第五课： Spark Streaming 原理和实践

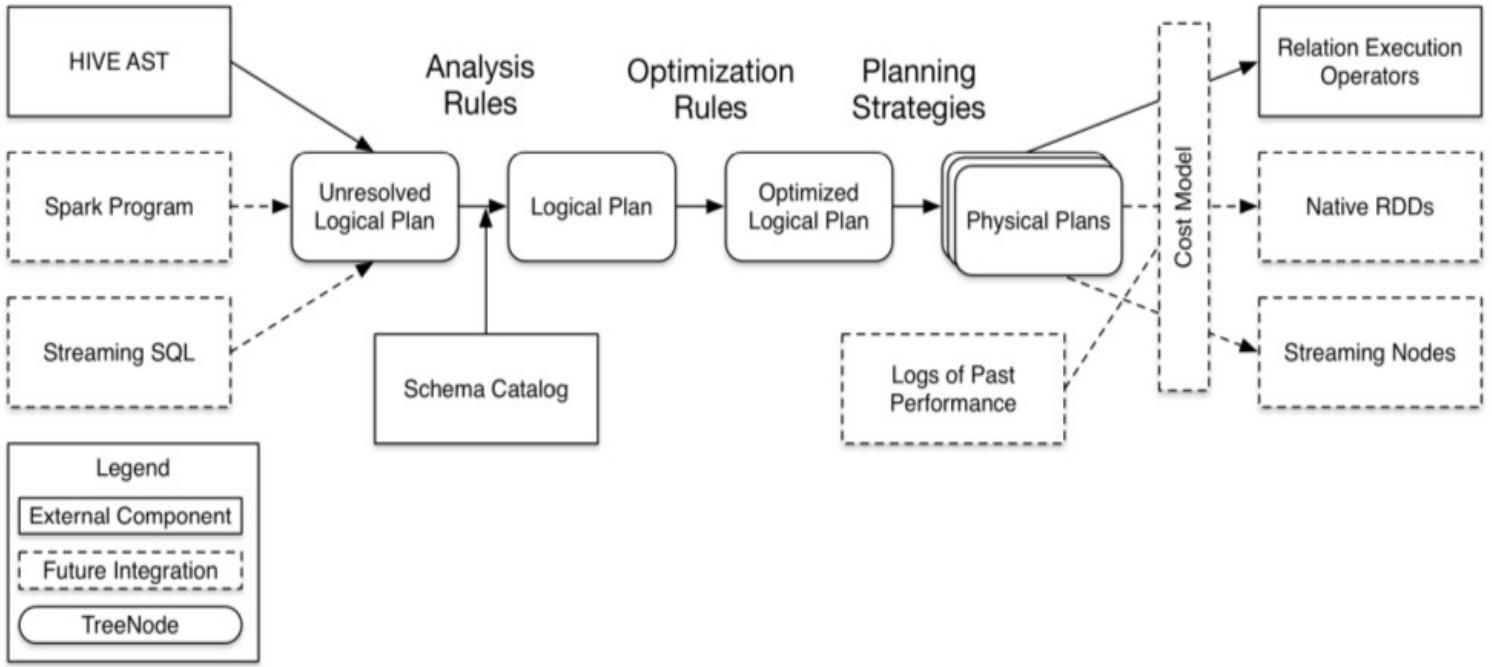
法律声明

【声明】本视频和幻灯片为炼数成金网络课程的教学资料，所有资料只能在课程内使用，不得在课程以外范围散播，违者将可能被追究法律和经济责任。

课程详情访问炼数成金培训网站

http://edu.dataguru.cn

上周回顾



- Hive
- Shark
- SparkSQL

本周内容

- Storm 简介
- Spark Streaming 原理
  - 运行原理
  - 三种运用场景
  - 编程模型 DStream
  - 持久化和容错
  - 优化
- 实例演示
  - 网络数据处理
  - 文本数据处理
  - Stateful 操作
  - window 操作

为什么需要流处理？

Many big-data applications need to process large data streams in realtime

Website monitoring

Fraud detection

Ad monetization

为什么需要流处理？

- MapReduce 不满足时效性要求的原因
  - 一批数据启动一次，处理完进程停止
    - 启动本身是需要时间的：输入切分、调度、启动进程
    - 共享集群 Job 比较复杂，可能需要等待资源
  - 所有数据都需要读写磁盘

- 解决方案
  - 进程常驻运行
  - 数据在内存中

- 常用流处理系统
  - Storm
  - Trient
  - S4
  - Spark Streaming

本周内容

- Storm 简介
- Spark Streaming 原理
  - 运行原理
  - 三种应用场景
  - 编程模型 DStream
  - 持久化和容错
  - 优化
- 实例演示
  - 网络数据处理
  - 文本数据处理
  - Stateful 操作
  - window 操作

