hy562: Advanced Topics in Databases

Programming with Data Streams

Instructor: Panagiotis Papadakos
Fall 2017

Computer Science Department
University of Crete
Agenda

General things
  – Stream Data
  – Streams Processing

Apache Kafka
  – Basics
  – Example

Apache Spark
  – Basics
  – Example
Stream Data

Stream data can come from:

- Devices
- Sensors
- Web sites
- Social media feeds
- Applications
- Infrastructure systems
Real time analysis

Batch processing gives great insights about what happened in the past
– But it lacks the ability to answer the question of what is happening right now

Process events as they arrive (efficiently and at scale)
– Website monitoring
– Network monitoring
– Web clicks
– Advertising
– Internet of things

Stream Processing

Increased demand for stream processing
– processing big volumes of data is not enough
Data has to be processed fast
– a firm can react to changing business conditions in real time

Stream processing is the real-time processing of data continuously and concurrently
– process data streams or sensor data (usually a high ratio of event throughput versus numbers of queries)

Complex event processing (CEP) utilizes event-by-event processing and aggregation (e.g. on potentially out-of-order events from a variety of sources – often with large numbers of rules or business logic)
– to infer events or patterns that suggest more complicated circumstances
Stream Analytics Pipeline

Event production → Event queuing & stream ingestion → Stream analytics → Storage & batch analysis → Presentation & action → Archiving for long term storage/batch analytics → Real-time dashboard → Automation to kick-off workflows

Stream Processing Frameworks

Kafka vs Spark

Kafka
- Low latency
- Easy to use event time support
- Well suited for certain types of tasks
- No need for special cluster manager
- Supports also simple topic-to-topic transformations, count elements by key, enrich a stream with data from another topic, run aggregations or real-time processing

Spark
- Throughput...
- Spark is mature/stable framework
- Popular platform for big data
- If event time is not relevant and latencies in the seconds range are acceptable
- The code used for batch applications can also be used for the streaming applications as the API is the same
Kafka vs Spark

Some performance results over the Yahoo streaming benchmark
- Emulates a simple advertisement application, where ad events are consumed from Kafka.
- Compute event-time windowed counts of ad campaigns that are “viewed.”
- Regarding throughput
- Focused on another streaming system (Flink)
- https://github.com/databricks/benchmarks/tree/master/streaming

- Kafka Streams has rather low throughput

Apache Kafka

An Apache project initially developed at LinkedIn
Kafka is a messaging and processing system
Designed to be fast, scalable, fault-tolerant and durable
Open source
Distributed publish-subscribe messaging system
Designed for processing of real time activity stream data
(e.g. logs, metrics collections)
  – Low latency (process records as they occur)
Written in Scala and Java (latest version 1.0.0)
https://kafka.apache.org/
Why Kafka?

In big data an enormous volume of data is used

Kafka helps in:

- Building *real-time streaming data pipelines* that can get data between systems and applications
- Building *real-time streaming applications* to react to the stream of data
  
  Real-time analytics
  
  Transform, react, aggregate, join real-time data flows
Kafka use cases

Metrics / KPIs gathering
– Aggregate statistics from many resources
Event sourcing
– Used with micro services and actor systems
Commit Log
– External commit log for distributed systems
– Replicated data between nodes, resync for nodes to restore state

Real-time data analytics, Stream processing, Log aggregation, Messaging, Click-stream tracking, Audit trail
Need for a messaging system

A messaging system is a system that is used for transferring data from one application to another ... so applications can focus on data ... and not how to share it

Kafka is a distributed publish-subscribe messaging system
– Publishers are called **Producers**
– Subscribers are called **Consumers**

**Messages are persisted in a topic**
Consumers can subscribe to one or more topic and consume all the messages in that topic
Kafka Benefits

- **Reliability**
  - Distributed, partitioned, replicated and fault-tolerant (load balancing in the case of failure)

- **Scalability**
  - Distributed system that scales quickly and easily without any downtime

