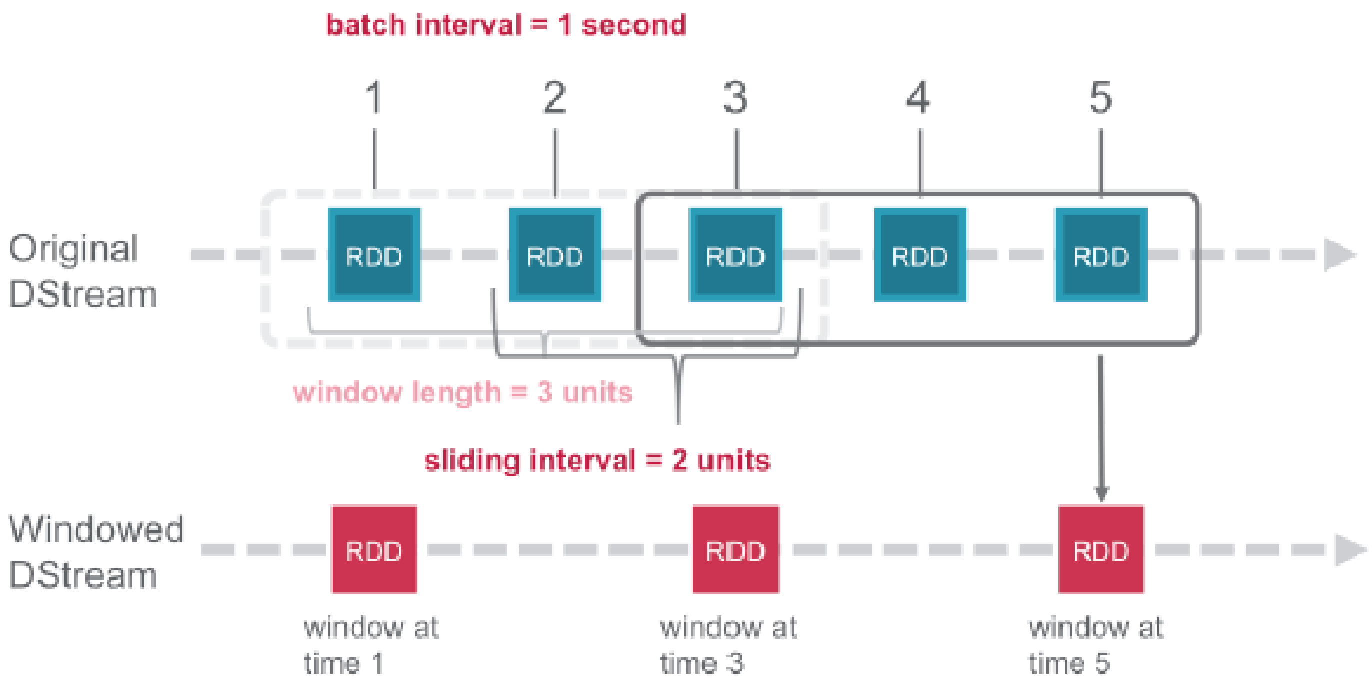
`updateStateByKey` Word Count Example

```
ssc.checkpoint("checkpoint") # check point directory must be configured

# RDD with initial state (key, value) pairs
initialStateRDD = sc.parallelize([(u'hello', 1), (u'world', 1)])

def updateFunc(new_values, last_sum):
    return sum(new_values) + (last_sum or 0) # add new values to last_sum

lines = ssc.socketTextStream('localhost', '9999')
# define a stateful DStream using UpdateStateByKey
running_counts = lines.flatMap(lambda line: line.split(" ")) \
 .map(lambda word: (word, 1)) \
 .updateStateByKey(updateFunc, initialRDD=initialStateRDD)

running_counts.pprint()
```

https://github.com/apache/spark/blob/v2.4.4/examples/src/main/python/streaming/stateful_network_wordcount.py

Using "window" to create a windowed DStream

- the "`window(windowLen, slideDur)`" operation span RDDs over a given duration
  – `[Dstream].window(3,2)`: returns a new DStream with each RDD being a sliding window of 3 batch intervals, computed every 2 batch intervals

