Spark Streaming的流数据处理和分析

本课目标 (Objective)

- [x] 介绍Spark Streaming架构
- [x] 掌握Spark Streaming工作原理
- [x] 掌握基于Spark Streaming的数据源和数据处理和输出（重点）
- [x] 掌握基于Spark Structured Streaming的开发与数据处理（了解）
- [x] 掌握Spark Streaming与Kafka的集成（重点）

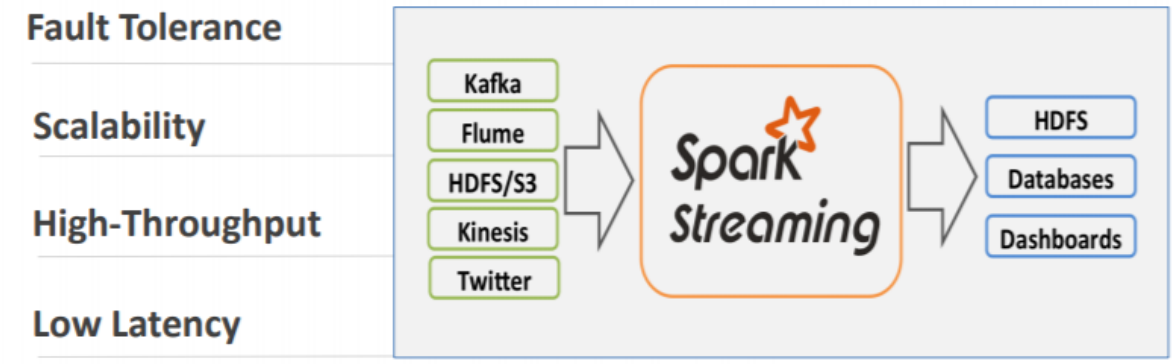
Motivation for Real-Time Data Processing（实时数据处理的动机）

- Data is being Created at Unprecedented Rates（正在以前所未有的速度产生数据）
  - Exponential data growth from mobile, web, social, IoT…（来自移动，网络，社交，物联网的指数数据增长...）
  - Over 1 trillion sensors by 2020（到2020年超过1万亿个传感器）
- How can We Harness the Value of Data in Real-Time?（及时的利用数据的价值）
  - From reactive analysis to direct operational impact（从被动分析到直接的运营影响）
  - Unlocks new competitive advantages（释放新的竞争优势）

Use Cases across Industries（典型的用例）

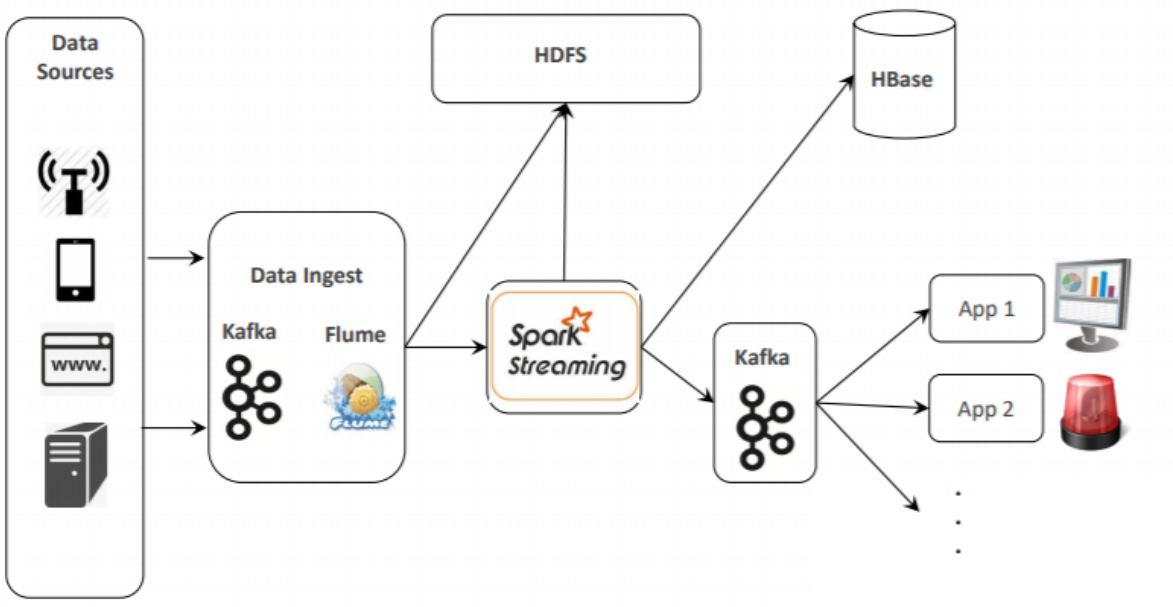
What is Spark Streaming?（什么是Spark Streaming）

- Extension of Apache Spark's Core API, for Stream Processing.（Apache Spark核心API的扩展，用于流处理）
- The Framework Provides（特点）



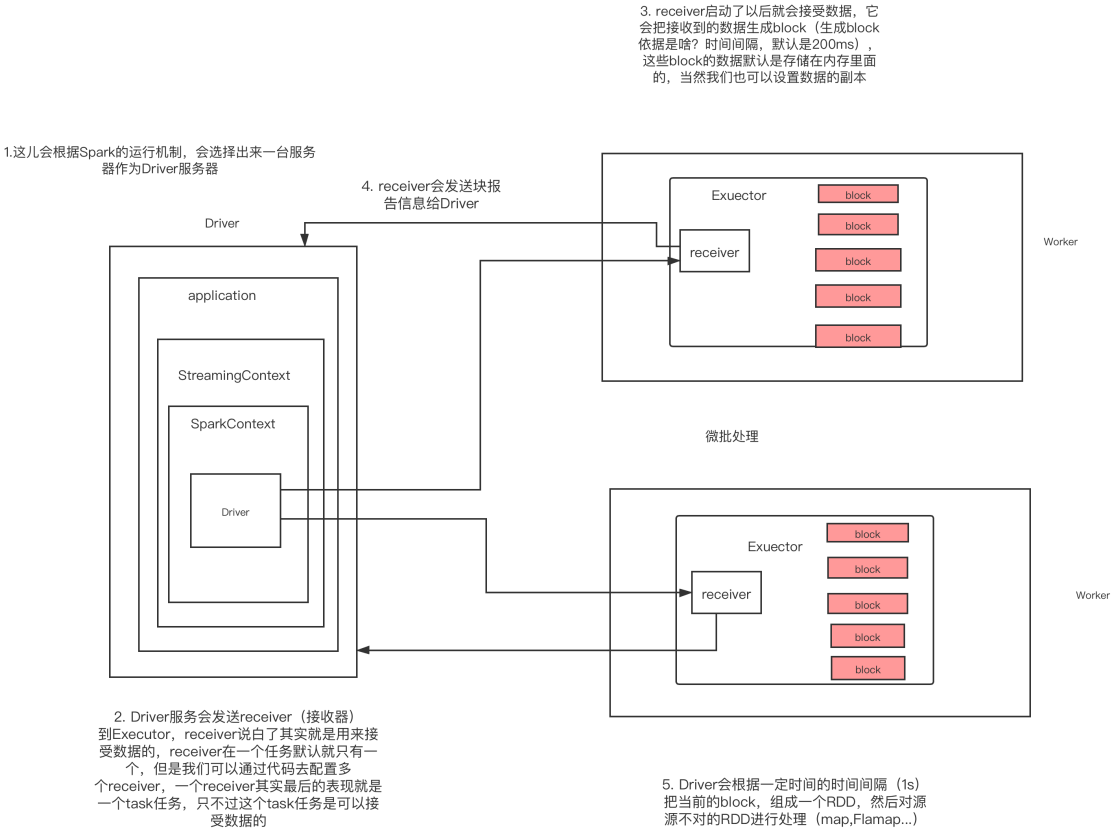
Note：故障容错，可扩展性，高吞吐，低延迟

• Stream Processing Architecture（流处理架构）



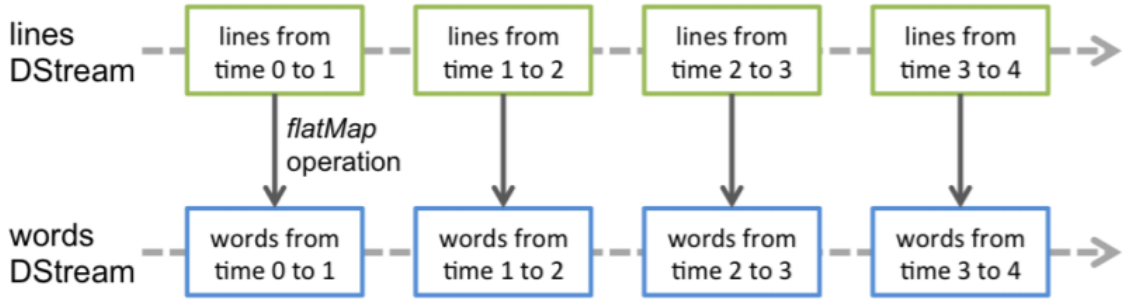
• SparkStreaming任务的运行流程

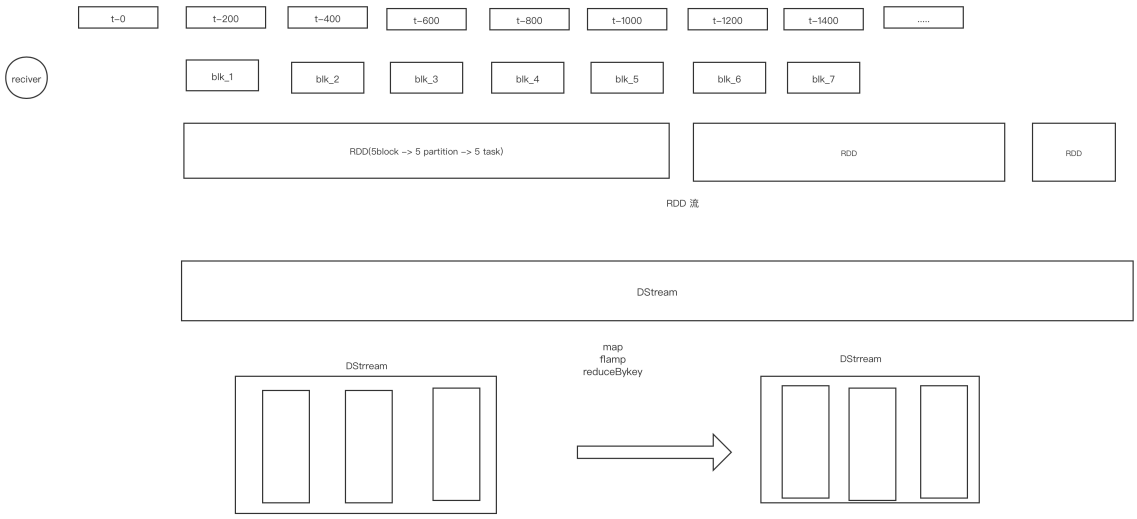
- spark.streaming.blockInterval = 200 （默认200毫秒采集一次数据，形成一个block。
- batch Interval表示一个sparkstreaming的任务，多久运行一次。