- **Durability**
  - Distributed commit log
  - Messages persist on disk as fast as possible (intra-cluster replication)

- **Performance**
  - High throughput for both publishing and subscribing messages
  - Stable performance even with terabytes of messages
Kafka Basics – Records / Topics

A record has a key, value and timestamp
Kafka cluster retains all published records
  – Time based – configurable retention period
  – Size based
  – Compaction
It is available for consumption until discarded
  – Consumption speed not impacted by size
Kafka Basics – Records

Kafka clusters store a stream of records in categories called topics

- A topic is a feed name or category to which records are published (a named stream of records)
- Topics are always multi-subscriber (0 or many consumers)
- For each topic the Kafka cluster maintains a log
Kafka Basics – Partitions / Replicas

A topic log may have many partitions, spread on many nodes
Order of records guaranteed only inside each partition
A partition is an immutable record sequence
Each partition has a number of offset (a unique sequential id)
Replicas are backups of partitions based on a replication factor
– To tolerate $f$ failures need $2f + 1$ replicas
– A partition has a leader handling reads and writes and followers who take over if the leader dies (ISR – In Sync Replica)
Kafka Basics — Producers

Producers write to new offsets (tails of logs)
- Picks partition based on key (hash of keys)
- Other partitioners include round-robin, priority, etc.

Kafka is fast, due to sequential writes to filesystem
- No on disk mutable BTrees
- Data gets appended to end of the log and compacted
- Takes advantage of OS disk caching
- Multiple producers can write to different partitions
Kafka Basics — Consumers

Consumers are grouped in Consumer Group (unique id)
Each consumer group is a subscriber
Consumers read the logs at their own pace
Consumers groups remember offset where they left off
Each consumer group has its own offset
A record is delivered to one consumer
– Load balance of record consumption
– Multiple consumers can read from different partitions
Kafka Basics — Consumers

Messages in each partition log are read sequentially.

As the consumer makes progress, it commits the offsets of messages it has successfully processed.

When partition is reassigned to another consumer in the group the initial position is set to the last committed offset.

If the consumer crashes then the group member taking over the partition begins consumption from offset 1 (last committed offset).

The Log End Offset is the offset of the last message.

The high watermark is the offset of the last message successfully replicated.

Consumer reads until high watermark.
Kafka Basics – Brokers

Brokers are systems responsible for maintaining published data
  – Consists of many processes on many servers
  – They are stateless
  – Use zookeeper for maintaining cluster state
  – Each broker may have zero or more partitions per topic

If we have more than one brokers then we have a Kafka cluster

A Kafka cluster can be expanded without downtime
Zookeeper for fault tolerance

A high performance coordination service for distributed applications (open-source project)

- Makes cluster coordination
  - Fast
  - Scalable
- Runs on a cluster of nodes
- Locks and Synchronization
- Naming Service
- Group Membership
- Ensemble selects a leader
  (re-elected in case of failure)
- Reliability (no single-point of failure)
- Notifies producers/consumers about presence or failure of a new broker

http://zookeeper.apache.org
Kafka Batching - Compression

Kafka producers support record batching
  – Good for network IO
  – Speeds up throughput drastically
  – But with more latency

Kafka provides end-to-end batch compression
  – Instead of compressing a record at a time
  – Compress a batch of records
  – Kafka supports GZIP, Snappy and LZ4 compression protocols.
Kafka Architecture

Apache

Zookeeper (Core Dependency)

Input Systems → Kafka Producers → Kafka → Kafka Consumers → Output Systems

Apache Core KAFKA
Kafka Streams

Kafka stream - an unbounded continuously updating dataset
- Replayable, and fault tolerance sequence of immutable data records (key-value)
- Record stream (KStream) — independent key-value pairs
- Changelog stream (KTable) — updates to earlier records with the same key
- Write results back to Kafka or send the final output to an external system