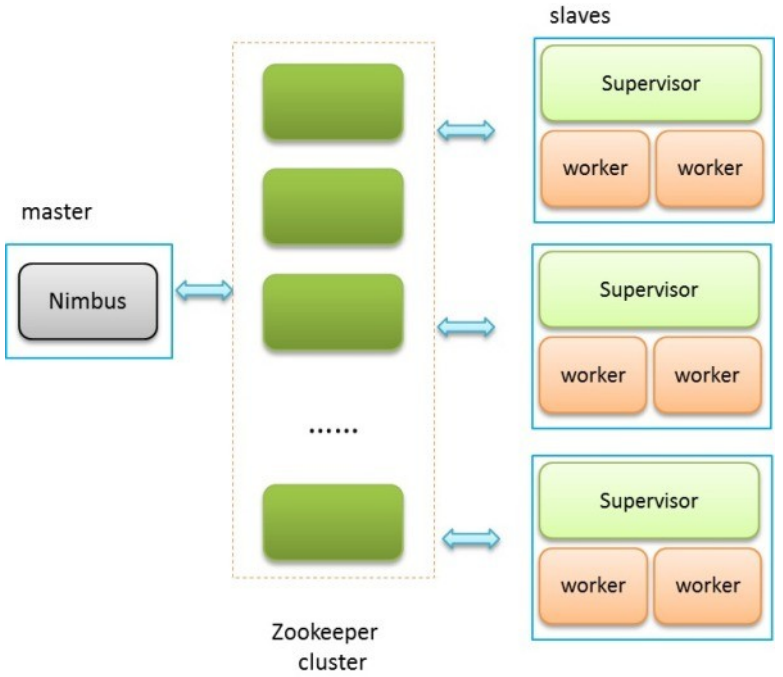
Storm 简介

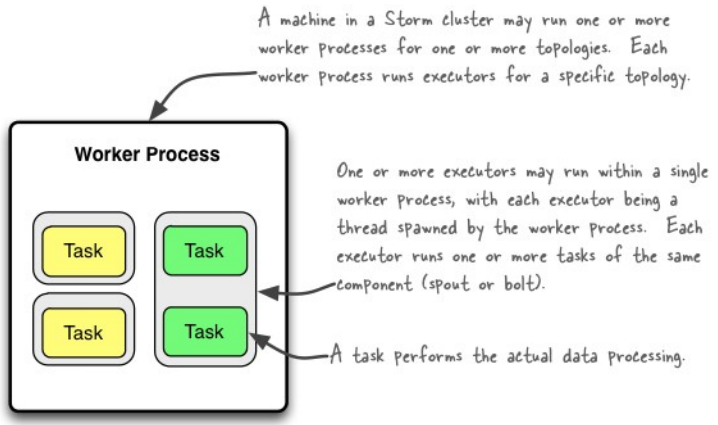
- Nimbus ：负责资源分配和任务调度，把任务相关的元信息写入 Zookeeper 相应目录。
- Supervisor ：负责接受 nimbus 分配的任务，启动和停止属于自己管理的 worker 进程。
- Worker ：运行具体处理组件逻辑的进程。
- Executor ：运行 spout/bolt 的线程
- Task ： worker 中每一个 spout/bolt 的线程称为一个 task.

- Topology ： storm 中运行的实时应用程序，消息在各个组件间流动形成逻辑上的拓扑结构。
- Spout ：在一个 topology 中产生源数据流的组件， Spout 是一个主动的角色。
- Bolt ：在一个 topology 中接受数据然后执行处理的组件。 Bolt 可以执行过滤、函数操作、合并、写数据库等任何操作。 Bolt 是一个被动的角色。

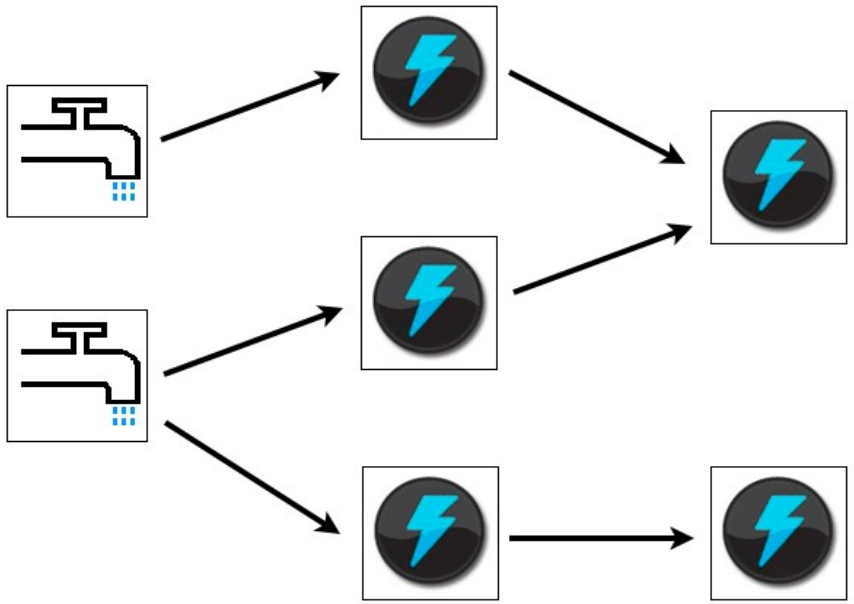
- Tuple ：消息传递的基本单元。
- Stream ：源源不断传递的 tuple 就组成了 stream 。
- stream grouping ：即消息的 partition 方法。 Storm 中提供若干种实用的 grouping 方式，包括 shuffle, fields hash, all, global, none, direct 和 localOrShuffle 等

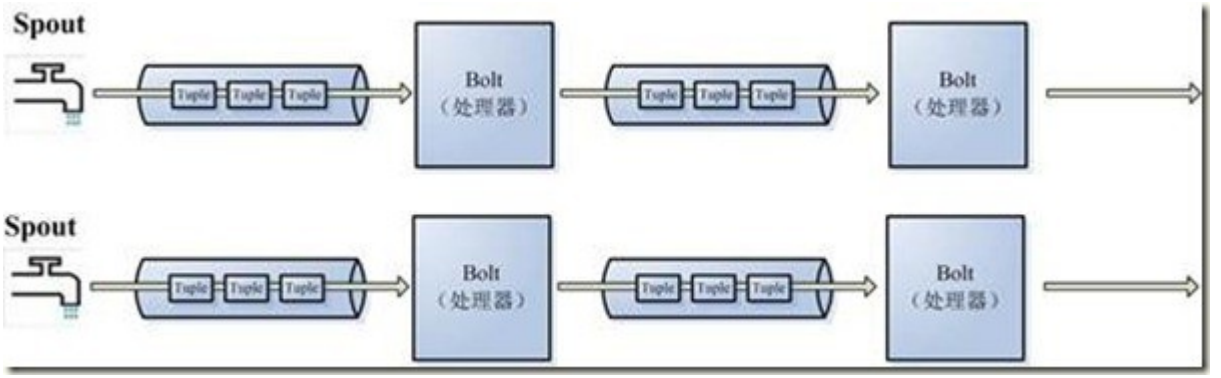
Storm 简介

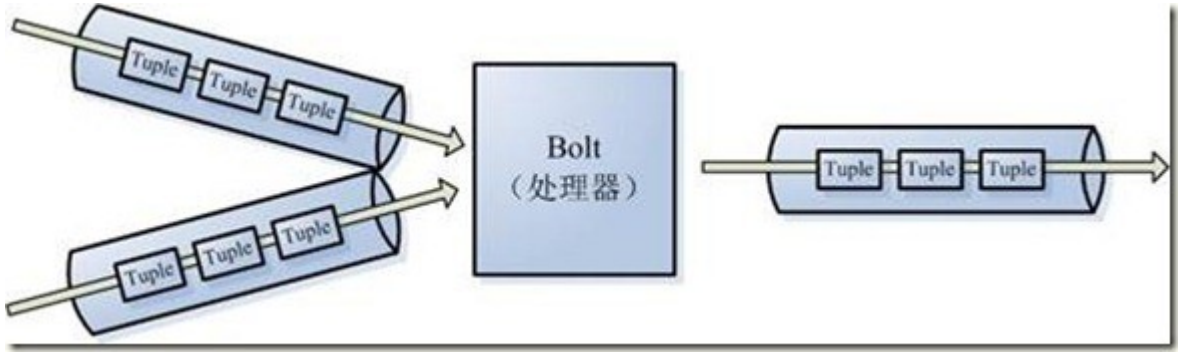

slaves
Supervisor
worker
worker
master
Nimbus
Supervisor
worker
worker
.....
Supervisor
worker
worker
Zookeeper
cluster


A machine in a Storm cluster may run one or more worker processes for one or more topologies. Each worker process runs executors for a specific topology.
Worker Process
Task
Task
Task
Task
One or more executors may run within a single worker process, with each executor being a thread spawned by the worker process. Each executor runs one or more tasks of the same component (spout or bolt).
A task performs the actual data processing.