• DStream – Discretized Stream（离散流）

- Discretized Stream or DStream is the basic abstraction provided by Spark Streaming.（离散流或DStream是Spark Streaming提供的基本抽象）
  - sparkCore -> rdd
  - SparkSQL -> DataFream
  - SparkStreaming -> DStream
- A DStream is represented by a continuous series of RDDs（DStream由一系列连续的RDD表示）





• Micro-Batch Architecture（微批量架构）

- Incoming data represented as Discretized Streams (DStreams)（表示为离散流（DStream）的传入数据）
- Stream is broken down into micro-batches（流被细分为微批）
  - Latency 1 milli-second since Spark 2.3.1 (before around 100 milli-seconds)（自Spark 2.3.1以来的延迟1毫秒，大约100毫秒之前
- Each micro-batch is an RDD – can share code between batch and streaming（每个微批处理都是一个RDD –可以在批处理和流之间共享代码）

• Streaming Context（SparkStreaming程序入口）

- Streaming Context consumes a stream of data in Spark. （Streaming Context消费Spark中的数据流）

```
val conf = new SparkConf().setMaster("local[2]").setAppName("wordcountstream")
val ssc = new StreamingContext(conf, Seconds(5))
```

• Input DStreams & Receivers（输入DStreams和接收器）

- Input DStreams represent the stream of input data received from streaming sources.（Input DStreams表示从流源接收的输入数据流）
- Every input DStream (except file stream) is associated with a Receiver object which receives the data from a source and stores it in Spark's memory for processing.（每个input DStream（文件流除外）都与接收数据的Receiver对象相关联）
- Multiple Input DStreams could be created under the same StreamingContext（可以在同StreamingContext下创建多个输入DStream）
- Streaming Sources:（流数据源）

- Basic sources Available from Streaming API sc.fileStream, sc.socketStream（基本资源可从Streaming API 中获得）
- Advanced sources Kafka, , Flume, etc（更高级的输入源来自于Kafka，Flume等）

Key Points for InputStream（InputStream的重点）

- When running Spark-Streaming program locally, always use “local[n]” as the master URL, where n > number of receivers;（在本地运行SparkStreaming程序时，请始终使用“ local [n]”作为主URL，其中n>receivers）
- When running on a cluster, the number of cores allocated to the Spark Streaming application must be more than the number of receivers.（在集群上运行时，分配给Spark Streaming应用程序的核心数必须大于接收者数）

Sources of Spark Streaming （数据源）

- Spark StreamingContext has the following built-in Support for creating Streaming Sources:（具有以下内置对创建流源的支持）

```
//Process files in directory (从目录形成数据流，不获取子目录) –
hdfs://namenode:8020/logs/
def textFileStream(directory: String): DStream[String]

//Create an input stream from a TCP source (从网络端口形成数据流)
def socketTextStream(hostname: String, port: Int, storageLevel: StorageLevel
    StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[String]

// Flume Sink for Spark Streaming (从Flume sink)
val ds = FlumeUtils.createPollingStream(streamCtx, [sink hostname], [sink port]);

//Kafka Consumer for Spark Streaming (从kafka Consumer)
val ds = KafkaUtils.createStream(streamCtx, zooKeeper, consumerGrp, topicMap);
```

Demo – Spark Streaming （课堂案例-词频统计）

```
//网络端口开放
nc -lk 9999

//案例1,从socket数据源加载
object WordCountStream {
  def main(args: Array[String]): Unit = {
    //创建程序入口，local[2]表示启动两个线程，如果是1的话，代码在执行但是看不到print出来的数据。
    //至少需要设置2个线程，因为要保证一个线程接受数据，一个处理数据。
    //如果这儿不写，默认的线程的个数，跟你的本地的电脑的cpu core的个数是一样的。
```

```
    //电脑的cpu core的个数，肯定是超过1的.
    val conf = new
SparkConf().setMaster("local[2]").setAppName(this.getClass.getSimpleName)
    val ssc = new StreamingContext(conf,Seconds(2))

    //数据处理
    val dStream: ReceiverInputDStream[String] =
ssc.socketTextStream("192.168.134.130",8888)
    val lines: DStream[String] = dStream.flatMap(_.split(","))
    val wordDStream: DStream[(String,Int)] = lines.map((_,1))
    val result = wordDStream.reduceByKey(_+_)

    //数据保存
    result.print()

    //启动SparkStreaming
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}

//案例2，从HFDS数据源加载
object WordCountForHDFS {
  def main(args: Array[String]): Unit = {
    //设置了日志的级别
    Logger.getLogger("org").setLevel(Level.INFO)
    val conf = new
SparkConf().setMaster("local[2]").setAppName(this.getClass.getSimpleName)
    val ssc = new StreamingContext(conf,Seconds(5))

    /**
      * 如果你的地址是一个高可用的一个地址。
      * core-site.xml
      * hdfs-site.xml
      * 复制到maven项目的 resources目录
      *
      * hdfs://localhost:8020//
      */
    val dataDStream =
ssc.textFileStream("hdfs://192.168.134.130:8020//sparkstreaming//hdfs")

    val reuslt = dataDStream.flatMap(_.split(","))
      .map((_, 1))
      .reduceByKey(_ + _)

    reuslt.print()

    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
```

```
}

//案例3，自定义数据源
class MyCustomReceiver(host: String, port: Int) extends Receiver[String]
(StorageLevel.MEMORY_ONLY) with Logging {

  def onStart() {
    // 启动一个线程，开始接收数据
    new Thread("Socket Receiver") {
      override def run() {
        receive()
      }
    }.start()
  }

  def onStop() {
    // There is nothing much to do as the thread calling receive()
    // is designed to stop by itself isStopped() returns false
  }
  /** Create a socket connection and receive data until receiver is stopped */
  private def receive() {
    var socket: Socket = null
    var userInput: String = null
    try {
      logInfo("Connecting to " + host + ":" + port)
      socket = new Socket(host, port)
      logInfo("Connected to " + host + ":" + port)

      val reader = new BufferedReader(
        new InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8))

      userInput = reader.readLine()

      while(!isStopped && userInput != null) {
        store(userInput)
        userInput = reader.readLine()
      }
      reader.close()
      socket.close()
      logInfo("Stopped receiving")
      restart("Trying to connect again")
    } catch {
      case e: java.net.ConnectException =>
        restart("Error connecting to " + host + ":" + port, e)
      case t: Throwable =>
        restart("Error receiving data", t)
    }
  }
}

//案例3，自定义数据源
object CustomReceiver {
```

```
  def main(args: Array[String]): Unit = {
    //设置了日志的级别
    Logger.getLogger("org").setLevel(Level.ERROR)

    // Create the context with a 1 second batch size
    val sparkConf = new
SparkConf().setAppName("CustomReceiver").setMaster("local[2]")
    val sc = new SparkContext(sparkConf)
    val ssc = new StreamingContext(sc, Seconds(3))


    val lines = ssc.receiverStream(new MyCustomReceiver("192.168.134.130", 9999))


    val words = lines.flatMap(_.split(","))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)


    wordCounts.print()


    ssc.start()
    ssc.awaitTermination()
    ssc.stop(false)
  }
}
```

Transformations on DStreams（Join案例）

- 常规的算子操作
  - map, flatMap
  - filter
  - count, countByValue
  - repartition
  - union, join, cogroup
  - reduce, reduceByKey
- SparkStreaming的API操作
  - transform
  - updateStateByKey

UpdateStateByKey Operation

- The updateStateByKey operation allows to maintain arbitrary state while continuously updating it with new information. （updateStateByKey操作允许维持任意状态，同时用新信息连续更新它）
  - Define the state – the state can be any data type（定义状态-状态可以是任何数据类型）

- Define the state update function（定义状态更新功能）

```
案例4 -> job4
//实现累加词频统计
object UpdateStateByKeyTest {
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)
    //程序入口
    val conf = new
SparkConf().setMaster("local[2]").setAppName(this.getClass.getSimpleName)
    val ssc = new StreamingContext(conf,Seconds(3))

    ssc.checkpoint("E:\\project_workspace\\bw-sparkstreaming\\job4")


    // 数据源
    val dataDStream = ssc.socketTextStream("192.168.134.130",9999)

    //数据处理
    val wDStream = dataDStream.flatMap(_.split(",")) //
hive,hadoop,hadoop,hbase,flink......
    // (hive,1)
    // (hive,1)
    // (hadoop,1)
    val wordDStram = wDStream.map((_,1))
    /**
      *  updateFunc: (Seq[V], Option[S]) => Option[S]
      *  参数二: Seq[V]
      *  hive,1
      *  hive,1
      *  hive,1
      *  hive,1
      *  分组:
      *  {hive,(1,1,1,1)} values=> (1,1,1,1)
      *
      *  参数二: Option[S]
      *  当前的这个key的上一次的状态
      *
      *
      *  返回值: Option[S] 当前key出现的次数。
      *
      *  结论: 将当前的这个key的状态，和上一次出现的状态加起来作为返回值就行了。
      */
    /**
      * hadoop,hadoop,hadoop
      * flink,flink,flink
      * hadoop,hadoop
      * hadoop
      * flink
      *
```

```
      * values => 当前的key对应的value的集合  hadoop,hadoop,hadoop -> (hadoop,
[1,1,1])
      * state => 当前key的上一次的状态 => hadoop,2
      * 返回值：把当前key的当前的状态+上一次的状态进行汇总然后返回。
      */
    val result = wordDStram.updateStateByKey((values:Seq[Int],state:Option[Int])
=> {
      val currentCount = values.sum
      val lastCount = state.getOrElse(0)
      Some(currentCount+lastCount) // (hadoop,[1,1,1]) + hadoop,2 =>  (hadoop,5)
    })

    //输出
    result.print()

    //启动程序
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

- **mapWithState Operation**