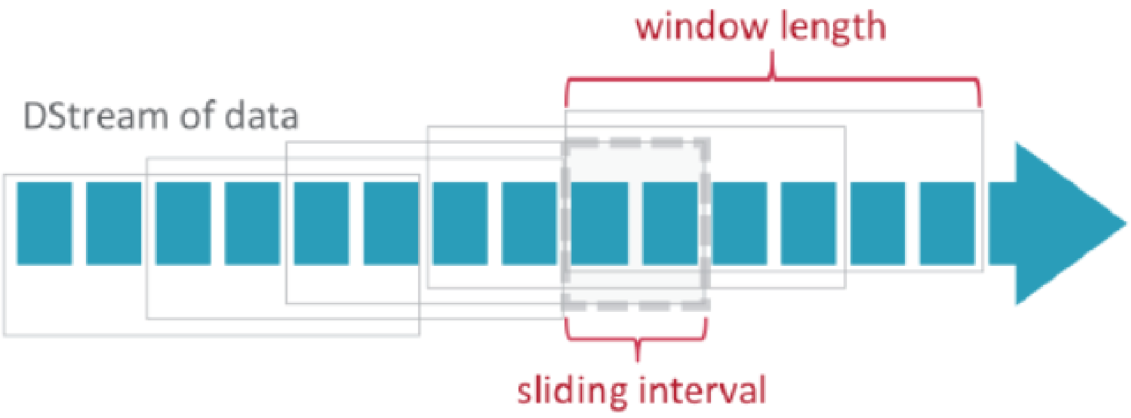


Window-based Transformations: `reduceByKeyAndWindow`

`reduceByKeyAndWindow(fun,invFun,windowDur,slidDur)`: Return a new DStream by applying incremental reduceByKey over a sliding window

- The `fun` is used to reduce the new values that entered the window
- The `invFunc` is used to reduce the new values that left the window

```
# dstream consists of RDDs of key-value pairs (word, 1)
func = lambda x, y: x + y
invFunc = lambda x, y: x - y
newDStream = dstream.reduceByKeyAndWindow(func, invFunc, 6, 2)
# word count over a sliding window of 6 batch intervals, computed every 2 intervals.
```



Window-based transformations

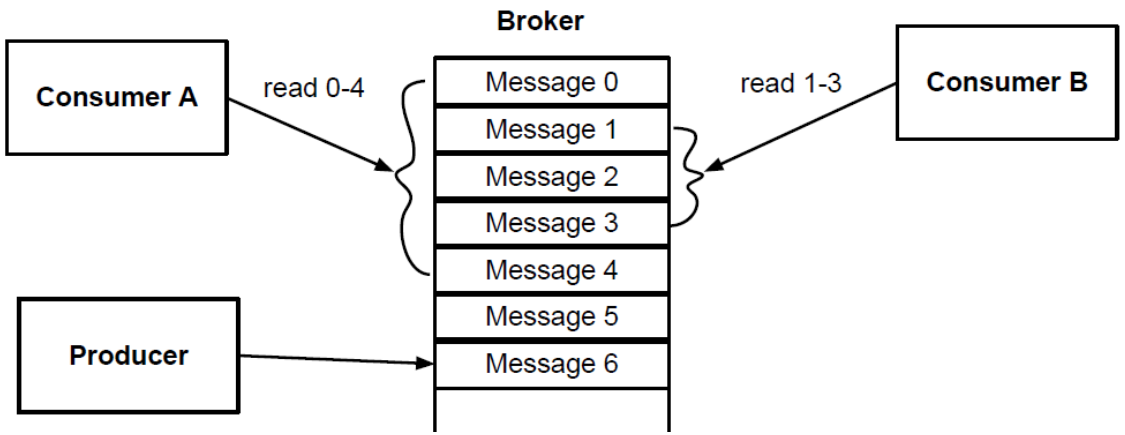
Window Operation	Description
`window(windowLen, slideDur)`	Returns new DStream computed based on windowed batches of source DStream
`countByWindow(windowDur, slideDur)`	Returns a sliding window count of elements in the stream
`reduceByWindow(func,invFunc, windowDur,slideDur)`	Returns a new single-element stream created by aggregating elements over sliding interval using `func`
`reduceByKeyAndWindow(func, invFunc,windowDur,slideDur)`	Returns a new DStream of (K,V) pairs from DStream of (K, V) pairs; aggregates using given reduce function `func` over batches of sliding window
`countByValueAndWindow(windowDur, slideDur)`	Returns new DStream of (K,V) pairs where value of each key is its frequency within a sliding window, it acts on DStreams of (k, v) pairs.