Kafka streams address a lot of difficult problems in stream processing
- Event-at-a-time processing (not microbatch) with millisecond latency
- Stateful processing including distributed joins and aggregations
- A convenient DSL
- Windowing with out-of-order data
- Distributed processing and fault-tolerance with fast-failover
- No-downtime rolling deployments
- No separate cluster requirements (integrated with Kafka)
A stream processor is a node in the processor topology (a processing step)

- Source processor (a processor without any upstream processors)
- Sink processor (a processor without any downstream processor)

https://dzone.com/articles/kafka-streams-more-than-just-dumb-storage
Apache Kafka
Kafka core APIs

Producer API
- Allows an application to publish a stream of records to one or more Kafka topics

Consumer API
- Allows an application to subscribe to one or more topics and process the stream of records produced to them

Streams API
- Allows an application to act as a stream processor (transform input streams to output streams)
- Consuming an input stream from one or more topics
- Producing an output stream to one or more topics

Connector API
- Reusable producers or consumers that connect Kafka topics to existing applications or data systems (e.g. a connector to a relational database)

https://kafka.apache.org/documentation/#api
Download Kafka — Run server

Download kafka binary from https://kafka.apache.org/downloads

Kafka needs zookeeper to be running https://zookeeper.apache.org/

Untar the binary and start the server

# Start kafka server
sudo ./kafka/bin/kafka-server-start.sh kafka/config/server.properties
Run Console Producers/Consumers

# Create a topic - Producer
./kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic hy562Topic

# Show available topics
./kafka/bin/kafka-topics.sh --list -zookeeper localhost:2181

# Describe a topic
./kafka/bin/kafka-topics.sh --describe -zookeeper localhost:2181 --topic hy562Topic

# Create a consumer that will bring topics from producer
./kafka/bin/kafka-console-consumer.sh -zookeeper localhost:2181 --topic hy562Topic --from-beginning
Creating our own topic

#!/usr/bin/env bash

## Create topics
kafka/bin/kafka-topics.sh --create \
  --replication-factor 1 \
  --partitions 13 \
  --topic hy562-topic \
  --zookeeper localhost:2181

## List created topics
kafka/bin/kafka-topics.sh --list \
  --zookeeper localhost:2181
package kafka.example.producer;

// Some imports
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.serialization.LongSerializer; // Used for key
import org.apache.kafka.common.serialization.StringSerializer; // Used for value

public class KafkaProducerExample {

    private final static String TOPIC = "my-example-topic"; // topic
    private final static String BOOTSTRAP_SERVERS = "localhost:9092"; // These are the servers
KafkaProducerExample (2/4)

// Create a new producer
private static Producer<Long, String> createProducer() {
    Properties props = new Properties();  // Use a properties class
    // Populate it with appropriate configs, servers and key-value types
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);  // Used for logging of specific client
    props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
    // Implements Kafka serializer class for Kafka for record keys
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
               LongSerializer.class.getName());
    // Implements Kafka serializer class for Kafka for record values
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
               StringSerializer.class.getName());
    // Now create the producer
    return new KafkaProducer<>(props);
}
KafkaProducerExample (3/4)

// Run the producer and send sendMessageCount records synchronously
static void runProducer(final int sendMessageCount) throws Exception {
    // Create the producer
    final Producer<Long, String> producer = createProducer();
    long time = System.currentTimeMillis();
    try {
        for (long index = time; index < time + sendMessageCount; index++) {
            // Create a new producer record
            final ProducerRecord<Long, String> record = new ProducerRecord<>(TOPIC, index, "Hello times " + index);
            // Send the record. Returns a java future
            // Record metadata holds partition number and offset
            RecordMetadata metadata = producer.send(record).get();
            long elapsedTime = System.currentTimeMillis() - time;
            System.out.printf("sent record(key=%s value=%s) meta(partition=%d, offset=%d) time=%d\n", record.key(), record.value(), metadata.partition(), metadata.offset(), elapsedTime);
        }
    } finally {
        producer.flush();
        producer.close();
    }
}
public static void main(String... args) throws Exception {
    if (args.length == 0) {
        runProducer(5);
    } else {
        runProducer(Integer.parseInt(args[0]));
    }
}
package kafka.example.consumer;