```
案例5 -> job5
object MapWithStateAPITest {
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)

    val sparkConf = new
SparkConf().setAppName("NetworkWordCount").setMaster("local[2]")
    val sc = new SparkContext(sparkConf)
    val ssc = new StreamingContext(sc, Seconds(2))

    ssc.checkpoint("E:\\project_workspace\\bw-sparkstreaming\\mapwithstatedir")
    val lines = ssc.socketTextStream("192.168.134.130", 9999)
    val words = lines.flatMap(_.split(","))
    val wordsDStream = words.map(x => (x, 1))

    //定义初始值
    val initialRDD = sc.parallelize(List(("flink", 100L), ("sparkstreaming",
32L)))
    /**
      *
      *
      * // currentBatchTime : 表示当前的Batch的时间
      * // key: 表示需要更新状态的key
      * // value: 表示当前batch的对应的key的对应的值
      * // currentState: 对应key的当前的状态
     */
```

```
    val stateSpec =StateSpec.function((currentBatchTime: Time, key: String,
value: Option[Int], lastState: State[Long]) => {

      val sum = value.getOrElse(0).toLong + lastState.getOption.getOrElse(0L)

      val output = (key, sum)
      //如果你的数据没有超时
      if (!lastState.isTimingOut()) {
        lastState.update(sum)
      }
      //最后一行代码是返回值
      Some(output)
    }).initialState(initialRDD)
      .numPartitions(3).timeout(Seconds(10))
    //timeout: 当一个key超过这个时间没有接收到数据的时候，这个key以及对应的状态会被移除掉

    /**
      * reduceByKey
     *
      * udpateStateByKey
      * mapWithState // bykey -> 顺带就完成了合并的操作
      */
    val result = wordsDStream.mapWithState(stateSpec)

   // result.print()

    result.stateSnapshots().print()

    //启动Streaming处理流
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }

}
```

• Transform Operation（重要）

- The transform operation (along with its variations like transformWith) allows arbitrary RDD-to-RDD functions to be applied on a Dstream.（变换操作（以及它的诸如transformWith之类的变体）允许将任意RDD-to-RDD函数应用于DStream）

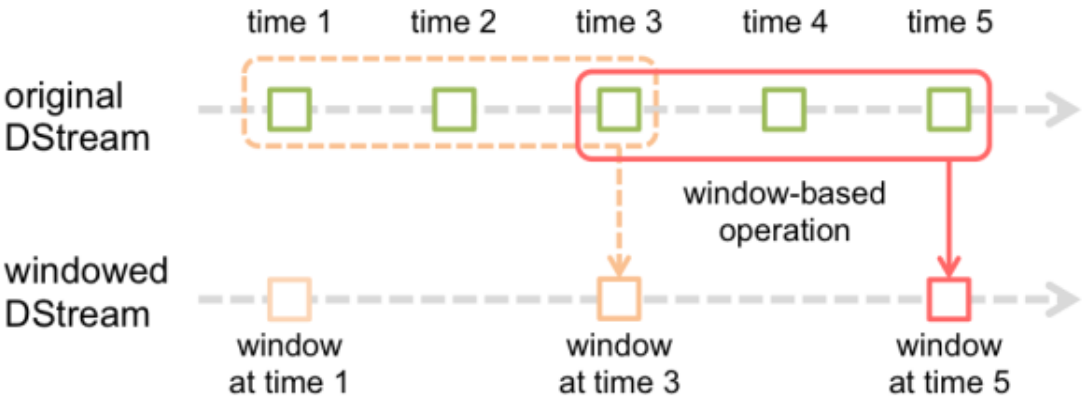
```
案例6 -> job6
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

object BlackWordCount {
  def main(args: Array[String]): Unit = {
```

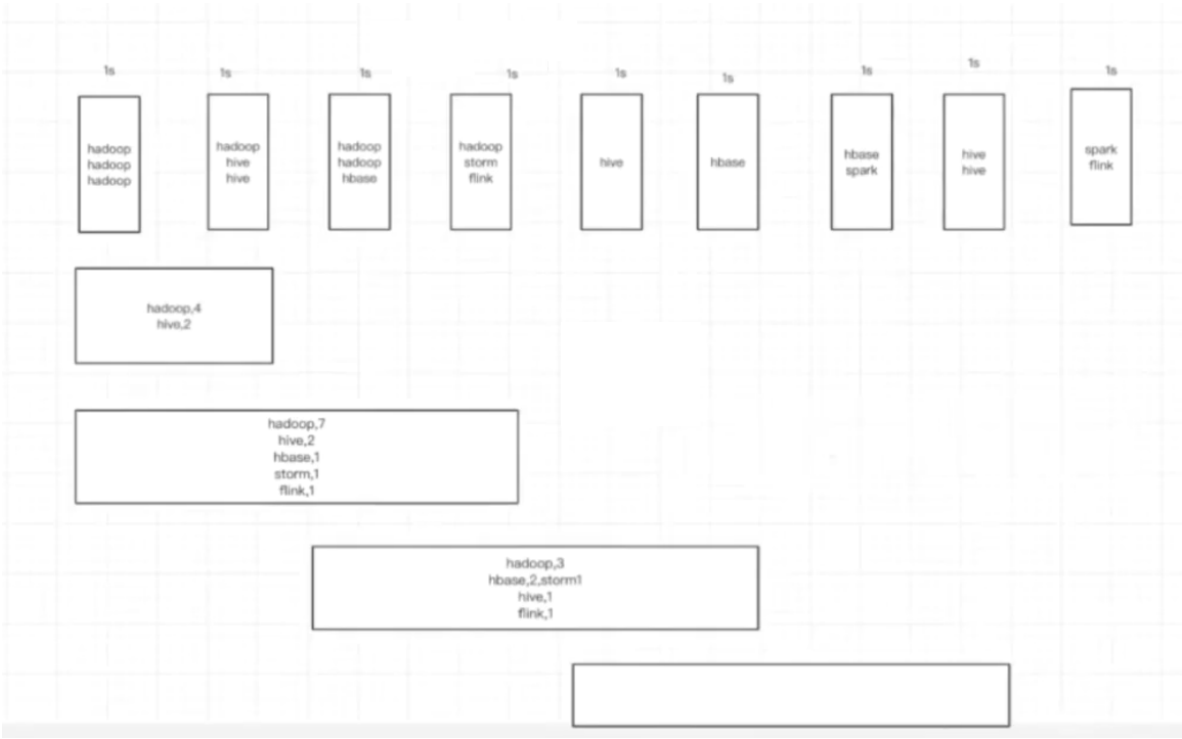
```
    Logger.getLogger("org").setLevel(Level.ERROR)

    val sparkConf = new
SparkConf().setAppName("BlackWordCount").setMaster("local[2]")
    val sc = new SparkContext(sparkConf)
    val ssc = new StreamingContext(sc, Seconds(2))

    val lines = ssc.socketTextStream("localhost", 9998)
    val words = lines.flatMap(_.split(","))
    val wordsDStream = words.map(x => (x, 1))

    /**
     * 首先要获取到黑名单，我们有可能是从Mysql，Redis里面去获取。
     *
     * &
     */
    val filterRDD: RDD[(String, Boolean)] =
ssc.sparkContext.parallelize(List("$","?","!")).map((_,true))
    //广播黑名单数据
    val filterBroadBast = ssc.sparkContext.broadcast(filterRDD.collect())
    //mapRDD
    val filterResultRDD: DStream[(String, Int)] = wordsDStream.transform(rdd => {
      val filterRDD = ssc.sparkContext.parallelize(filterBroadBast.value)
      //RDD join  filterBroadBast  如果join不上的数据是不是就是我们需要的数据?
      /**
       *
       * (String(key), (Int(1), Option[Boolean])) 如果这个option没值
       */
      val result: RDD[(String, (Int, Option[Boolean]))] =
rdd.leftOuterJoin(filterRDD)
      val joinResult = result.filter(tuple => {
        tuple._2._2.isEmpty //过滤出来我们需要的数据
      })
      //在Scala里面最后一行就是方法的返回值
      joinResult.map(tuple => (tuple._1, tuple._2._1)) //hadoop,1
    })

    val result = filterResultRDD.reduceByKey(_+_)
    result.print()
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

• Window Operations（窗口操作）

- Any window operation needs to specify two parameters （任何窗口操作都需要指定两个参数）
    - The window length -（窗口长度）

- The sliding interval - the interval at which the window operation is performed （滑动间隔-执行窗口操作的间隔）
- countByWindow
- reduceByWindow
- reduceByKeyAndWindow
- countByValueAndWindow





• Window Operation Explained（窗口操作）

```
案例7 -> job7
//案例演示
object WindowTest {
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)
    val sparkConf = new
SparkConf().setAppName(this.getClass.getSimpleName).setMaster("local[2]")
    val sc = new SparkContext(sparkConf)
    val ssc = new StreamingContext(sc, Seconds(2))
```

```
    val lines:ReceiverInputDStream[String] =
ssc.socketTextStream("192.168.134.130", 8888)
    val words:DStream[String] = lines.flatMap(_.split(","))
    val wordsDStream = words.map(x => (x, 1))

    /**
      * reduceFunc: (V, V) => V,
      * windowDuration: Duration,
      * slideDuration: Duration 滑动窗口的单位
      * 每隔2秒计算一下，最近4秒的单词出现的次数。
      */
    val result = wordsDStream.reduceByKeyAndWindow((x:Int,y:Int) =>
x+y,Seconds(4),Seconds(2))
    result.print()
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

• Output Operations on DStreams（输出）

Output Operation	Meaning
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. Python API This is called **pprint()** in the Python API.
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]*". Python API This is not available in the Python API.
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]*". Python API This is not available in the Python API.
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.

