第八章　流计算系统

Spark Streaming

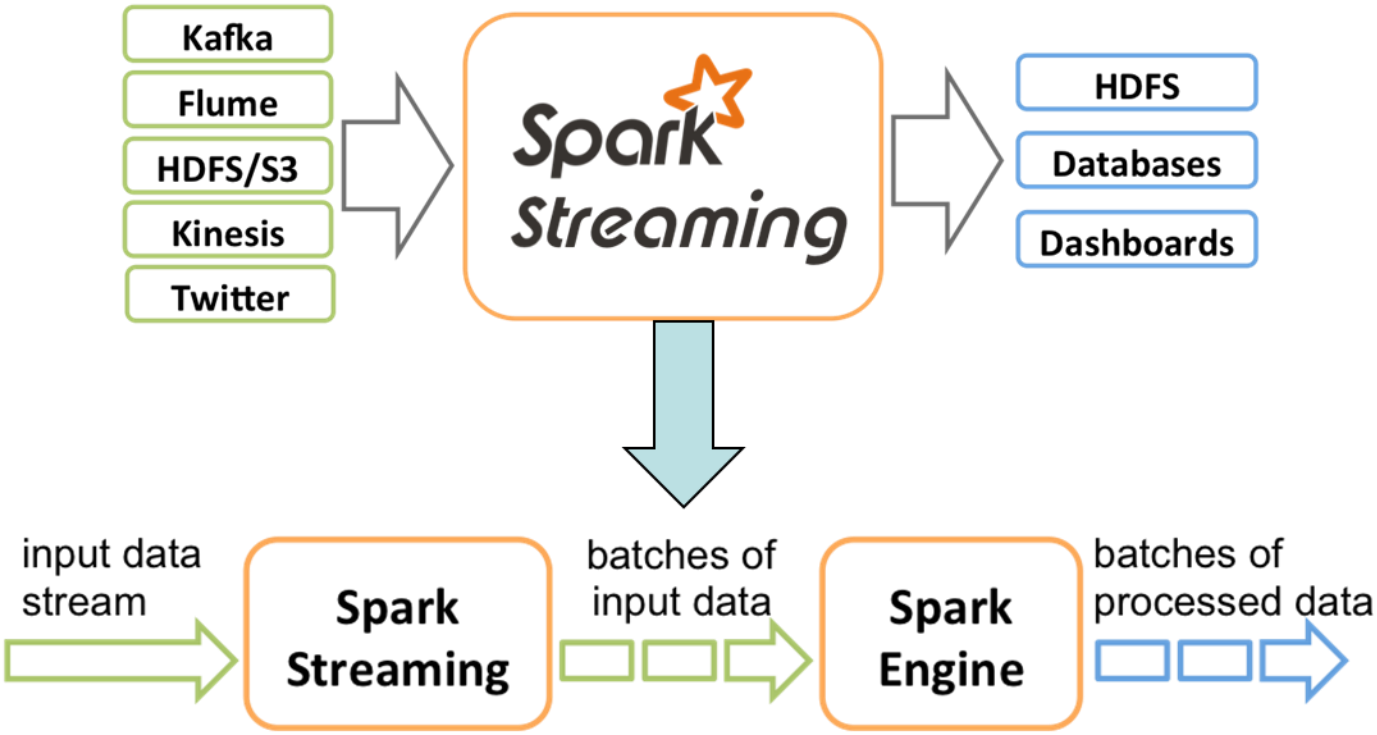
徐　辰

cxu@dase.ecnu.edu.cn

華東師範大學

什么是Spark Streaming?