import java.util.Collections;
import java.util.Properties;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.LongDeserializer;
import org.apache.kafka.common.serialization.StringDeserializer;

public class KafkaConsumerExample {

    private final static String TOPIC = "hy562-topic";
    private final static String BOOTSTRAP_SERVERS = "localhost:9092";
private static Consumer<Long, String> createConsumer() {
    final Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaConsumerExample");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                LongDeserializer.class.getName()); // Key Deserializer
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName()); // Value Deserializer

    // Create the consumer using props.
    final Consumer<Long, String> consumer = new KafkaConsumer<>(props);

    // Subscribe to the topic!
    consumer.subscribe(Collections.singletonList(TOPIC));
    return consumer;
}
KafkaConsumerExample (3/3)

```java
static void runConsumer() throws InterruptedException {
    final Consumer<Long, String> consumer = createConsumer();

    final int giveUp = 100;
    int noRecordsCount = 0;

    while (true) {
        final ConsumerRecords<Long, String> consumerRecords
            = consumer.poll(1000);
        if (consumerRecords.count() == 0) {
            noRecordsCount++;
            if (noRecordsCount > giveUp) {
                break;
            } else {
                continue;
            }
        }

        consumerRecords.forEach(record -> {
            System.out.printf("Consumer Record: (%d, %s, %d, %d)\n",
                record.key(), record.value(),
                record.partition(), record.offset());
        });

        consumer.commitAsync();
    }

    consumer.close();
    System.out.println("DONE");
}
```
Kafka Stream Processor API

Provide ways to enable processing of data that is consumed from Kafka and will be written back to Kafka

High-level Kafka Streams DSL

Lower-level processor that provides APIs for data processing, composable processing and local state storage
Kafka Stream Processor API

Stateless transformations (do not depend on state)

- branch
- filter
- map
- mapValue
- flatMap
- flatMapValues
- groupBy
- groupByKey
- more...

https://kafka.apache.org/10/documentation/streams/developer-guide
Kafka Stream Processor API

Stateful transformations

- depend on state for processing inputs and producing outputs
- e.g. in aggregating operations, a windowing state store stores aggregation results per window
- e.g. in join operations, a windowing state store is used to store all the records received so far within the defined window boundary
- Are fault-tolerant (fully restored state after failure)

https://kafka.apache.org/10/documentation/streams/developer-guide
Kafka Stream Processor API

Stateful transformations

- aggregate
- aggregate (windowed)
- count
- count (windowed)
- reduce
- reduce (windowed)

https://kafka.apache.org/10/documentation/streams/developer-guide

join and their variations (outerJoin, innerJoin, leftJoin)
- different semantics for sliding windows KStream - KStream, KStream – KTable join and KTable – KTable

https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Join+Semantics
Kafka Stream Processor API

- **KStream**
  - Stream-stream joins: windowed
  - Stream-table joins: non-windowed
  - Can be stateful or stateless
  - `join()`, `leftJoin()`, `outerJoin()`, `groupBy()`, `groupByKey()`, `transform()`, `transformValues()`

- **KTable**
  - Table-table joins: non-windowed
  - Can be stateful or stateless
  - `join()`, `leftJoin()`, `outerJoin()`

- **KGroupedStream**
  - Can be windowed or non-windowed
  - `groupBy()`, `count()`, `reduce()`

- **KGroupedTable**
  - Can be windowed or non-windowed
  - `groupBy()`, `count()`, `reduce()`