```
案例8 -> job8
//案例演示 -> WordCountForeachRDD
object WordCountForeachRDD {
  def main(args: Array[String]) {
    //做单词计数
    val sparkConf = new
SparkConf().setAppName(this.getClass.getSimpleName).setMaster("local[2]")
    val sc = new SparkContext(sparkConf)
    val ssc = new StreamingContext(sc, Seconds(5))
    val lines = ssc.socketTextStream("192.168.134.130", 8888)
```

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


    //方案一
    wordCounts.foreachRDD { (rdd, time) =>
      //executed at the driver
      Class.forName("com.mysql.jdbc.Driver")
      val conn = DriverManager.getConnection("jdbc:mysql://hadoop:3306/test",
"root", "root")
      val statement = conn.prepareStatement(s"insert into wordcount(ts, word,
count) values (?, ?, ?)")
      rdd.foreach { record =>
        //executed at the worker(Executor)
        //statement需要通过网络传输到Executor, statement不支持序列化。
        //connection object not serializable
        statement.setLong(1, time.milliseconds)
        statement.setString(2, record._1)
        statement.setInt(3, record._2)
        statement.execute()
      }
      statement.close()
      conn.close()
    }
    //启动Streaming处理流
    ssc.start()
    ssc.stop(false)



    //方案二
    wordCounts.foreachRDD { (rdd, time) =>
      rdd.foreach { record =>
        /**
          * 为每一条数据都创建了一个连接。连接使用完了以后就关闭。频繁的创建和关闭连接。其实对
数据性能影响很大。
          * executor,worker
          */
        Class.forName("com.mysql.jdbc.Driver")
        val conn =
DriverManager.getConnection("jdbc:mysql://hadoop:3306/sparkstreaming", "root",
"root")
        val statement = conn.prepareStatement(s"insert into wordcount(ts, word,
count) values (?, ?, ?)")
        statement.setLong(1, time.milliseconds)
        statement.setString(2, record._1)
        statement.setInt(3, record._2)
        statement.execute()
        statement.close()
        conn.close()
      }
    }
```

```
    //方案三
    wordCounts.foreachRDD { (rdd, time) =>
      rdd.foreachPartition { partitionRecords =>
        /**
          *executor,Worker
          * 为每个partition的数据创建一个连接，比如这个partition里面有1w条数据，
          * 那么这1w条数据，就共用一个连接。这样子的话，连接数就减少了。
          */
        Class.forName("com.mysql.jdbc.Driver")
        val conn = DriverManager.getConnection("jdbc:mysql://hadoop1:3306/test",
"root", "root")
        val statement = conn.prepareStatement(s"insert into wordcount(ts, word,
count) values (?, ?, ?)")
        partitionRecords.foreach { case (word, count) =>
          statement.setLong(1, time.milliseconds)
          statement.setString(2, word)
          statement.setInt(3, count)
          statement.execute()
        }
        statement.close()
        conn.close()
      }
    }

    //方案四
    wordCounts.foreachRDD { (rdd, time) =>
      rdd.foreachPartition { partitionRecords =>
        //使用数据库连接池，提高性能，缺点：只能一条数据的插入，效率太低下。
        val conn = ConnectionPool.getConnection
        val statement = conn.prepareStatement(s"insert into wordcount(ts, word,
count) values (?, ?, ?)")
        partitionRecords.foreach { case (word, count) =>
          statement.setLong(1, time.milliseconds)
          statement.setString(2, word)
          statement.setInt(3, count)
          statement.execute()
        }
        statement.close()
        //返还connection
        ConnectionPool.returnConnection(conn)
      }
    }

    //方案五，使用批处理
    wordCounts.foreachRDD { (rdd, time) =>
      rdd.foreachPartition { partitionRecords =>
        //开启手动管理事务
        val conn = ConnectionPool.getConnection
        conn.setAutoCommit(false)
        val statement = conn.prepareStatement(s"insert into wordcount(ts, word,
count) values (?, ?, ?)")
```

```
        partitionRecords.zipWithIndex.foreach { case ((word, count), index) =>
          statement.setLong(1, time.milliseconds)
          statement.setString(2, word)
          statement.setInt(3, count)
          statement.addBatch()
          //一个批次设置为1000条进行提交。
          if (index != 0 && index % 1000 == 0) {
            statement.executeBatch()
            conn.commit()
          }
        }
        statement.executeBatch()
        statement.close()
        conn.commit()
        conn.setAutoCommit(true)
        ConnectionPool.returnConnection(conn)
      }
    }

    //等待Streaming程序终止
    ssc.awaitTermination()
  }
}
```

• DataFrame & SQL Operations

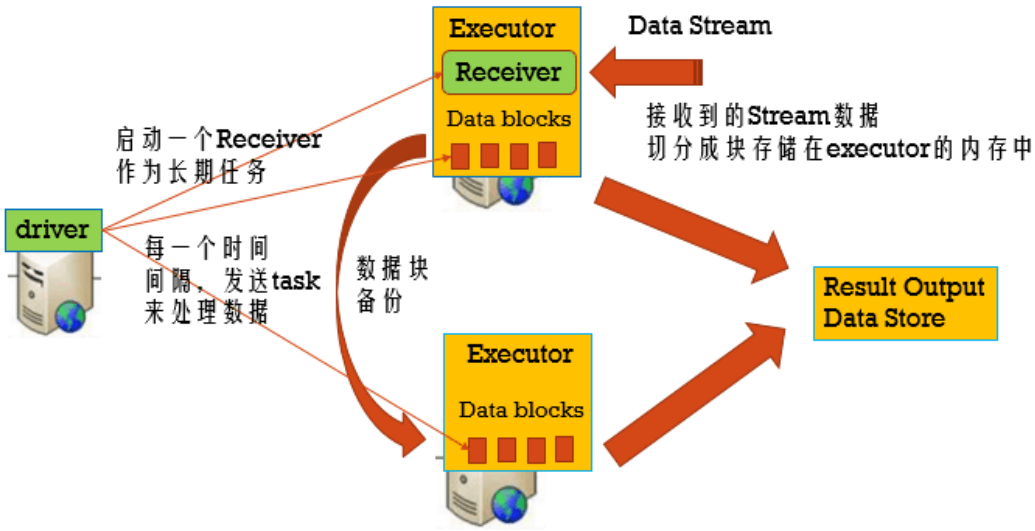
- When using DataFrames and SQL operations on streaming data. a SparkSession needs to be created by using the SparkContext that the StreamingContext is using.（在流数据上使用DataFrames和SQL操作时。 需要使用StreamingContext使用SparkContext创建一个SparkSession）

```
案例9 -> job9
object StreamAndSQLTest {
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)

    val sparkConf = new
SparkConf().setAppName("sparkStreamingSQL").setMaster("local[2]")
    val sc = new SparkContext(sparkConf)
    val ssc = new StreamingContext(sc, Seconds(2))
    val lines = ssc.socketTextStream("192.168.134.130", 8888)
    val words = lines.flatMap(_.split(","))
    //遍历每一个RDD
    words.foreachRDD( rdd =>{
      val spark =
SparkSession.builder().config(rdd.sparkContext.getConf).getOrCreate()
      //隐式转换
      import spark.implicits._
      val wordDataFrame = rdd.toDF("word")
      wordDataFrame.createOrReplaceTempView("words")
```

```
      spark.sql("select word,count(*) as totalCount from words group by word ")
        .show()
    })
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

• SparkStreaming的容错处理

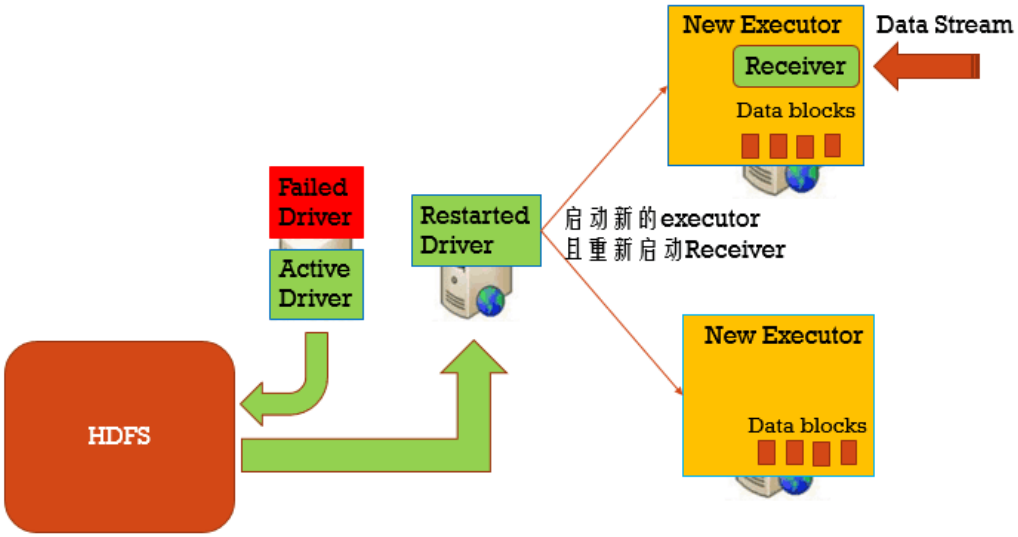
- 1.sparkStreaming运行流程



- 2.Executor失败如何容错

- 如果说receiver所在的Executor节点失败了？会自动选择一台节点重启Receiver任务继续接收数据。
- 如果执行任务的Excutor节点失败了，Driver会重新发送处理数据的task到有备份块所在的executor。
- task和Receiver自动重启，不需要做任何配置。

- 3.Driver失败如何容错



- Driver宕机的话所有的计算和接受到的数据都丢失了
- 利用CheckPoint机制恢复失败的Driver

```
在spark-submit中增加以下两个参数:
Spark的提交模式有两种: client模式, cluster模式。
--deploy-mode cluster
--supervise