- 为大规模流数据处理而实现的Spark扩展



大纲

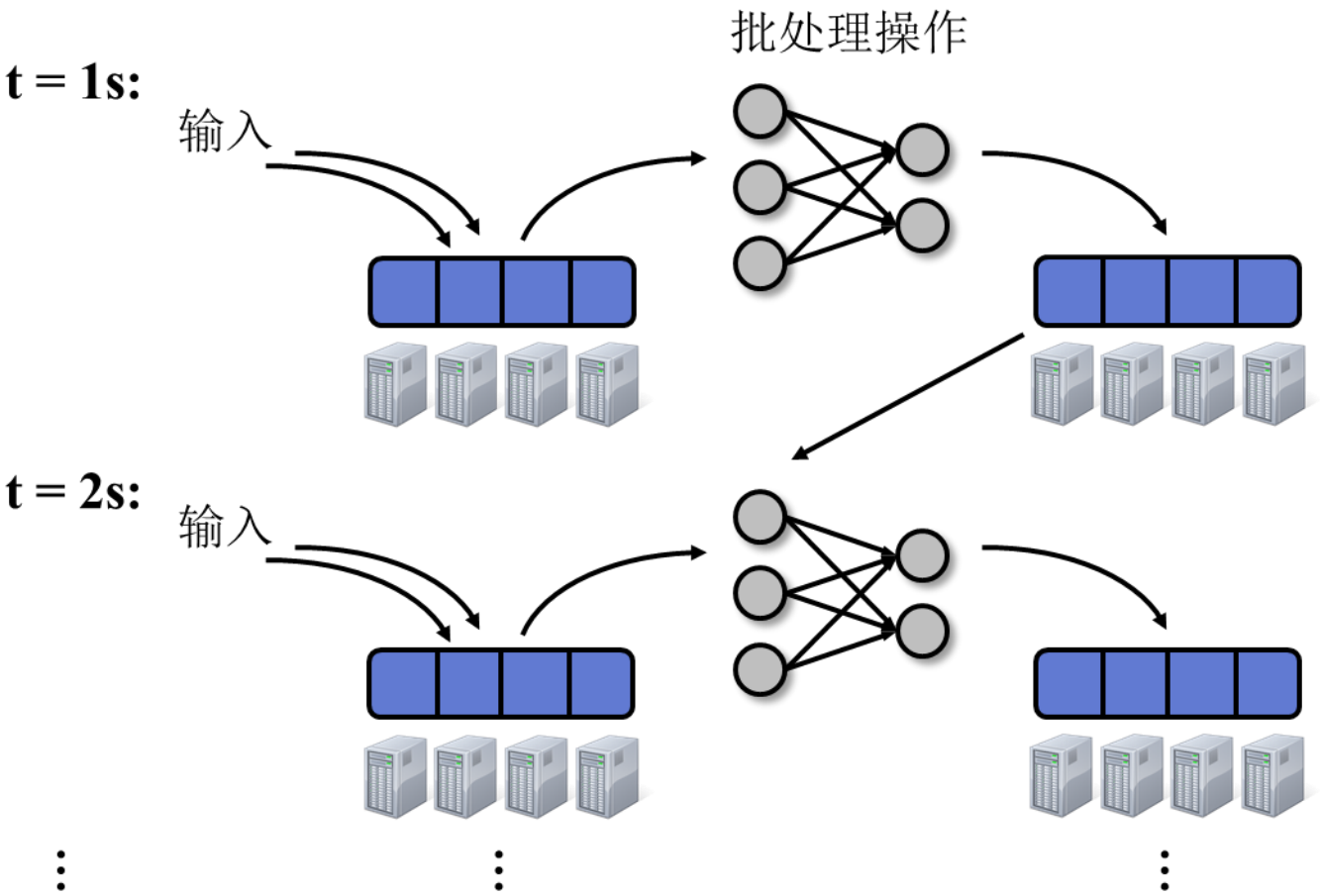
- 设计思想
  - 微批处理
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

基本思想

- 针对批处理系统进行一定改造，将流计算作业转化为一组微小的批处理作业

- 批处理系统能够较快地执行这些微小的批处理作业，从而满足流计算应用低延迟的需求

微批处理(Micro-batch processing)