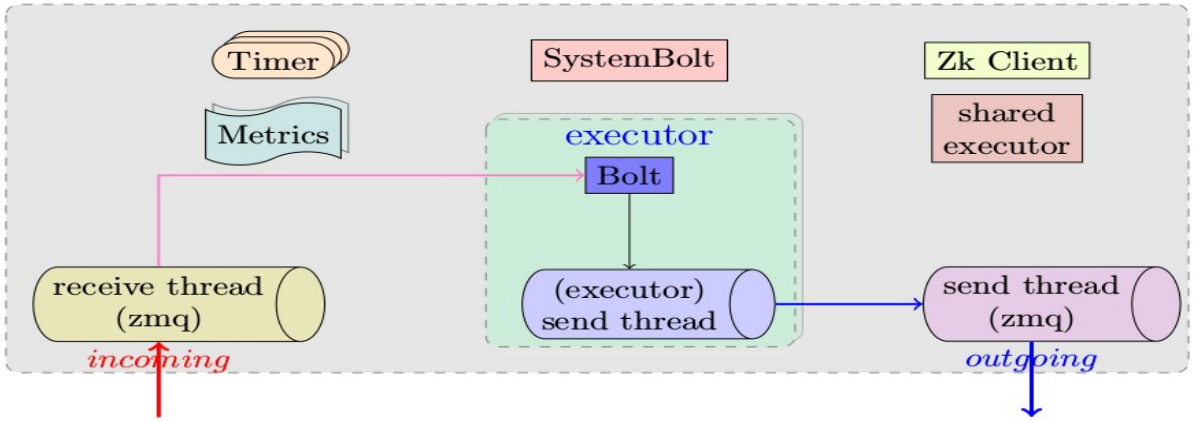
Storm 简介

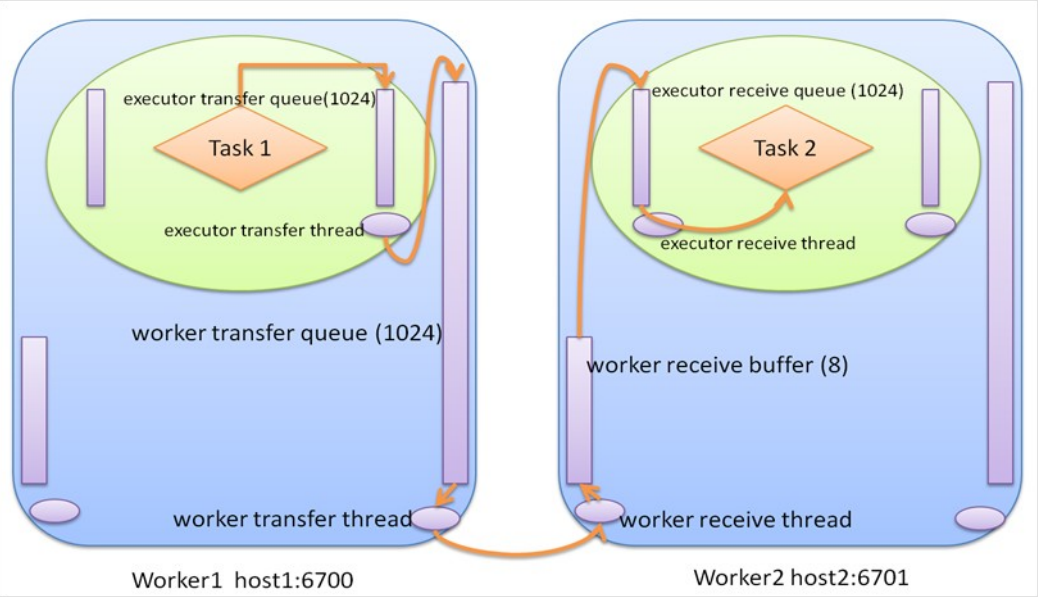



Spout
Tuple
Tuple
Tuple
Bolt
（处理器）
Tuple
Tuple
Tuple
Bolt
（处理器）
Spout
Tuple
Tuple
Tuple
Bolt
（处理器）
Tuple
Tuple
Tuple
Bolt
（处理器）