Yarn:
在spark-submit中增加以下参数:
--deploy-mode cluster
在yarn配置中设置yarn.resourcemanager.am.max-attemps
<property>
    <name>yarn.resourcemanager.am.max-attempts</name>
    <value>2</value>
</property>
```

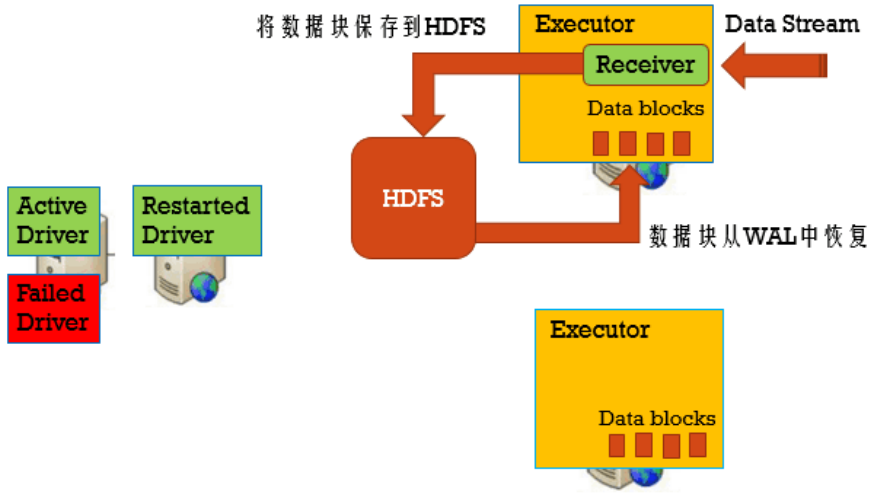
```
案例10 -> job10
object WordCountHA {
  //设置保存点为HDFS的目录
  val checkpointDirectory="E:\\project_workspace\\bw-sparkstreaming\\job10";

  def functionToCreateContext(): StreamingContext = {
    Logger.getLogger("org").setLevel(Level.ERROR)
    val conf = new SparkConf()
    conf.setMaster("local[2]")
    conf.setAppName("DriverHA")
    val ssc = new StreamingContext(conf,Seconds(5))
    ssc.checkpoint(checkpointDirectory)

    val dataStream: DStream[String] =
ssc.socketTextStream("192.168.134.130",8888)
    val lineDSteram: DStream[String] = dataStream.flatMap(_.split(","))
    val wordAndOneDStream: DStream[(String, Int)] = lineDSteram.map((_,1))
```

```
    val result = wordAndOneDStream.updateStateByKey((values: Seq[Int], state:
Option[Int]) => {
      val currentCount = values.sum
      val lastCount = state.getOrElse(0)
      Some(currentCount + lastCount)
    })
    result.print()
    ssc
  }

  /**
    * 根据checkpoint目录里面的元数据信息，生成Driver，如果程序是第一次运行，或者说之前没有
checkpoint目录，那么新创建一个程序入口
    */
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)
    val ssc = StreamingContext.getOrCreate(checkpointDirectory,
functionToCreateContext _)
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

4.如何保证数据不会丢失?

利用WAL（预写日志）把数据写入到HDFS中

WAL使用在文件系统和数据库中用于数据操作的持久性，先把数据写到一个持久化的日志中，然后对数据做操作，如果操作过程中系统挂了，恢复的时候可以重新读取日志文件再次进行操作。对于像kafka和flume这些使用接收器来接收数据的数据源。接收器作为一个长时间的任务运行在executor中，负责从数据源接收数据，如果数据源支持的话，向数据源确认接收到数据，然后把数据存储在executor的内存中，然后Driver在exector上运行任务处理这些数据。如果wal启用了，所有接收到的数据会保存到一个日志文件中去（HDFS), 这样保存接收数据的持久性，此外，如果只有在数据 写入到log中之后接收器才向数据源确认，这样drive重启后那些保存在内存中但是没有写入到log中的数据将会重新发送，这两点保证的数据的无丢失。



步骤一：设置checkpoint目录

//Driver + Exuector数据信息

streamingContext.setCheckpoint(hdfsDirectory)

步骤二：开启WAL日志

sparkConf.set(“spark.streaming.receiver.writeAheadLog.enable”, “true”)