大纲

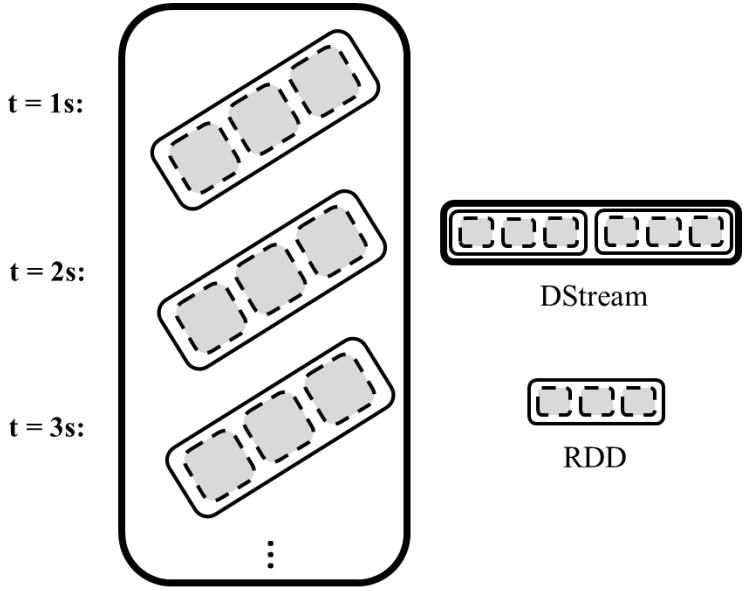
- 设计思想
  - 微批处理
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

流数据的离散化

- 在Storm的数据模型中，我们将流数据看作一系列连续的元组
- Spark Streaming将连续的流数据切片，即离散化，生成一系列小块数据

DStream数据模型

- 按照数据到来的时间间隔将连续的数据流离散化
- 得到的每一小批数据都是独立的RDD，一组RDD序列抽象为流数据的DStream

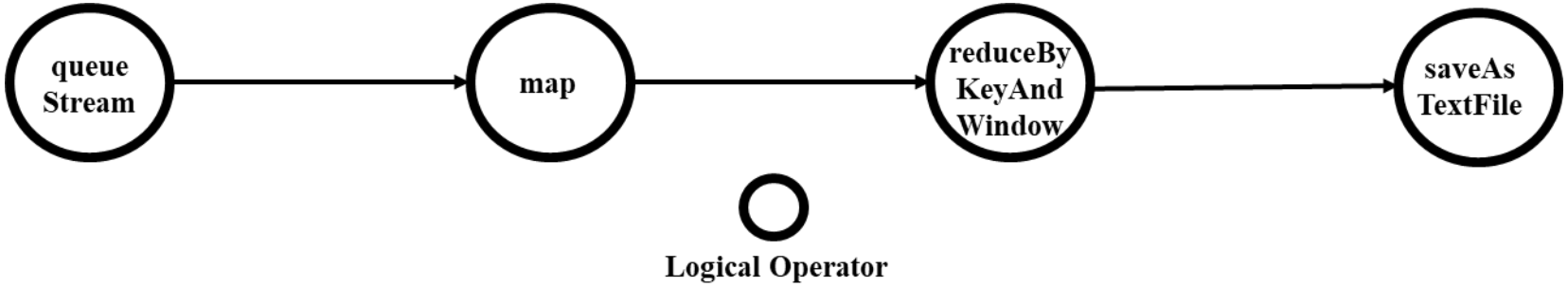


大纲

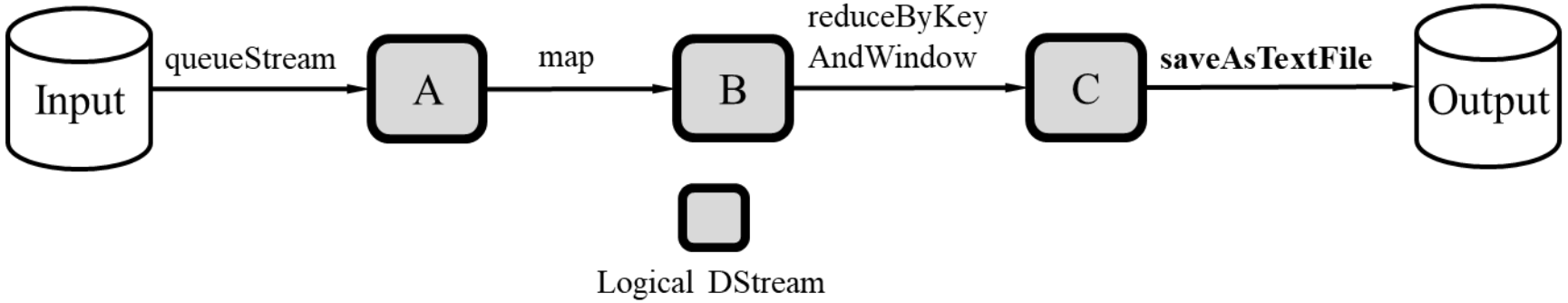
- 设计思想
  - 微批处理
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