Tuple
Tuple
Tuple
Tuple
Tuple
Tuple
Bolt
（处理器）
Tuple
Tuple
Tuple

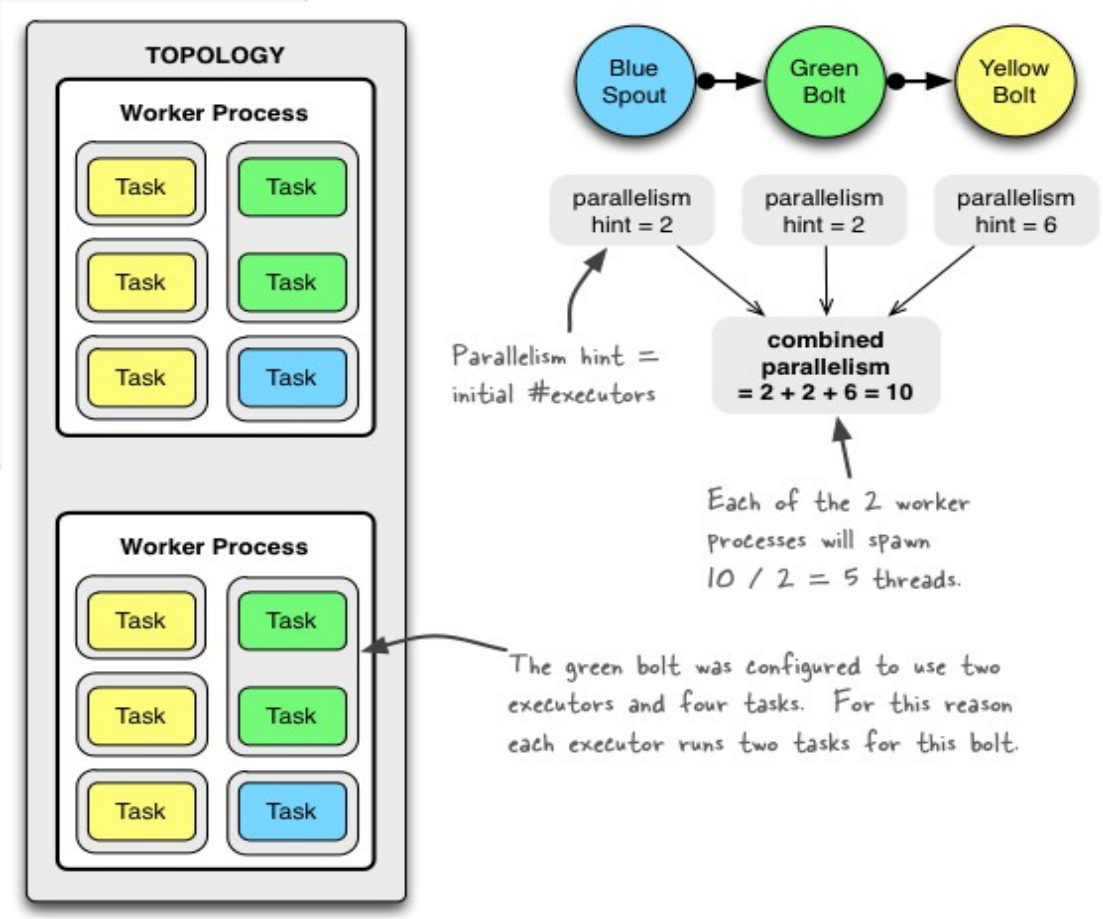
Storm 简介


Timer
Metrics
SystemBolt
executor
Bolt
Zk Client
shared executor
receive thread (zmq)
incoming
(executor) send thread
send thread (zmq)
outgoing


executor transfer queue(1024)
Task 1
executor transfer thread
worker transfer queue (1024)
worker transfer thread
Worker1 host1:6700
executor receive queue (1024)
Task 2
executor receive thread
worker receive buffer (8)
worker receive thread
Worker2 host2:6701

Storm 简介

```
Config conf = new Config();
conf.setNumWorkers(2); // use two worker processes

topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2); // set parallelism hint to 2

topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
               .setNumTasks(4)
               .shuffleGrouping("blue-spout");

topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
               .shuffleGrouping("green-bolt");

StormSubmitter.submitTopology(
        "mytopology",
        conf,
        topologyBuilder.createTopology()
    );
```



Storm 简介

- 多语言编程。可以在 Storm 之上使用各种编程语言。默认支持 Clojure 、 Java 、 Ruby 和 Python 。要增加对其他语言的支持，只需实现一个简单的 Storm 通信协议即可。

- 容错性。 Storm 会管理工作进程和节点的故障。如果您执行的计算过程中有错误， Storm 将重新分配任务；此外，通过 Transactional Topology ， Storm 可以保证每个 tuple“ 被且仅被处理一次”。 Storm 确保一个计算可以一直运行下去（或直到你杀死计算）。

- 水平扩展。计算是在多个线程、进程和服务器之间并行进行的。

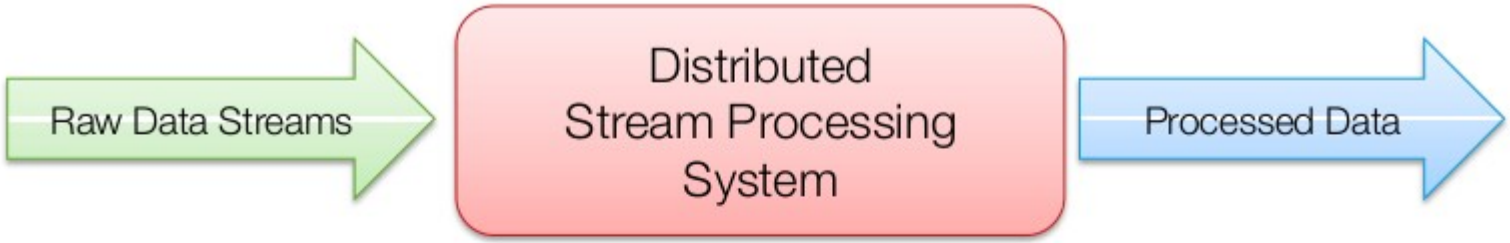
- 快速。系统的设计保证了消息能得到快速的处理，使用 ?MQ 作为其底层消息队列。

- 系统可靠性。 Storm 这个分布式流计算框架是建立在 Zookeeper 的基础上的，大量系统运行状态的元信息都序列化在 Zookeeper 中。这样，当某一个节点出错时，对应的关键状态信息并不会丢失，换言之 Zookeeper 的高可用保证了 Storm 的高可用。

本周内容

- Storm 简介
- Spark Streaming 原理
  - 运行原理
  - 三种运用场景
  - 编程模型 DStream
  - 持久化和容错
  - 优化
- 实例演示
  - 网络数据处理
  - 文本数据处理
  - Stateful 操作
  - window 操作