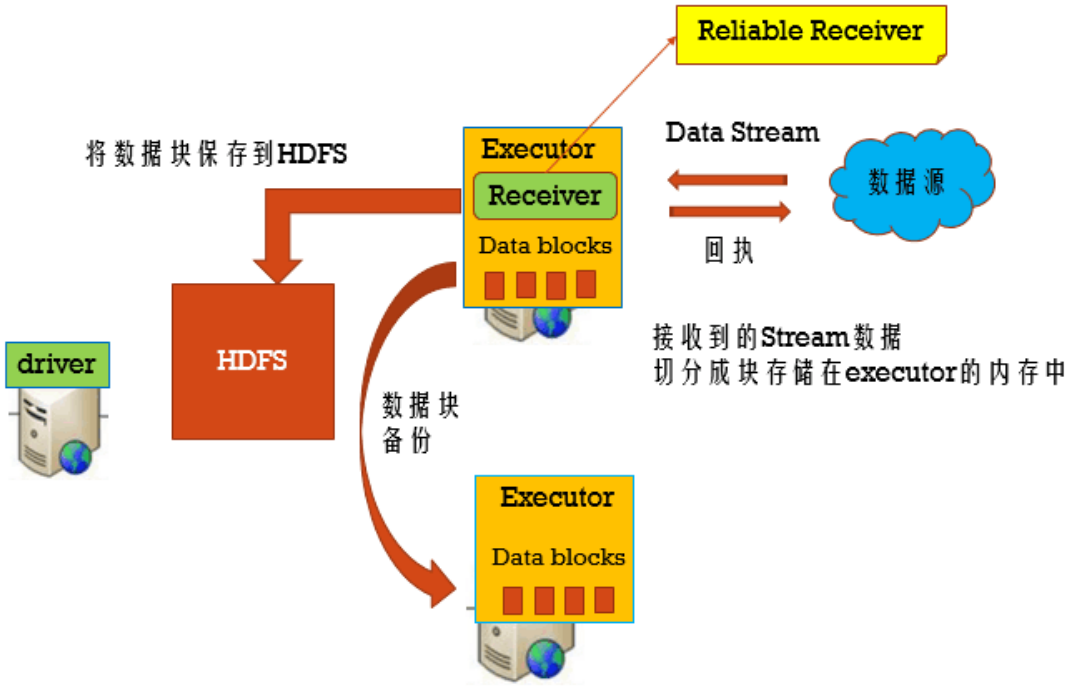
步骤三：需要可靠的数据源

当数据写完了WAL后，才告诉数据源数据已经消费，对于没有告诉数据源的数据，可以从数据源中重新消费数据

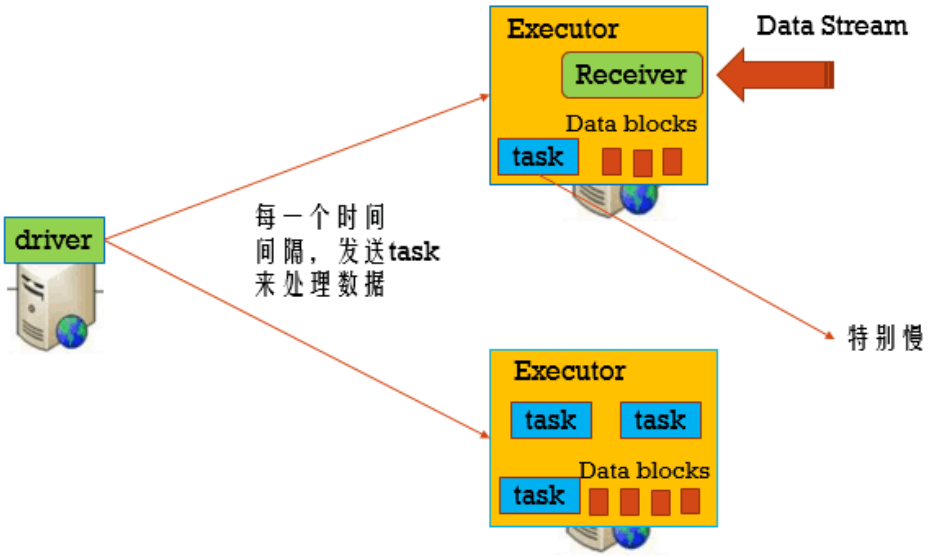
步骤四：取消备份

使用StorageLevel.MEMORY_AND_DISK_SER来存储数据源，不需要后缀为2的策略了，因为HDFS已经是多副本了。

缺点：性能太差，目前产品环境基本上不会使用。

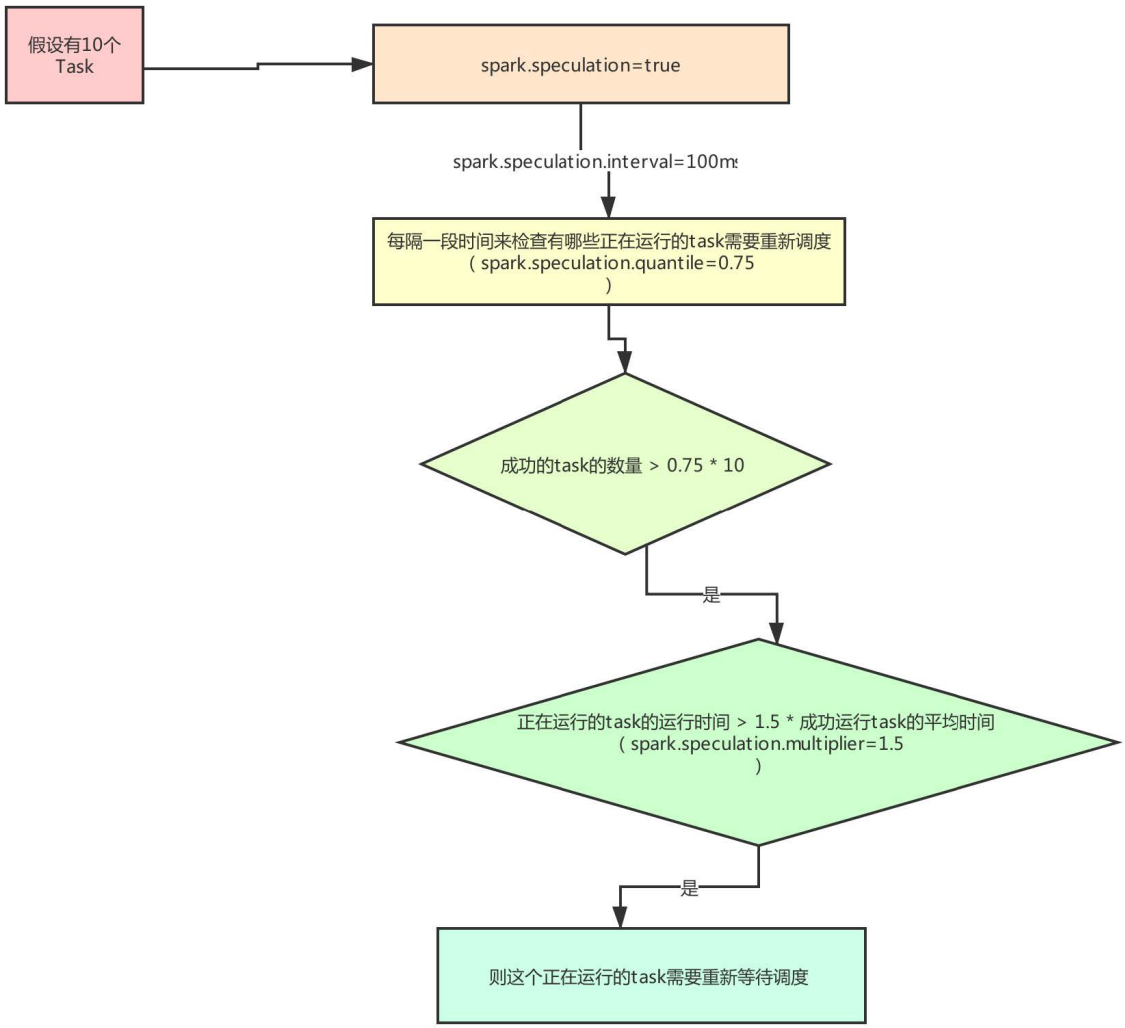


- **5.task运行太慢怎么处理?**



开启推测机制：

spark.speculation=true，每隔一段时间来检查有哪些正在运行的task需要重新调度（spark.speculation.interval=100ms），假设总的task有10个，成功的task的数量 > 0.75 *10（spark.speculation.quantile=0.75），正在运行的task的运行时间 > 1.5 * 成功运行task的平均时间（spark.speculation.multiplier=1.5），则这个正在运行的task需要重新等待调度。



SparkStreaming生产方案

- At most once 一条记录要么被处理一次，要么没有被处理（丢数据）
- At least once 一条记录可能被处理一次或者多次，可能会重复处理（重复消费）
- Exactly once 一条记录只被处理一次（仅一次）

1. SparkStreaming与Kafka整合

方案：http://spark.apache.org/docs/2.3.3/streaming-kafka-integration.html

1.1：Receiver-based Approach（不推荐使用）

此方法使用Receiver接收数据。Receiver是使用Kafka高级消费者API实现的。与所有接收器一样，从Kafka通过Receiver接收的数据存储在Spark执行器中，然后由Spark Streaming启动的作业处理数据。但是，在默认配置下，此方法可能会在失败时丢失数据（请参阅接收器可靠性。为确保零数据丢失，必须在Spark Streaming中另外启用Write Ahead Logs（在Spark 1.2中引入）。这将同步保存所有收到的Kafka将数据写入分布式文件系统（例如HDFS）上的预写日志，以便在发生故障时可以恢复所有数据，但是性能不好。

```
object ReceiverKafkaWordCount {
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)
    //步骤一：初始化程序入口
    val sparkConf = new
SparkConf().setMaster("local[2]").setAppName("ReceiverKafkaWordCount")
    val ssc = new StreamingContext(sparkConf, Seconds(5))

    val kafkaParams =  Map[String, String](
      "zookeeper.connect"->"hadoop1:2181,hadoop2:2181,hadoop3:2181",
      "group.id" -> "sparkStreamingKafka"
    )

    val topics = "streaming1,streaming2,streaming3".split(",").map((_,1)).toMap
    //步骤二：获取数据源
    //默认只会有一个receiver（3 consumer）

    /**
     * String,String  k,v
     * StringDecoder,StringDecoder k,v
     *
     */
//    val lines =
KafkaUtils.createStream[String,String,StringDecoder,StringDecoder](
//     ssc,kafkaParams,topics, StorageLevel.MEMORY_AND_DISK_SER)

    val kafkaStreams = (1 to 20).map(_ => {
      KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](
        ssc, kafkaParams, topics, StorageLevel.MEMORY_AND_DISK_SER)
    })
```

```
    val lines = ssc.union(kafkaStreams)

    //步骤三：业务代码处理
    lines.map(_._2).flatMap(_.split(",")).map((_,1)).reduceByKey(_+_).print()
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

1.2：Direct Approach (No Receivers)

这种新的不基于Receiver的直接方式，是在Spark 1.3中引入的，从而能够确保更加健壮的机制。替代掉使用Receiver来接收数据后，这种方式会周期性地查询Kafka，来获得每个topic+partition的最新的offset，从而定义每个batch的offset的范围。当处理数据的job启动时，就会使用Kafka的简单consumer api来获取Kafka指定offset范围的数据。

这种方式有如下优点：

1、简化并行读取：如果要读取多个partition，不需要创建多个输入DStream然后对它们进行union操作。Spark会创建跟Kafka partition一样多的RDD partition，并且会并行从Kafka中读取数据。所以在Kafka partition和RDD partition之间，有一个一对一的映射关系。

2、高性能：如果要保证零数据丢失，在基于receiver的方式中，需要开启WAL机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到WAL中。而基于direct的方式，不依赖Receiver，不需要开启WAL机制，只要Kafka中作了数据的复制，那么就可以通过Kafka的副本进行恢复。

3、一次且仅一次的事务机制：

基于receiver的方式，是使用Kafka的高阶API来在ZooKeeper中保存消费过的offset的。这是消费Kafka数据的传统方式。这种方式配合着WAL机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为Spark和ZooKeeper之间可能是不同步的。

4、降低资源。

Direct不需要Receivers，其申请的Executors全部参与到计算任务中；而Receiver-based则需要专门的Receivers来读取Kafka数据且不参与计算。因此相同的资源申请，Direct 能够支持更大的业务。

5、降低内存。

Receiver-based的Receiver与其他Exectuor是异步的，并持续不断接收数据，对于小业务量的场景还好，如果遇到大业务量时，需要提高Receiver的内存，但是参与计算的Executor并无需那么多的内存。而Direct 因为没有Receiver，而是在计算时读取数据，然后直接计算，所以对内存的要求很低。实际应用中我们可以把原先的10G降至现在的2-4G左右。

6、鲁棒性更好。

Receiver-based方法需要Receivers来异步持续不断的读取数据，因此遇到网络、存储负载等因素，导致实时任务出现堆积，但Receivers却还在持续读取数据，此种情况很容易导致计算崩溃。Direct 则没有这种顾虑，其Driver在触发batch 计算任务时，才会读取数据并计算。队列出现堆积并不会引起程序的失败。

1.3：整合kafka0.8版本

```
object DirectKafkaWordCount {
```

```
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)
    //步骤一：初始化程序入口
    val sparkConf = new
SparkConf().setMaster("local[2]").setAppName(this.getClass.getSimpleName)
    val ssc = new StreamingContext(sparkConf, Seconds(5))

    val kafkaParams =  Map[String, String](
      "bootstrap.servers"->"kafka1:9091",
      "group.id" -> "directKafkaWordCount"
    )
    val topics = "direct01,direct02".split(",").toSet
    //  ssc: StreamingContext,
    //      kafkaParams: Map[String, String],
    //      topics: Set[String]
    //k(metadata),v(message)

    val lines = KafkaUtils.createDirectStream[String, String, StringDecoder,
StringDecoder](ssc, kafkaParams, topics)
      .map(_._2)
    val result = lines.flatMap(_.split(",")).map((_, 1)).reduceByKey(_ + _)
    result.print()
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