Introduction to Spark Streaming

INTEGRATION WITH STREAMING SOURCES: KAFKA AND KINESIS

Apache Kafka – A brief overview

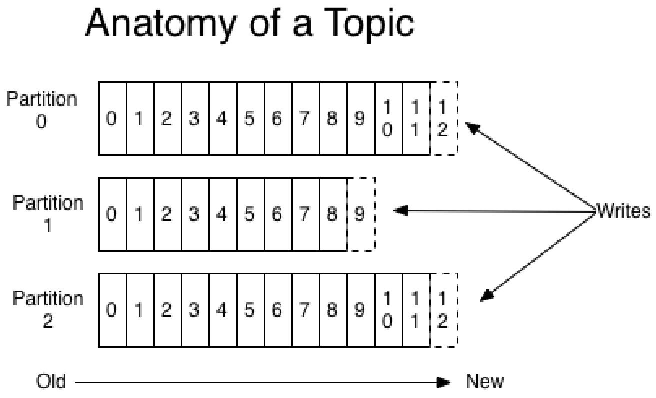
- Is a persistent, distributed, replicated, publish/subscription message broker system (with utilities for stream processing)
  - Originated from LinkedIn, written in Scala

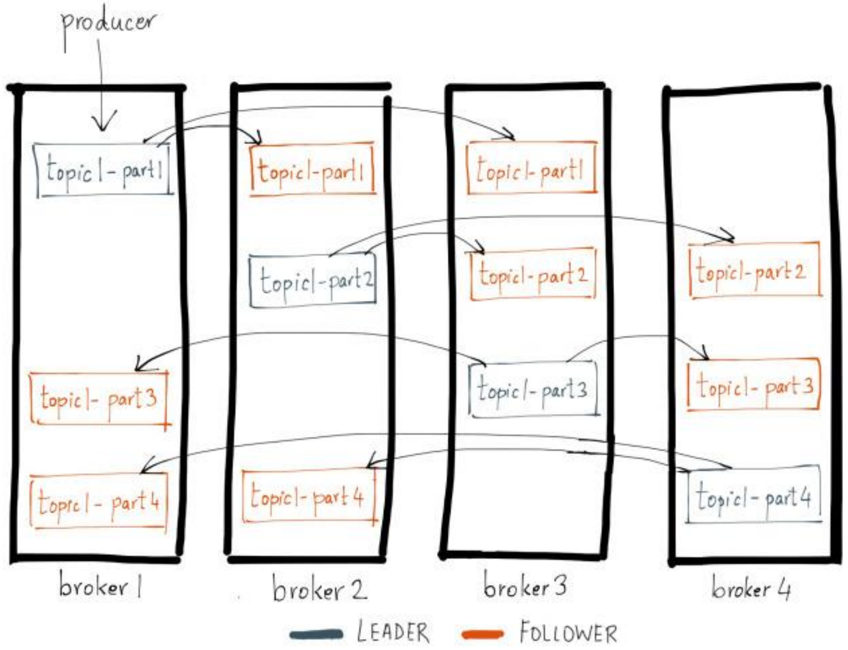


  - **Publishers** send messages to a cluster of **brokers (usually one per node)**, who persist the messages to disk
  - **Consumers** request a range of messages using an (offset, length) style API.

Apache Kafka – A brief overview (cont.)

- Messages are organized by **topics** (queues).
- Each topic is a collection of **partitions** (to allow parallelism)
- Partitions are replicated (to allow high availability)
- A broker contains some of the partitions for a topic
- Brokers are coordinated by **Zookeeper**