什么是 Spark Streaming



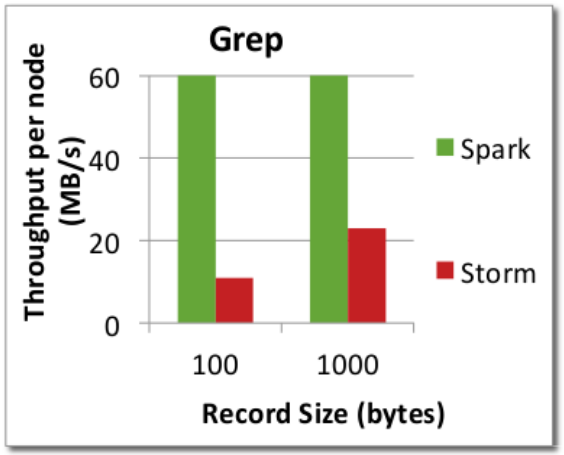
Scales to hundreds of nodes

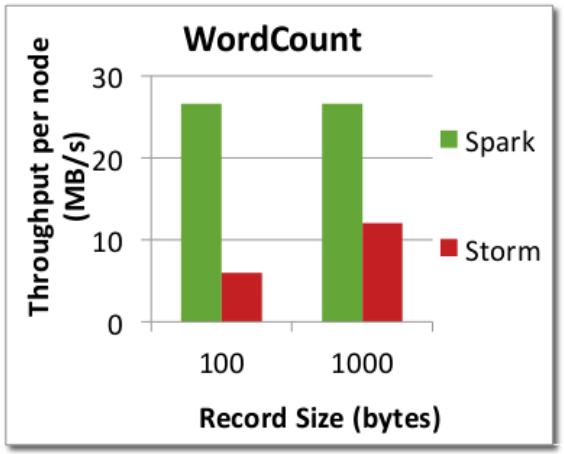
Achieves low latency

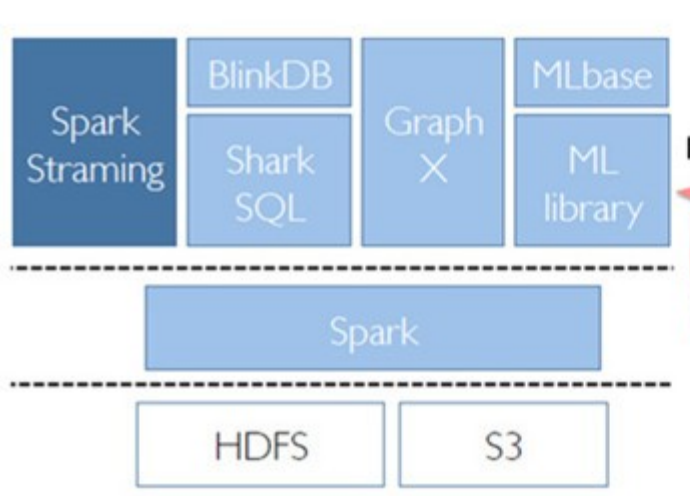
Efficiently recover from failures

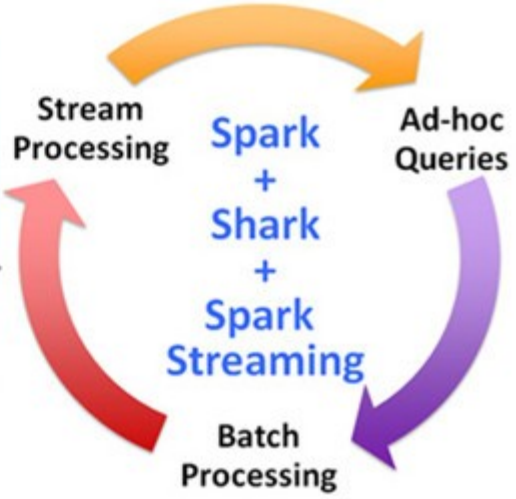
Integrates with batch and interactive processing