**Legend**
- **GlobalKTable**
  - no direct operations
- **Stateful operations**
- **Stateless operations**

**Process**
- `process()`
package kafka.example.kstreams;
import java.util.Arrays;
import java.util.Properties;
import java.util.regex.Pattern;
import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
public class WordCount {

    public static void main(final String[] args) throws Exception {
        final String bootstrapServers = "localhost:9092";
        final Properties streamsConfiguration = new Properties();
        streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-example"); // Unique name
        streamsConfiguration.put(StreamsConfig.CLIENT_ID_CONFIG, "wordcount-example-client"); // Where to find Kafka broker(s).
        streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        // Specify default (de)serializers for record keys and for record values.
        streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.Long().getClass().getName());
        streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
        // Records should be flushed every 10 seconds
        streamsConfiguration.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 10 * 1000);
        // For illustrative purposes we disable record caches
        streamsConfiguration.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
    }
}
// Set up serializers and deserializers, which we will use for overriding the default serdes
// specified above.
final Serde<String> stringSerde = Serdes.String();
final Serde<Long> longSerde = Serdes.Long();

// In the subsequent lines we define the processing topology of the Streams application.
final KStreamBuilder builder = new KStreamBuilder();
final KStream<Long, String> textLines = builder.stream(longSerde, stringSerde, "hy562-topic");

final Pattern pattern = Pattern.compile("\W+", Pattern.UNICODE_CHARACTER_CLASS);
final KTable<String, Long> wordCounts = textLines
    .flatMapValues(value +> Arrays.asList(pattern.split(value.toLowerCase())))
    // Count the occurrences of each word (record key).
    // This will change the stream type from `KStream<Long, String>` to `KTable<String, Long>`
    // (word → count). In the `count` op we must provide a name for the resulting KTable,
    // which will be used to name e.g. its associated state store and changelog topic.
    .groupBy((key, word) +> word)
    .count("Counts");

// Write the `KStream<String, Long>` to the output topic.
wordCounts.to(stringSerde, longSerde, "WordsWithCountsTopic");

// Now that we have finished the definition of the processing topology we can actually run
// it via `start()`. The Streams application as a whole can be launched just like any
// normal Java application that has a `main()` method.
final KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
// Always (and unconditionally) clean local state prior to starting the processing topology.
streams.start();

// Add shutdown hook to respond to SIGTERM and gracefully close Kafka Streams
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
References - Kafka

- Kafka documentation
- Confluent blog
- Creating Stream Applications videos
  - Part 1, Part 2, Part 3, Part 4
- Cloudurable tutorials
- An Introduction to Apache Kafka (Dzone)
- Tutorials Point – Apache Kafka
- Apache hub with papers and presentations
- Introduction to Kafka Streams
- https://github.com/confuentinc/examples
- https://bitbucket.org/papadako/kafka-hy562-example
Spark Streaming

Extension of the core Spark API

Spark streaming receives live input data streams and divides the data into batches (micro-batching)

- Batches are then processed by the spark engine to create the final stream of data
- Can use most RDD transformations
- Also DataFrame/SQL and MLlib operations
DStream

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

A StreamingContext object has to be created (batch interval)

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

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

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

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

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

**Use Case:** Converting a stream of lines to words by applying the operation flatMap on each RDD in the “lines DStream”.

```
lines DStream  lines from time 0 to 1  lines from time 1 to 2  lines from time 2 to 3  lines from time 3 to 4
  \------->          \------->          \------->          \------>
  flatMap operation

words DStream  words from time 0 to 1  words from time 1 to 2  words from time 2 to 3  words from time 3 to 4
  \------->          \------->          \------->          \------>
```
// Create a DStream that will connect to a server
// listening on a TCP socket, say <IP>:9990
val ssc = new StreamingContext(conf, Seconds(5))
val lines = ssc.socketTextStream("<Some_IP>", 9990)

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

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

• Besides sockets, the StreamingContext API provides methods for creating DStreams from files.

• Reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.)