https://www.slideshare.net/mumrah/kafka-talk-tri-hug

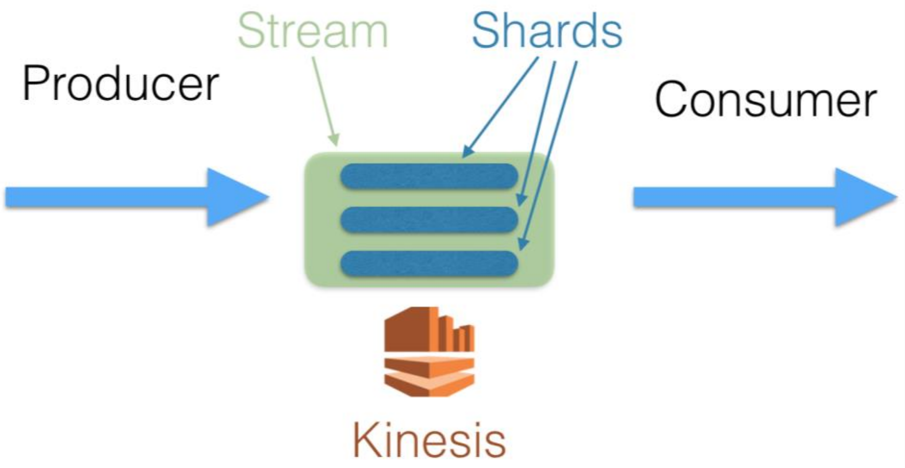
Kafka/Spark Streaming Integration

- [Two approaches] with different programming models, performance characteristics
  - (older) receiver based approach (deprecated in Spark 2.3.0)
  - (newer) direct approach without receiver

```
# receiver approach
from pyspark.streaming.kafka import KafkaUtils
kafkaStream = KafkaUtils.createStream(ssc,zkQuorum, groupId, {topic:numPartitions})
# zkQuorum: (hostname:port, hostname:port,…)
# direct approach
from pyspark.streaming.kafka import KafkaUtils
directKafkaStream = KafkaUtils.createDirectStream(ssc, [topic], \
 {"metadata.broker.list": brokers})
```

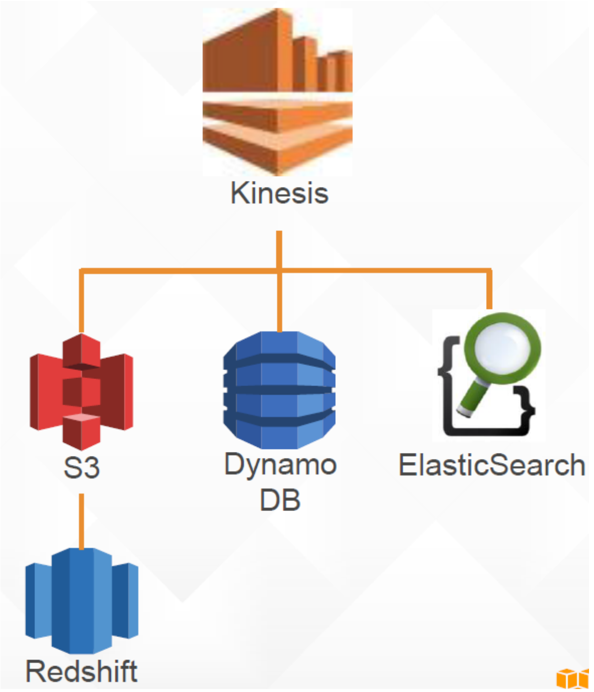
Amazon Kinesis – A brief overview

- **Streams** (like Kafka topics): named event stream stored for 24 hours by default. Each Kinesis data stream is a set of shards
- **Shards**: Each shard has a sequence of data records and has a fixed unit of capacity (5 transactions/sec for reads). Each record has a sequence number assigned by Kinesis data streams
- **Partition Key**: each data record has an associated partition key for determining which shard it belongs to
- **Producer** using a PUT call to write events to a stream;
- **Consumer** (commonly an **Amazon Kinesis application** running on a fleet of EC2 instances) uses Kinesis Client Library (KCL) to process records in each shard. KCL natively supports python but also supports other languages



Amazon Kinesis – A brief overview

- Use DynamoDB for state management
- Output of a Kinesis application can be input for another stream
- Integration with other AWS tools: S3, Redshift, DynamoDB, ElasticSearch



Kinesis/Spark Streaming Integration

- Kinesis receiver can create an input DStream using the Kinesis Client Library.
- To use it, you must add the spark-streaming-kinesis library as a dependency

```
from pyspark.streaming.kinesis import KinesisUtils, InitialPositionInStream

kinesisStream = KinesisUtils.createStream(
         streamingContext, [Kinesis app name], [Kinesis stream name], [endpoint URL],
         [region name], [initial position], [checkpoint interval],
         StorageLevel.MEMORY_AND_DISK_2)
```