1.4：整合kafka0.10版本

```
object DirectKafka010 {
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)
    //步骤一：获取配置信息
    val conf = new SparkConf().setAppName("DirectKafka010").setMaster("local[5]")
  //  conf.set("spark.streaming.kafka.maxRatePerPartition", "5")
    //conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
    val ssc = new StreamingContext(conf,Seconds(5))

    val brokers = "10.0.0.9:9092,10.0.0.8:9092"
    val topics = "nxtest"
    val groupId = "class3_consumer" //注意，这个也就是我们的消费者的名字

    val topicsSet = topics.split(",").toSet

    val kafkaParams = Map[String, Object](
      "bootstrap.servers" -> brokers,
      "group.id" -> groupId,
      //sparkstremaing消费的kafka的一条消息，最大可以多大
      //默认是1M，比如可以设置为10M，生产里面一般都是设置10M。
```

```
      "fetch.message.max.bytes" -> "209715200",
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer]
    )

    //步骤二：获取数据源
    val stream: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream[String, String](
      ssc,
      LocationStrategies.PreferConsistent,
      ConsumerStrategies.Subscribe[String, String](topicsSet, kafkaParams))
    //key,value(message)
    val result = stream.map(_.value()).flatMap(_.split(","))
      .map((_, 1))
      .reduceByKey(_ + _)

    result.print()


    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

• 生产环境下如何保证数据不丢失（手动管理offset）

0.8版本

0.10版本

• DStream Caching & Persistence（DStream缓存和持久化）

- The data in a DStream can be persisted in memory （DStream中的数据可以保留在内存中）
  - Calling persist() method to persist RDD in DStream （调用persist方法将RDD持久保存在DStream中）
  - DStreams generated by window-based or state-based operations are automatically persisted in memory （通过基于窗口或基于状态的操作生成的DStream会自动保存在内存中）
- Unlike RDDs, the default persistence level of DStreams keeps the data serialized in memory. （与RDD不同，DStream的默认持久性级别将数据序列化在内存中）

• Spark Streaming Checkpointing （保存点）

- Spark Streaming Checkpointing is a fault tolerance mechanism. （检查点是一种容错机制）

- The checkpoint will save DAG and RDDs, when the Spark application is restarted from failure, it will continue to compute. （该检查点将保存DAG和RDD，当Spark应用程序从故障中重新启动时，它将继续进行计算）
- Two types of data that are checkpointed: （有两种类型的数据检查点）
  - Metadata Checkpointing （元数据保存点）
    - Configuration - the configuration used to create the streaming application. （用于创建流应用程序的配置）
    - DStream operations - the set of DStream operations that define the streaming application （定义流应用程序的DStream操作集）
    - Incomplete batches - Batches whose jobs are queued but have not completed yet. （不完整的批次-作业排队但尚未完成的批次）
  - Data Checkpointing （数据保存点）
    - Saving of the generated RDDs to reliable storage. （将生成的RDD保存到可靠的存储中）
    - Intermediate RDDs of stateful transformations are periodically checkpointed to reliable storage (e.g. HDFS) to cut off the dependency chains. （有状态转换的中间RDDs定期被检查点到可靠存储(如HDFS)，以切断依赖链）

When to enable Checkpointing? （何时启用检查点）

- Usage of stateful transformations（有状态转换的使用）
  - If either updateStateByKey or reduceByKeyAndWindow is used, then the checkpoint directory must be provided to allow for periodic RDD checkpointing.（如果使用updateStateByKey reduceByKeyAndWindow，则必须提供checkpoint目录以允许定期的RDD检查点。）
- Recovering from failures of the driver running the application - Metadata checkpoints are used to recover with progress information. （从运行应用程序的驱动程序故障中恢复-元数据检查点用于恢复进度信息）

How to configure Checkpointing? （如何配置检查点）

- streamingContext.checkpoint(checkpointDirectory)
- dstream.checkpoint(checkpointInterval)
  - Typically, a checkpoint interval of 5 - 10 sliding intervals of a DStream is a good setting to try （通常，DStream的5-10个滑动间隔的检查点间隔是一个不错的尝试设置）

```
// Function to create and setup a new StreamingContext (创建和设置新的StreamingContext的功能)
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...) // new context
    val lines = ssc.socketTextStream(...) // create DStreams
    ...
    ssc.checkpoint(checkpointDirectory) // set checkpoint directory
    ssc
}
// Get StreamingContext from checkpoint data or create a new one (从检查点数据获取StreamingContext或创建一个新的)
val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext)
// Start the context
context.start()
context.awaitTermination()
```

• Accumulators, Broadcast Variables, …

- Accumulators, Broadcast variables cannot be recovered from checkpoint in Spark Streaming（累加器，广播变量无法从Spark Streaming中的检查点恢复）

```
object WordBlacklist {
    private var instance: Broadcast[Seq[String]] = null

    def getInstance(sc: SparkContext): Broadcast[Seq[String]] = {
        if (instance == null) {
        synchronized {
            if (instance == null) {
                val wordBlacklist = Seq("a", "b", "c")
                instance = sc.broadcast(wordBlacklist)
            }
        }
    }
    instance
    }
}

wordCounts.foreachRDD { (rdd: RDD[(String, Int)], time: Time) =>
    // Get or register the blacklist Broadcast  (获取或注册blacklist Broadcast)
    val blacklist = WordBlacklist.getInstance(rdd.sparkContext)
    // Use blacklist to drop words and use droppedWordsCounter to count them (使用blacklist删除单词，并使用dropedWordsCounter对其进行计数)
    val counts = rdd.filter { case (word, count) =>
    if (blacklist.value.contains(word)) {
        /* … */
    }
    }.collect().mkString("[", ", ", "]")
    val output = "Counts at time " + time + " " + counts
```

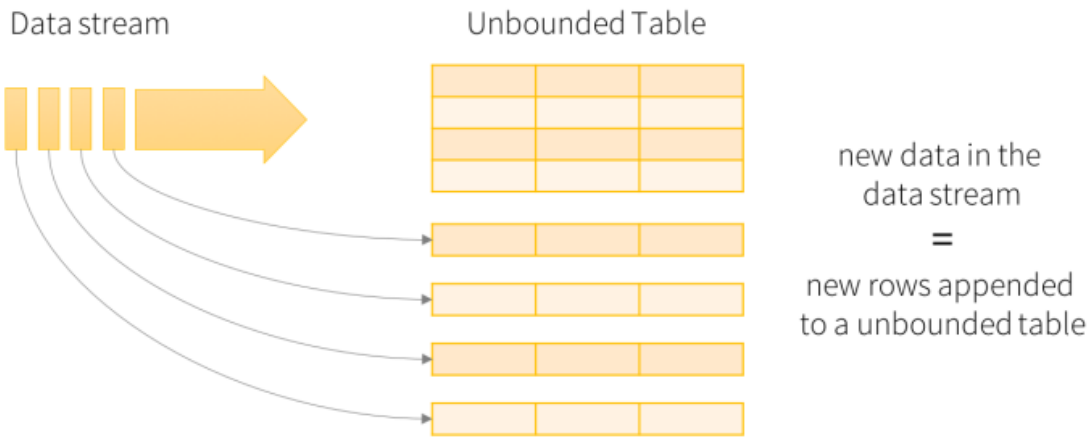
```
})
```

• Spark Structured Streaming（Spark结构化流）

- Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine （基于Spark SQL引擎构建的可扩展且容错的流处理引擎）
- The Spark SQL engine will take care of running the streaming computation incrementally and continuously and updating the final result as streaming data continues to arrive. （当流数据继续到达时，Spark SQL引擎将负责递增地，连续地运行流计算，并更新最终结果）
- The system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write Ahead Logs. （该系统通过检查点和预写日志来确保端到端的一次容错保证）

```
//The lines DataFrame represents an unbounded table containing the streaming text data. ( DataFrame表示一个包含流文本数据的未绑定表)
val lines = spark.readStream.format("socket").option("host", "localhost").option("port", 9999).load()

val words = lines.as[String].flatMap(_.split(" "))

val wordCounts = words.groupBy("value").count()
```

• “Input Table” is Unbounded （“输入表”是无限制的）

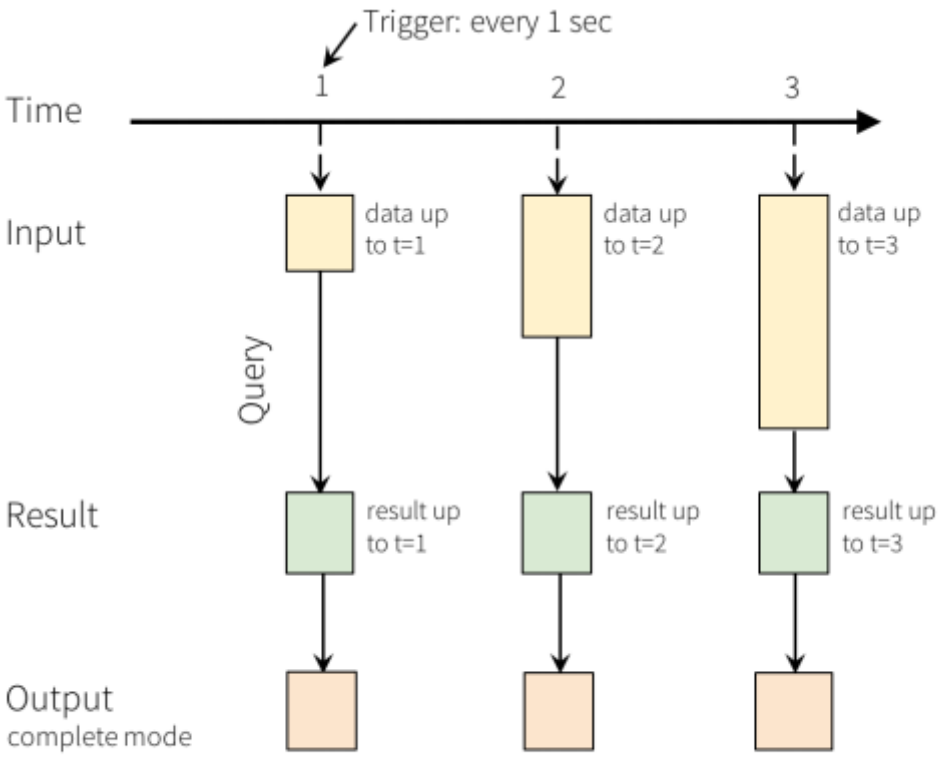


Data stream as an unbounded table

• Programming Model（编程模型）

- With every trigger interval, new rows get appended to the input table. （每次触发，新行就会追加到输入表中）
- Output Mode: （输出方式）

- Complete Mode （完整模式）
  - The entire updated result will be written to the external storage （整个更新结果将被写入外部存储器）
- Append Mode (default) （追加模式）
  - Only new rows appended in the Result Table since the last trigger will be written （只写入结果表中自最后一个触发器以来追加的新行）
- Update Mode （更新模式）
  - Only the rows what were updated in the Result Table since the last trigger will be written （只写入结果表中自上次触发器以来更新的行）



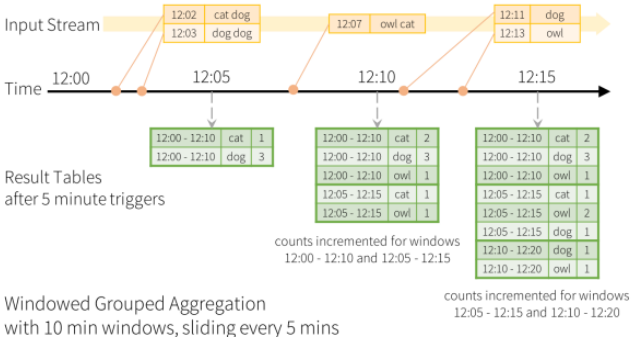
Programming Model for Structured Streaming

In Action