• Spark Streaming will monitor a directory and process any files created in that directory.
Input DStream and Receivers

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

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

A streaming application must be resilient to failures

Two types of check-points

• **Metadata check-pointing**
  
  Store information regarding the streaming computation to HDFS
  
  Recovery from failures - configuration, operations, incomplete batches

• **Data check-pointing**
  
  Store generated RDDs to HDFS
  
  This is necessary in some stateful transformations
Transformations on DStreams

• DStreams support most of the RDD transformations

  map  
  flatMap  
  filter  
  repartition  
  count  
  reduce  
  countByValue  
  union  
  reduceByKey  
  join  
  cogroup

• Also introduces special transformations related to state & windows
Stateless vs Stateful Operations

• By design streaming operators are stateless
  • they know nothing about any previous batches

• Stateful operations have a dependency on previous batches of data
  • continuously accumulate metadata overtime
  • data check-pointing is used for saving the generated RDDs to a reliable stage
## DStreams transformations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(func)</code></td>
<td>Return a new DStream by passing each element of the source DStream through a function <code>func</code>.</td>
</tr>
<tr>
<td><code>flatMap(func)</code></td>
<td>Similar to <code>map</code>, but each input item can be mapped to 0 or more output items.</td>
</tr>
<tr>
<td><code>filter(func)</code></td>
<td>Return a new DStream by selecting only the records of the source DStream on which <code>func</code> returns true.</td>
</tr>
<tr>
<td><code>repartition(numPartitions)</code></td>
<td>Changes the level of parallelism in this DStream by creating more or fewer partitions.</td>
</tr>
<tr>
<td><code>union(otherStream)</code></td>
<td>Return a new DStream that contains the union of the elements in the source DStream and <code>otherDStream</code>.</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.</td>
</tr>
<tr>
<td><code>reduce(func)</code></td>
<td>Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function <code>func</code> (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.</td>
</tr>
</tbody>
</table>
DStreams transformations (cont.)

**countByValue()**

When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.

**reduceByKey(func, [numTasks])**

When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. Note: By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property spark.default.parallelism) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks.

**join(otherStream, [numTask])**

When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.

**cogroup(otherStream, [numTask])**

When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
DStreams transformations (cont.)

transform(func)

Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.

updateStateByKey(func)

Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.
UpdateStateByKey Operation

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

Three requirements:

• Define the state - The state can be an arbitrary data type

• Define the “state update function” used for updating the current state using the previous state and the new values from an input stream

• Regardless of whether they have new data in a batch or not
  - If the update function returns None then the key-value pair will be eliminated.

• Requires check-pointing to be configured
UpdateStateByKey Example

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

runningCounts.print()

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

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

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

3. **apache spark**
   - Time: t3
     - (this,1)
     - (is,2)
     - (fast,1)
     - (apache,1)
     - (tutorial,1)
     - (spark,3)
Transform Operation

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

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

Example:

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

Windowed computations apply transformations over a sliding window of data.

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

[Diagram showing original and windowed DStream with time stamps and window-based operation]
Window Operations

Any window operation needs to specify two parameters:

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

window(windowLength, slideInterval)

Return a new DStream which is computed based on windowed batches of the source DStream.

countByWindow(windowLength, slideInterval)

Return a sliding window count of elements in the stream.

reduceByWindow(func, windowLength, slideInterval)

Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using func. The function should be associative and commutative so that it can be computed correctly in parallel.

reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])

When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function func over batches in a sliding window. Note: By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property spark.default.parallelism) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks.
Window transformations

reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])

A more efficient version of the above reduceByKeyAndWindow() where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enters the sliding window, and “inverse reducing” the old data that leaves the window. An example would be that of “adding” and “subtracting” counts of keys as the window slides. However, it is applicable only to “invertible reduce functions”, that is, those reduce functions which have a corresponding “inverse reduce” function (taken as parameter invFunc). Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument. Note that checkpointing must be enabled for using this operation.

countByValueAndWindow(windowLength, slideInterval, [numTasks])

When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument.
Join Operations

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

```scala
val s1: DStream[String, String] = ... //
val s2: DStream[String, String] = ... //
val jS1 = s1.join(s2)
val jS2 = s1.leftOuterJoin(s2)
val jS3 = s1.rightOuterJoin(s2)
val jS4 = s1.fullOuterJoin(s2)

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

We can also join a dataset with a windowed stream