Introduction to Spark Streaming

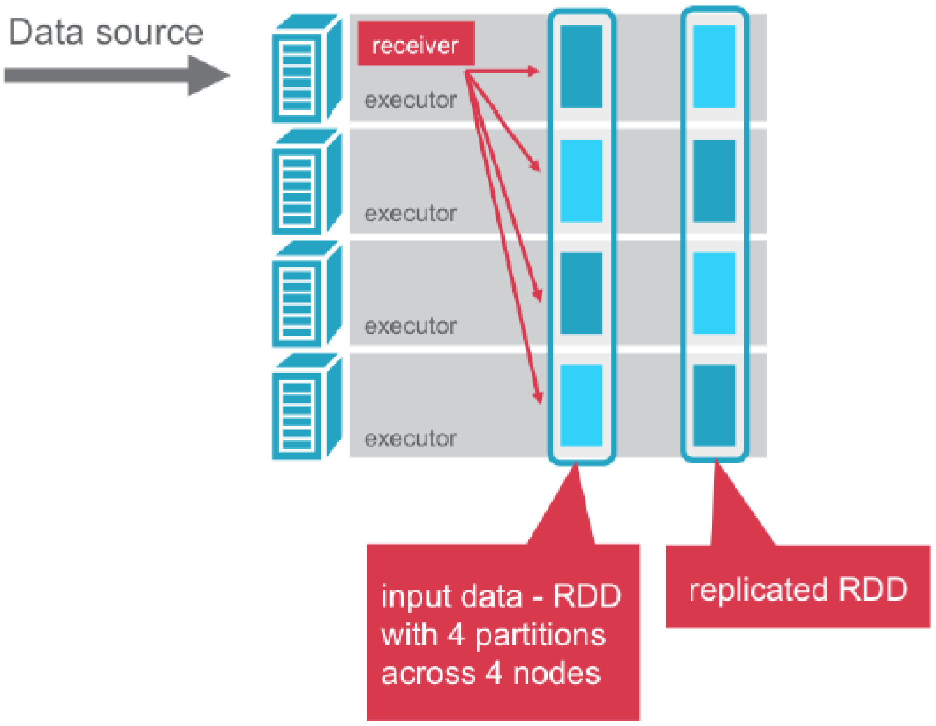
FAULT TOLERANCE IN SPARK STREAMING

Why Fault Tolerance for Streaming is Different

- Fault tolerance in RDDs
  - Each RDD remembers lineage
  - IF a RDD partition is lost, the partition will be recomputed based on lineage
  - Data in final transformed RDD always the same
  - Data comes from fault-tolerant systems (e.g. HDFS, S3) are also fault tolerant
- Not the case with Spark Streaming
  - Streaming source may not be fault-tolerant!
    - If you miss a message, you may not get it back

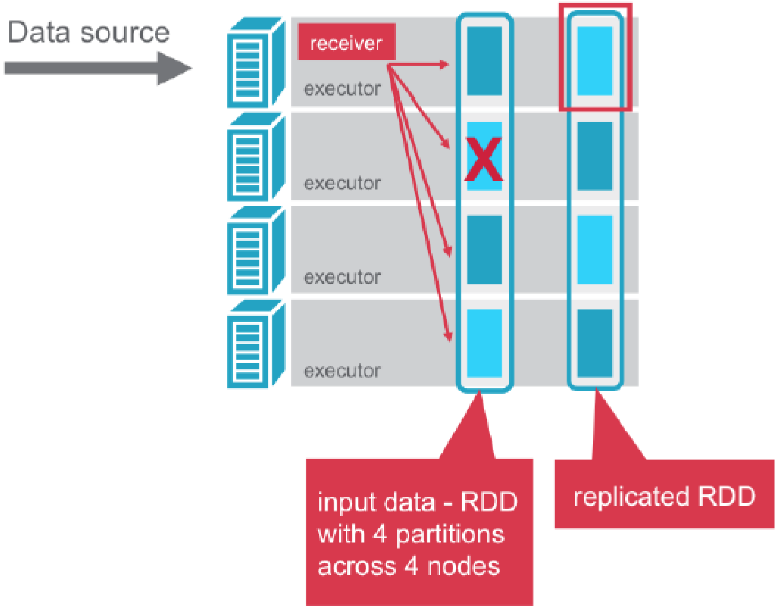
Fault Tolerance in Spark Streaming

- Spark Streaming launches receivers within an executor for each input source.
- The receiver receives input data that is saved as RDDs and replicated to other executors for fault tolerance. (default replication factor is 2)