什么是 Spark Streaming

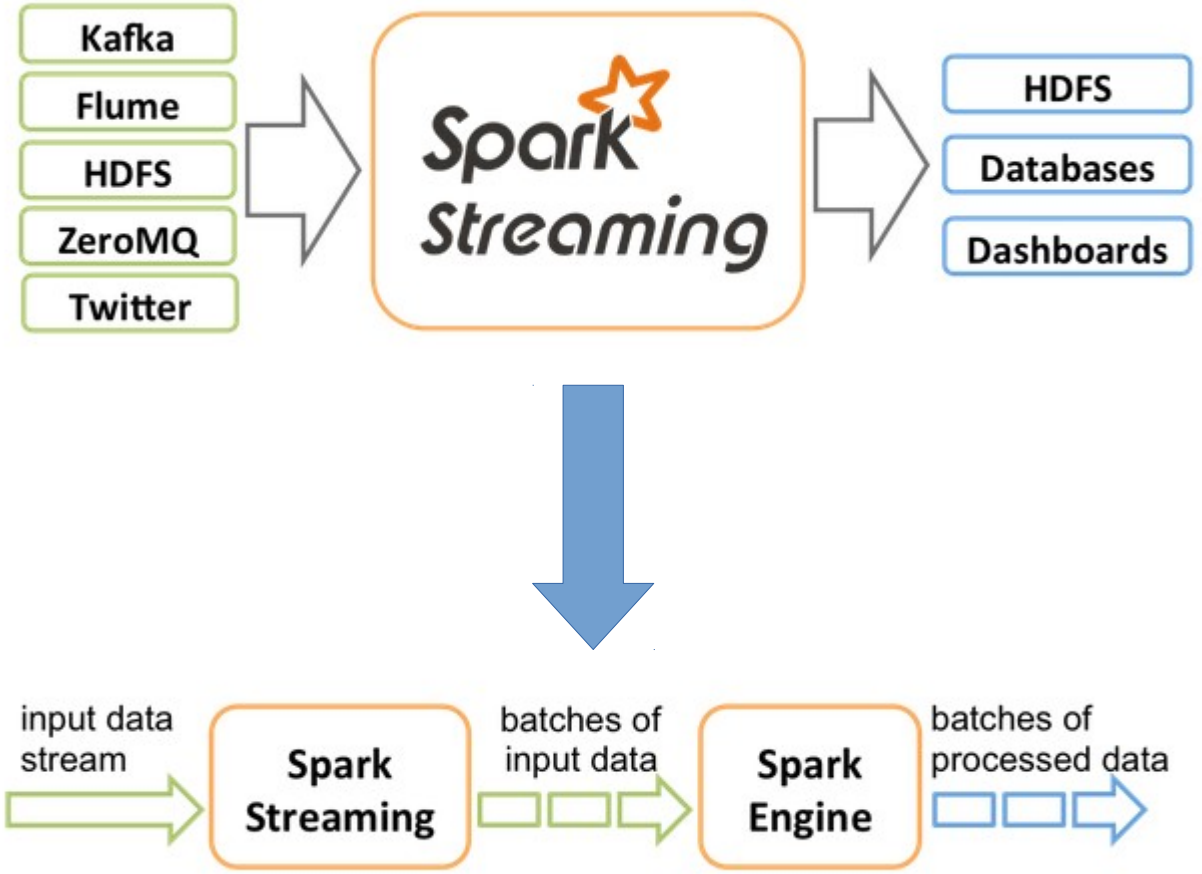






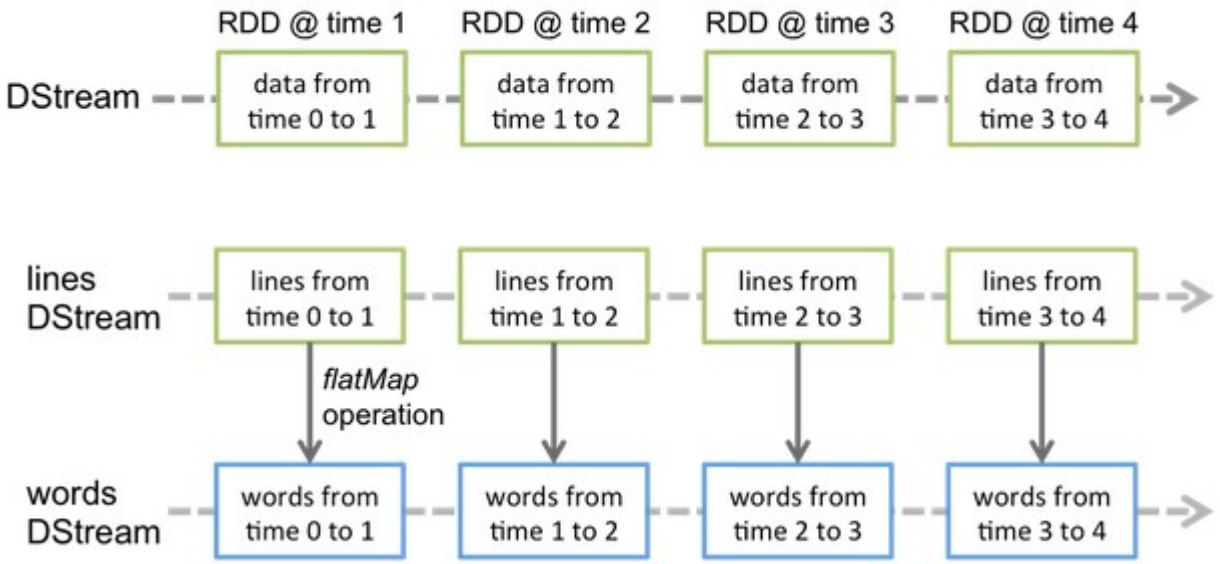


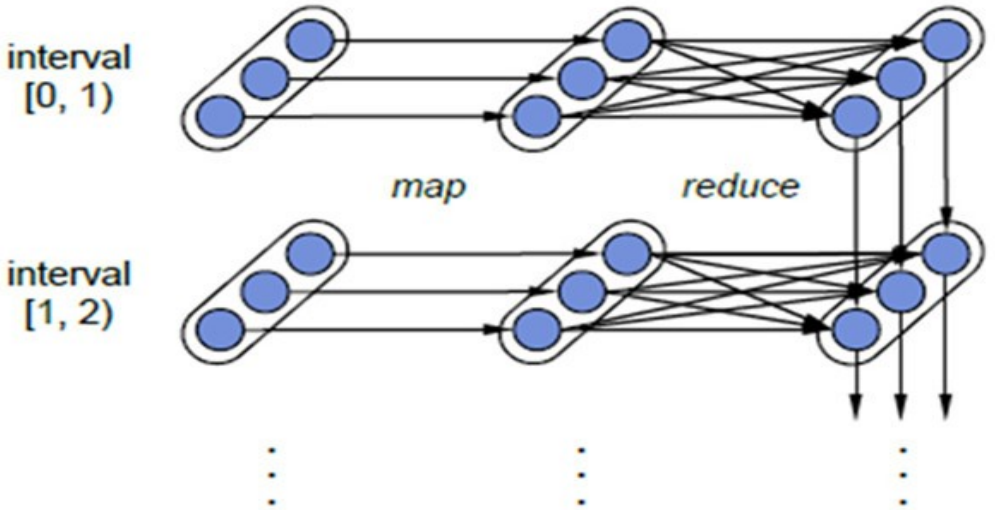
Spark Streaming 运行原理



Spark Streaming 运行原理

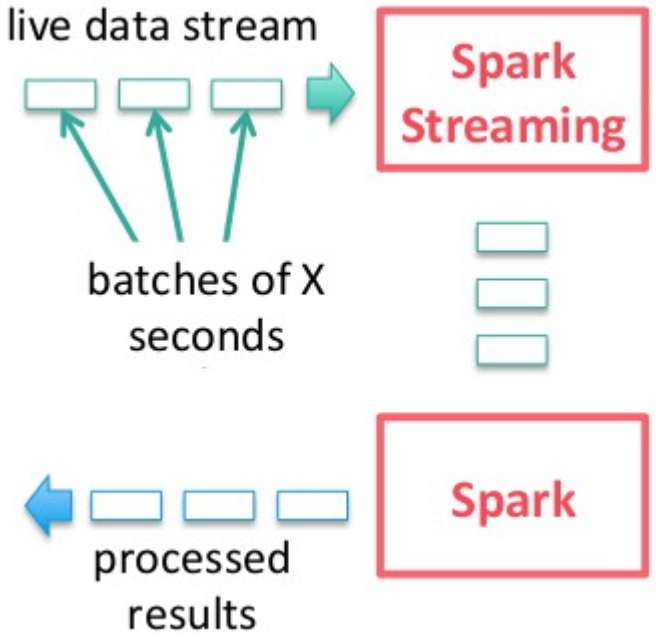
```
// 创建StreamingContext，1秒一个批次
val ssc = new StreamingContext(sparkConf, Seconds(1));

// 获得一个DStream负责连接 监听端口:地址
val lines = ssc.socketTextStream(serverIP, serverPort);

// 对每一行数据执行Split操作
val words = lines.flatMap(_.split(" "));
```