逻辑计算模型

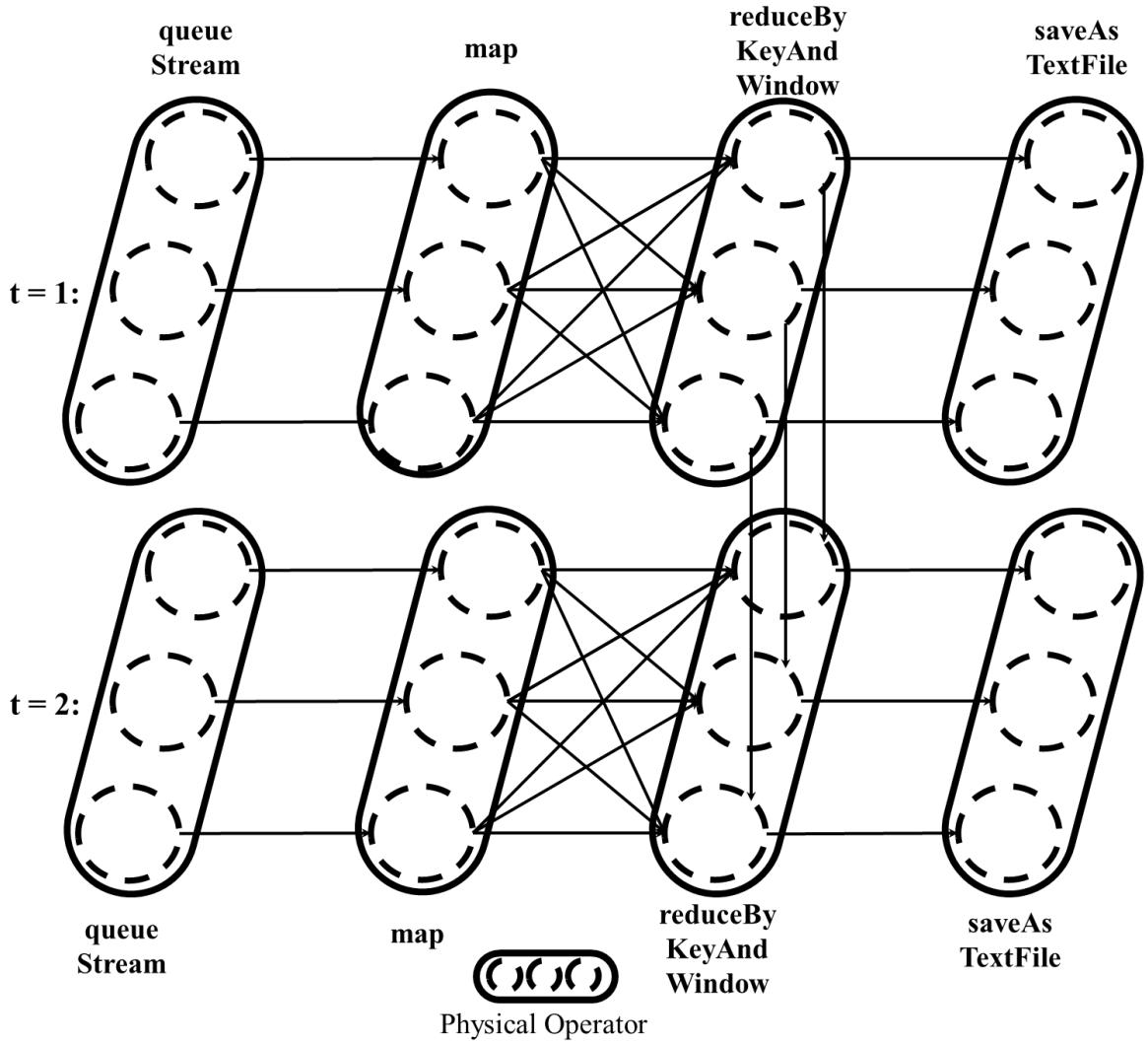
- Operator DAG：与RDD的operator类似