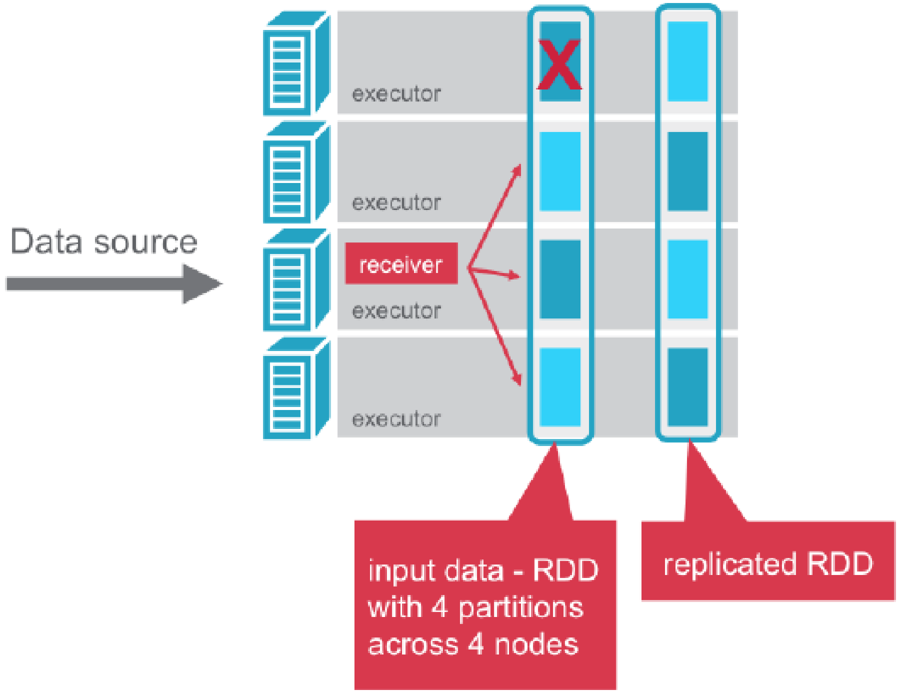
If a worker node fails

- If the data is from a network source (thus cannot be retrieved if data is lost) and one worker (#2) node fails
- The data still exists in node #1's memory.



If the receiver node fails

- There may be a loss of data that was received by the system but not yet replicated to other nodes
- The receiver will be started on a different node



Checkpointing for Driver Failures

- Checkpoint mechanism can periodically save data to a fault tolerant system
  - `ssc.checkpoint(path)`
- Two types of data are checkpointed:
  - *Metadata checkpointing* – save information defining the streaming computation, for recovery from driver failures
  - *Data checkpointing* – necessary if the computation is *stateful* (i.e. combine data across multiple batches, e.g. windowed operations)
- Checkpointing must be enabled if stateful transformations are used or recovery from driver failures are expected.

Introduction to Spark Streaming

STRUCTURED STREAMING

Section credits: Jules S. Damji's 2019 Spark+AI Summit Presentation "Writing Continuous Applications with Structured Streaming in PySpark"

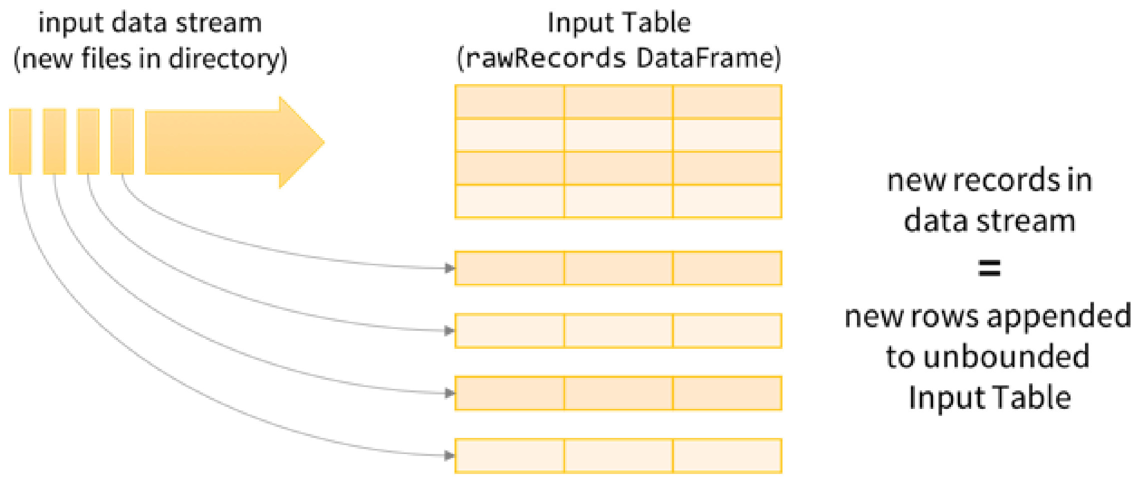
Preview: Structured Streaming

- Spark Structured Streaming is a new high-level API for streaming processing on the Spark SQL engine
  - Uses the **DataFrame** and Dataset API with streaming data
    - Support of Spark SQL data sources (json, parquet etc)
  - The ability to start/stop individual **queries** without needing to start/stop a separate `StreamingContext`
  - Support for continuous processing (vs micro-batches)
  - Support for event time (vs. processing time) aggregation