Spark Streaming 运行原理

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

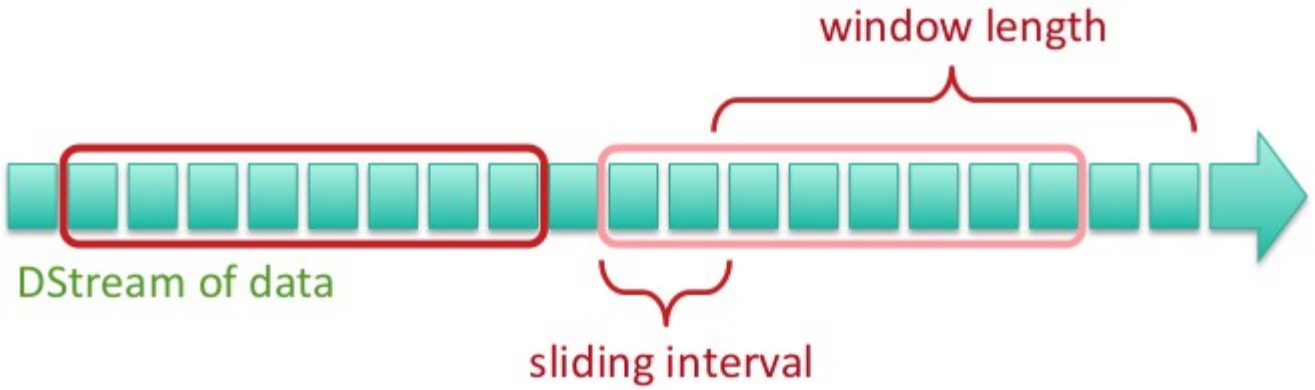


- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches

- Batch sizes as low as ½ sec, latency of about 1 sec
- Potential for combining batch processing and streaming processing in the same system

Spark Streaming 三种运用场景

- 无状态操作
- 状态操作
  - UpdateStateByKey
- window 操作



编程模型 DStream

- Discretized Stream
- 表示为数据流，实现为 RDD 序列
- 从流输入源创建，从现有 DStreams 通过 transformation 操作转换而来

```
 * This class contains the basic operations available on all DStreams, such as `map`, `filter` and
 * `window`. In addition, [[org.apache.spark.streaming.dstream.PairDStreamFunctions]] contains
 * operations available only on DStreams of key-value pairs, such as `groupByKeyAndWindow` and
 * `join`. These operations are automatically available on any DStream of pairs
 * (e.g., DStream[(Int, Int)] through implicit conversions when
 * `org.apache.spark.streaming.StreamingContext._` is imported.
 *
 * DStreams internally is characterized by a few basic properties:
 *  - A list of other DStreams that the DStream depends on
 *  - A time interval at which the DStream generates an RDD
 *  - A function that is used to generate an RDD after each time interval
 */

abstract class DStream[T: ClassTag] (
    @transient private[streaming] var ssc: StreamingContext
  ) extends Serializable with Logging {

  // ======================================================================
  // Methods that should be implemented by subclasses of DStream
  // ======================================================================

  /** Time interval after which the DStream generates a RDD */
  def slideDuration: Duration

  /** List of parent DStreams on which this DStream depends on */
  def dependencies: List[DStream[_]]

  /** Method that generates a RDD for the given time */
  def compute (validTime: Time): Option[RDD[T]]
```

编程模型 DStream

- Input

Input Sources

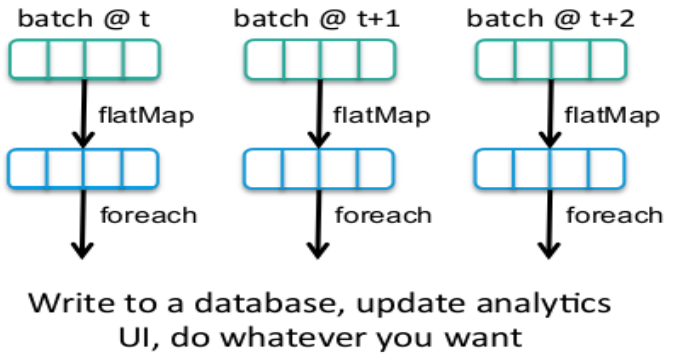
- Out of the box, we provide
  - Kafka, Flume, Akka Actors, Raw TCP sockets, HDFS, etc.
- Very easy to write a custom *receiver*
  - Define what to when receiver is started and stopped
- Also, generate your own sequence of RDDs, etc. and push them in as a “stream”

Source	Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

编程模型 DStream

Output

Output Operation	Meaning
print()	Prints first ten elements of every batch of data in a DStream on the driver.
foreachRDD(*func*)	The fundamental output operator. Applies a function, *func*, to each RDD generated from the stream. This function should have side effects, such as printing output, saving the RDD to external files, or writing it over the network to an external system.
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as a SequenceFile of serialized objects. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as a text files. 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 a Hadoop file. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.



编程模型 DStream

Transformation(1)

map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.
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*, [*numTasks*])	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*, [*numTasks*])	When called on DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.

编程模型 DStream

Transformation(2)

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.

Arbitrary Combinations of Batch and Streaming Computations

Inter-mix RDD and DStream operations!

- Example: Join incoming tweets with a spam HDFS file to filter out bad tweets

```
tweets.transform(tweetsRDD => {
    tweetsRDD.join(spamFile).filter(...)
})
```

Arbitrary Stateful Computations

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

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

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

val moods = tweetsByUser.updateStateByKey(updateMood _)
```

编程模型 DStream

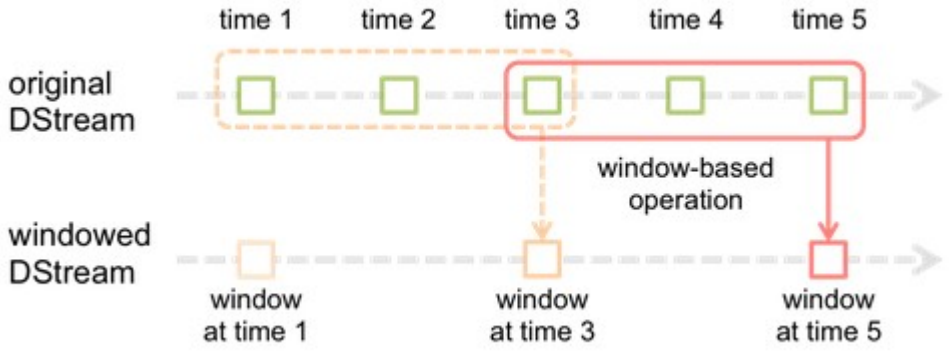
Transformation(3)

Transformation	Meaning
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 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.
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 enter the sliding window, and "inverse reducing" the old data that leave the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable to only "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.
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.

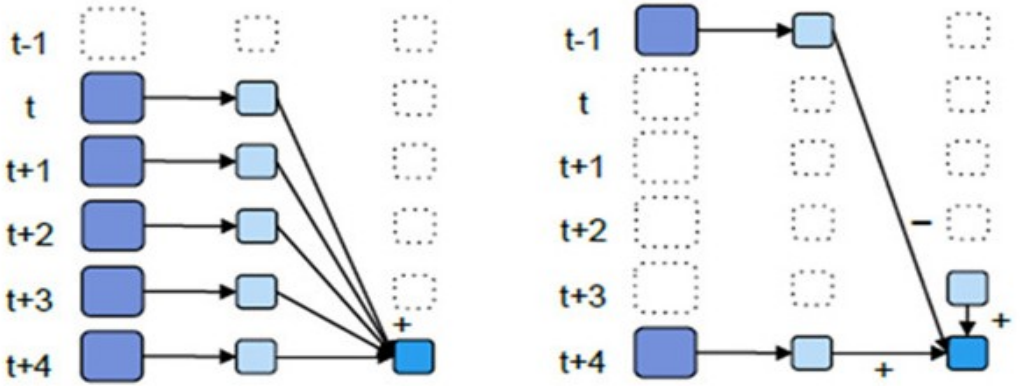
编程模型 DStream

- Transformation(3)

window 操作