```scala
val dataset: RDD[String, String] = ... //
val wS = stream.window(Seconds(20))
val jS = wS.transform { rdd ⇒ rdd.join(dataset) }
```

The dataset can be dynamically changed, since transform is evaluated in every batch interval and will use the current dataset that dataset points to
Output operations

print()

Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging.

saveAsTextFiles(prefix, [suffix])

Save this DStream's contents as text files. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".

saveAsObjectFiles(prefix, [suffix])

Save this DStream's contents as SequenceFiles of serialized Java objects. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".

saveAsHadoopFiles(prefix, [suffix])

Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".

foreachRDD(func)

The most generic output operator that applies a function, func, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function func is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.
Performance notes on foreachRDD

foreachRDD allows data to be sent out to external systems

Output operations are executed lazily (RDD actions inside the DStream output operations force the processing of data)

Output operations are executed one-at-a-time

```scala
// Serialization errors
// Have to be sent from the driver to the worker
dstream.foreachRDD { rdd ⇒
    // executed in the driver
    val conn = createNewConnection()
    rdd.foreach { record ⇒
        // executed at the worker
        conn.send(record)
    }
    conn.close()
}
```

```scala
// Inefficient resource management
// create a connection for each record
dstream.foreachRDD { rdd ⇒
    rdd.foreach { record ⇒
        val conn = createNewConnection()
        conn.send(record)
        conn.close()
    }
}
```
Performance notes on foreachRDD

// This is ok
dstream.foreachRDD { rdd ⇒
  rdd.foreachPartition {partRecs ⇒
    // create a connection for the
    // records in the partition
    val conn = createNewConnection()
    partRecs.foreach( record ⇒
      conn.send(record))
    conn.close()
  }
}

// This is even better
// Pool of connections that can be used
// for different batches
dstream.foreachRDD { rdd ⇒
  rdd.foreachPartition {partRecs ⇒
    // static, lazily pool of connections
    // for different batches of RDDs
    val conn = ConnectionPool.getConnection()
    partRecs.foreach( record ⇒
      conn.send(record))
    conn.close()
    ConnectionPool.returnConnection(conn)
  }
}
Using Kafka topics with spark (1/3)
Word count example

package spark

import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.SparkConf
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.kafka010._
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.kafka.common.serialization.LongDeserializer

object ScalaKafkaWordCount {
  def main(args: Array[String]) {
    // Create context with 2 second batch interval
    val sparkConf = new SparkConf().setAppName("ScalaKafkaWordCount")
    val ssc = new StreamingContext(sparkConf, Seconds(2))
/ Create direct kafka stream with brokers and topics
val topicsSet = "hy562-topic".split(",").toSet
val kafkaParams = Map(
  "bootstrap.servers" -> "localhost:9092",
  "key.deserializer" -> classOf[LongDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> "Scala-Kafka-Word-Count",
  "auto.offset.reset" -> "earliest"
)
val messages = KafkaUtils.createDirectStream[String, String](
  ssc,
  LocationStrategies.PreferConsistent,
  ConsumerStrategies.Subscribe[String, String](topicsSet, kafkaParams))
Using Kafka topics with spark (3/3)

Word count example

```scala
// Get the lines, split them into words, count the words and print
val lines = messages.map(_.value)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x ⇒ (x, 1L)).reduceByKey(_ + _)
wordCounts.print()

// Start the computation
ssc.start()
ssc.awaitTermination()
```
References — Spark Streams

- Spark Streaming
- Spark Streaming + Kafka Integration Guide
- Spark Streaming + Kafka Integration Guide
- Processing Data in Apache Kafka with Structured Streaming in Apache Spark 2.2
- Structured Streaming Programming
- https://bitbucket.org/papadako/spark-kafka-wordcount/
- Custom Reliable Receivers