API documentation available through pyspark.sql.streaming

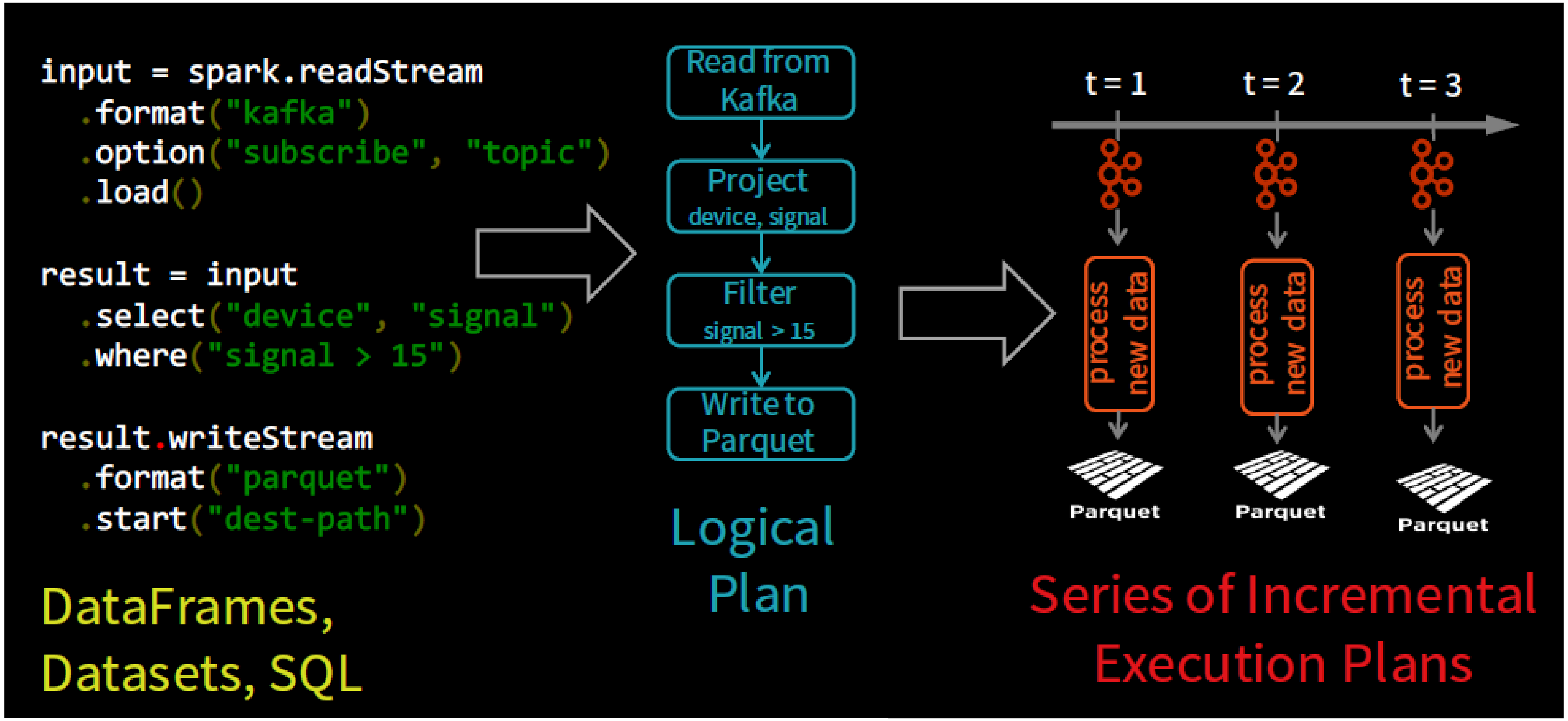
Treat Streams as Unbounded Tables

- You can write your Spark SQL queries
- Spark will continuously update the answer