```
val wordCounts = words.map(x => (x, 1)).reduceByKeyAndWindow(_ + _, Seconds(5s) , seconds(1))
```




```
val wordCounts = words.map(x => (x, 1)).reduceByKeyAndWindow(_ + _, _ -_, Seconds(5s) , seconds(1))
```

编程模型 DStream

DStreams + RDDs = Power

- Online machine learning
  - Continuously learn and update data models (*updateStateByKey* and *transform*)
- Combine live data streams with historical data
  - Generate historical data models with Spark, etc.
  - Use data models to process live data stream (*transform*)
- CEP-style processing
  - window-based operations (reduceByWindow, etc.)

CEP(Complex Event Processing，复杂事件处理)是近年来在互联网中不断升温一个词汇。在CEP的领域，国外已经有了很多研究成果和相当成熟的产品，而中国的这个市场才刚刚打开。流式数据处理是CEP的一个核心技术，流计算来自于一个信念：数据的价值随着时间的流逝而降低，所以事件出现后必须尽快地对它们进行处理，最好数据出现时便立刻对其进行处理，发生一个事件进行一次处理，而不是缓存成一批再处理。

持久化和容错

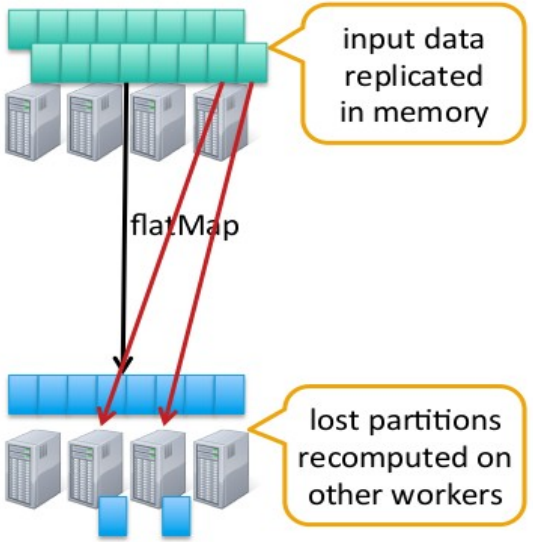
- 持久化
  - 允许用户调用 persist 来持久化
  - 默认的持久化： MEMORY_ONLY_SER
  - 对于来自网络的数据源 (Kafka, Flume, sockets 等) ： MEMORY_AND_DISK_SER_2
  - 对于 window 和 stateful 操作默认持久化

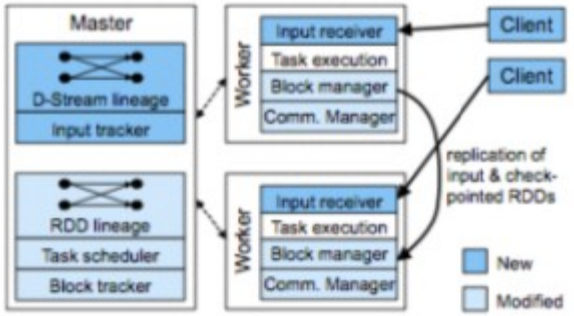
- checkpoint
  - 对于 window 和 stateful 操作必须 checkpoint
  - 通过 StreamingContext 的 checkpoint 来指定目录
  - 通过 DStream 的 checkpoint 指定间隔时间
  - 间隔必须是 slide interval 的倍数

```
def socketTextStream(
    hostname: String,
    port: Int,
    storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
  ): ReceiverInputDStream[String] = {
  socketStream[String](hostname, port, SocketReceiver.bytesToLines, storageLevel)
}
```

持久化和容错

- DStream 基于 RDD 组成， RDD 的容错性依旧有效
  - RDD 的某些 partition 丢失了，可以通过 lineage 信息重新计算恢复
  - 数据源来自外部文件系统，如 HDFS
    - 一定可以通过重新读取数据来恢复，绝对不会有数据丢失
  - 数据源来自网络
    - 默认会在两个不同节点加载数据到内存，一个节点 fail 了，系统可以通过另一个节点的数据重算
    - 假设正在运行 InputReceiver 的节点 fail 了，可能会丢失一部分数据





优化

- 利用集群资源，减少处理每个批次的数据的时间
  - 控制 reduce 数量，太多的 reducer, 造成很多的小任务，以此产生很多启动任务的开销。太少的 reducer, 任务执行行慢！
    - spark.streaming.blockInterval
    - inputStream.repartition
    - spark.default.parallelism
  - 序列化
    - 输入数据序列化
    - RDD 序列化
    - TASK 序列化
  - 在 Standalone 及 coarse-grained 模式下的任务启动要比 fine-grained 省时
- 给每个批次的数据量的设定一个合适的大小，原则：要来得及消化流进系统的数据
- 内存调优
  - 清理缓存的 RDD
    - 在 spark.cleaner.ttl 之前缓存的 RDD 都会被清除掉
    - 设置 spark.streaming.unpersis, 系统为你分忧
    - 使用并发垃圾收集器

本周内容

- Storm 简介
- Spark Streaming 原理
  - 运行原理
  - 三种运用场景
  - 编程模型 DStream
  - 持久化和容错
  - 优化
- 实例演示
  - 网络数据处理
  - 文本数据处理
  - Stateful 操作
  - window 操作

课程回顾

- Streaming 的原理
- Streaming 的三种应用场景
- MLlib
  - K-means
  - 推荐系统

Thanks

FAQ 时间