```
val lines = spark.readStream.format("socket").option("host",
"localhost").option("port", 9999).load()

val words = lines.as[String].flatMap(_.split(" "))

val wordCounts = words.groupBy("value").count()

// Start running the query that prints the running counts to the console (开始运行
将正在运行的计数打印到控制台的查询)
val query =
wordCounts.writeStream.outputMode("complete").format("console").start()
query.awaitTermination()
```

• Handling Event-time （处理事件时间）

- Event-time is the time embedded in the data itself, not the time when the data is received. （事件时间是嵌入数据本身的时间，而不是接收数据的时间）
  - The event-time is a column value in the each row （事件时间是每行中的一个列值）
- The window-based aggregations （基于窗口的聚合）



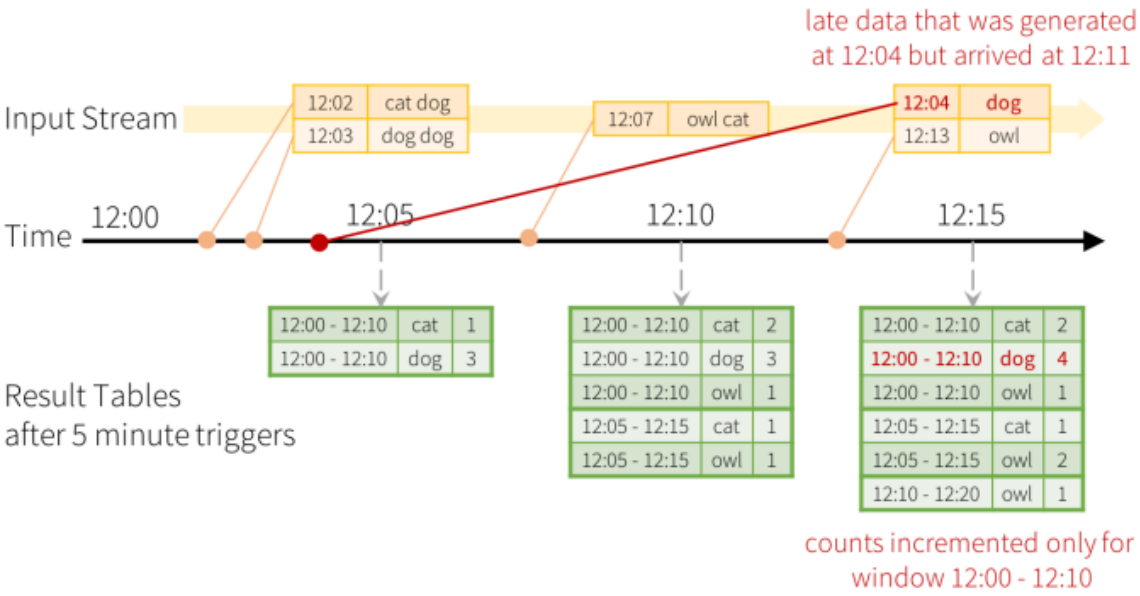
• DEMO Again

```
val lines = spark.readStream.format("socket").option("host",
"localhost").option("port", 9999).load()

// Split the lines into words
val words = lines.as[String].flatMap(_.split("
")).withColumnRenamed("value","word").withColumn("timestamp",
current_timestamp())

val windowedCounts = words.groupBy(window($"timestamp", "10 minutes", "5
minutes"), $"word").count()

// Start running the query that prints the running counts to the console (开始运行
将正在运行的计数打印到控制台的查询)
val query =
windowedCounts.writeStream.outputMode("complete").format("console").start()
query.awaitTermination()
```

Handling Late Data （后期数据处理）

- The event-time model allows to handle data that has arrived later than expected （事件-时间模型允许处理比预期晚到达的数据）
  - Update the old aggregates when there is late data （当有较晚的数据时，更新旧的聚合）
  - Clean up old aggregates （清除旧的aggregates）
- Watermarking
  - A time threshold for how late data can still be handled （一个时间阈值，指示如何仍可处理最新数据）

```
val windowedCounts = words.withWatermark("timestamp", "10 minutes")
.groupBy(window($"timestamp", "10 minutes", "5 minutes"), $"word")
.count()
```



Using DataFrame & Dataset

```
case class WordCount(word: String, count: Int)

val lines = spark.readStream.format("socket").option("host",
"localhost").option("port", 9999).load()

// Split the lines into words
val df = lines.as[String].flatMap(_.split(" ").map(w => (w,
1))).withColumnRenamed("_1", "word").withColumnRenamed("_2", "count")

df.createOrReplaceTempView("updates")

val df1 = spark.sql("select count(*) from updates")

val ds = df.as[WordCount]
```

```
import org.apache.spark.sql.expressions.scalalang.typed

val result = ds.groupByKey(_.word.substring(0, 1)).agg(typed.sum(_.count))

// Start running the query that prints the running counts to the console (开始运行将运行计数打印到控制台的查询)
val query = result.writeStream.outputMode("complete").format("console").start()
query.awaitTermination()
```

- **Join Operations**

```
import org.apache.spark.sql.functions.expr

val impressions = spark.readStream. ...
val clicks = spark.readStream. ...

// Apply watermarks on event-time columns (在事件时间列上应用水印)
val impressionsWithWatermark = impressions.withWatermark("impressionTime", "2 hours")
val clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")

// Join with event-time constraints (加入事件时间限制)
impressionsWithWatermark.join(clicksWithWatermark,
expr("clickAdId = impressionAdId AND clickTime >= impressionTime AND clickTime <= impressionTime + interval 1 hour"), "inner")
……
//Can be “inner”, “leftOuter”, “rightOuter” (可以是“内部”, “左外部”, “右外部”)
```

- **“Legal” Join Types**

Left Input	Right Iput	Join Type	
static	static	all types	Supported
stream	static	left outer	Supported
		right outer	Not supported
		full outer	Not supported
static	stream	left outer	Not supported
		right outer	Supported
		full outer	Not supported
stream	stream	inner	Supported
		left outer	Conditionally supported, must specify watermark on right + time constraints for correct results（有条件支持，必须指定正确的watermark+时间限制的正确结果）
		right outer	Conditionally supported, must specify watermark on left + time constraints for correct results（有条件支持，必须指定watermark在左边+时间限制的正确结果）
		full outer	Not supported

Import Notes:

- Joins can be cascaded - df1.join(df2, ...).join(df3, ...) （join可以级联）
- As of Spark 2.3, joins can only be applied when the query is in Append output mode （从Spark 2.3开始，仅当查询处于Append输出模式时才能应用联接）
- Cannot use streaming aggregations before joins Cannot use mapGroupsWithState and flatMapGroupsWithState in Update mode before joins（联接前不能使用流式聚合联接前不能在更新模式下使用mapGroupsWithState和flatMapGroupsWithState）

Streaming Deduplication

- Duplicate records can be dropped in data streams using a unique identifier in the events. （可以使用事件中的唯一标识符在数据流中删除重复记录）

```
val streamingDf = spark.readStream...
// Without watermark using guid column
streamingDf.dropDuplicates("guid")
// With watermark using guid and eventTime columns
streamingDf .withWatermark("eventTime", "10 seconds").dropDuplicates("guid",
"eventTime")
```

• Input Sources for Structured Streaming（结构化流的输入源）

```
//File source
val eventSchema = new StructType().add("event_id", "string").add("user_id",
"string") .add("start_time", "string").add("address", "string").add("city",
"string").add("state", "string").add("country", "string").add("latitude",
"float").add("longitude", "float")

val dfEvents =
spark.readStream.option("sep",",").schema(eventSchema).csv("hdfs:///user/events/e
vents_*.csv")

val result = dfEvents.groupBy("user_id").count()
// Start running the query that prints the running counts to the console
val query = result.writeStream.outputMode("complete").format("console").start()
query.awaitTermination()
```

• Spark Streaming with Kafka

- Since Kafka 0.10 or higher, only Direct DStream is supported （Kafka 0.10或更高，只支持直接DStream）
    - 1:1 correspondence between Kafka partitions and Spark partitions （Kafka分区和Spark分区之间的1:1通信）

```
val kafkaParams = Map[String, Object](
    "bootstrap.servers" -> "localhost:9092,anotherhost:9092",
    "key.deserializer" -> classOf[StringDeserializer],
    "value.deserializer" -> classOf[StringDeserializer],
    "group.id" -> "use_a_separate_group_id_for_each_stream",
    "auto.offset.reset" -> "latest",
    "enable.auto.commit" -> (false: java.lang.Boolean)
)

val topics = Array("topicA", "topicB")
```

```
val stream = KafkaUtils.createDirectStream[String, String](streamingContext,
PreferConsistent,Subscribe[String, String](topics, kafkaParams))
stream.map(record => (record.key, record.value))

//Each item in the stream is a ConsumerRecord (流中的每个项都是一个消费记录)
```

- **Query Kafka with Structured Streaming**

```
// Subscribe to 1 topic (Streaming Queries)
val df = spark.readStream
.format("kafka")
.option("kafka.bootstrap.servers", "host1:port1,host2:port2")
.option("subscribe", "topic1")
.load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)").as[(String,
String)]

// Subscribe to 1 topic  (Batch Queries)
val df = spark.read
.format("kafka")
.option("kafka.bootstrap.servers", "host1:port1,host2:port2")
.option("subscribe", "topic1")
.option("startingOffsets", "earliest")
.option("endingOffsets", "latest")
.load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)").as[(String,
String)]
```

- **Write Kafka with Structured Streaming**

```
//Streaming Writes
val ds = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
.writeStream
.format("kafka")
.option("kafka.bootstrap.servers", "host1:port1,host2:port2")
.option("topic", "topic1")
.start()

//Batch Writes
df.selectExpr("topic", "CAST(key AS STRING)", "CAST(value AS STRING)")
.write
.format("kafka")
.option("kafka.bootstrap.servers", "host1:port1,host2:port2")
.save()
```

- **本课总结 (Summary)**

- 介绍了Spark Streaming架构
- 理解了Spark Streaming工作原理
- 学习了基于Spark Streaming的开发与数据处理
- 学习了基于Spark Structured Streaming的开发与数据处理
- 学习了Spark Streaming与Kafka的集成
- 完成了基于Spark Streaming的项目实现