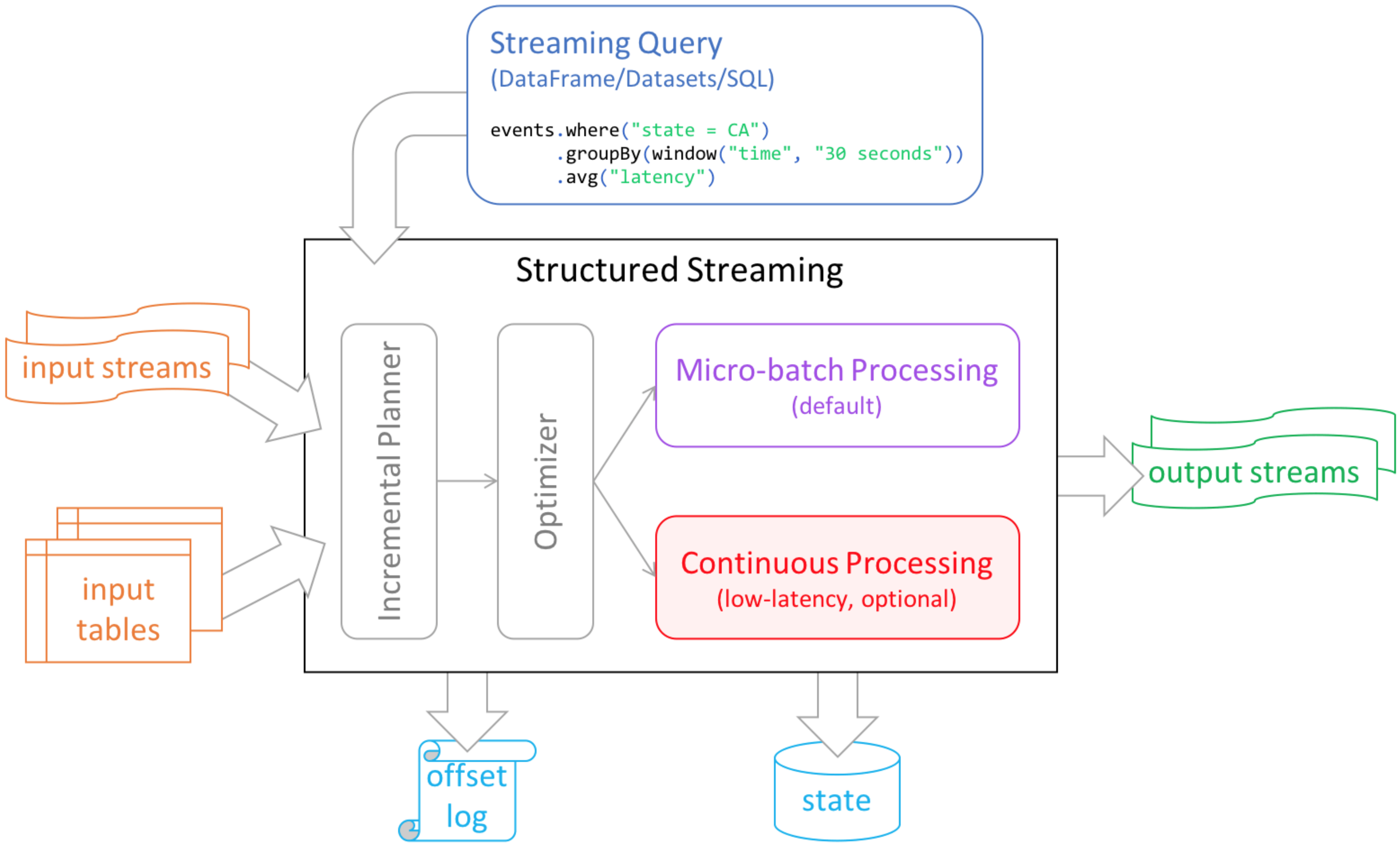
Structured Streaming Model
treat data streams as unbounded tables

Apache Spark automatically streamfies



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

Structured Streaming supports both micro-batch and continuous processing



Structured Streaming Example

```
from pyspark.sql import Trigger

spark.readStream
.format("kafka")
.option("subscribe", "input")
.load()

.groupBy("value.cast('string') as key")
.agg(count("*") as 'value’)

.writeStream()
.format("kafka")
.option("topic", "output")

.trigger("1 minute")
.outputMode("update")
.option("checkpointLocation", "…")
.withWatermark("timestamp","2 minutes")
.start()
```

Complete: output the whole answer each time
Update: output changed rows
Append: output new rows only

Build in support for Files/Kafka/Socket

Use DataFrame/SparkSQL to process data

Writer Stream outputs data in batches

Trigger: when to output (no trigger means as fast as possible)

Checkpoint location

watermark to drop very late events.