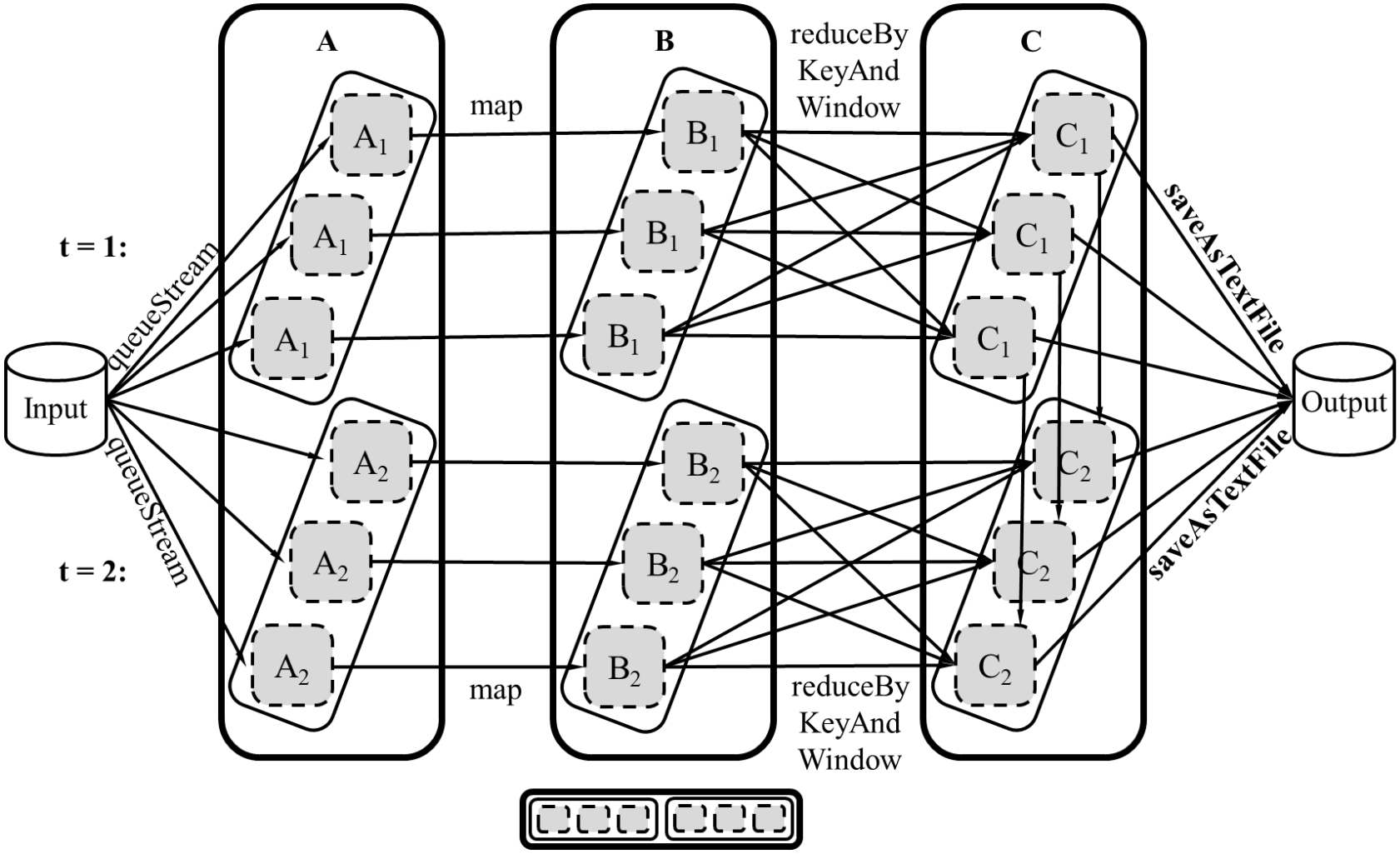
- DStream Lineage：借用RDD Lineage的概念



物理计算模型： Operator DAG



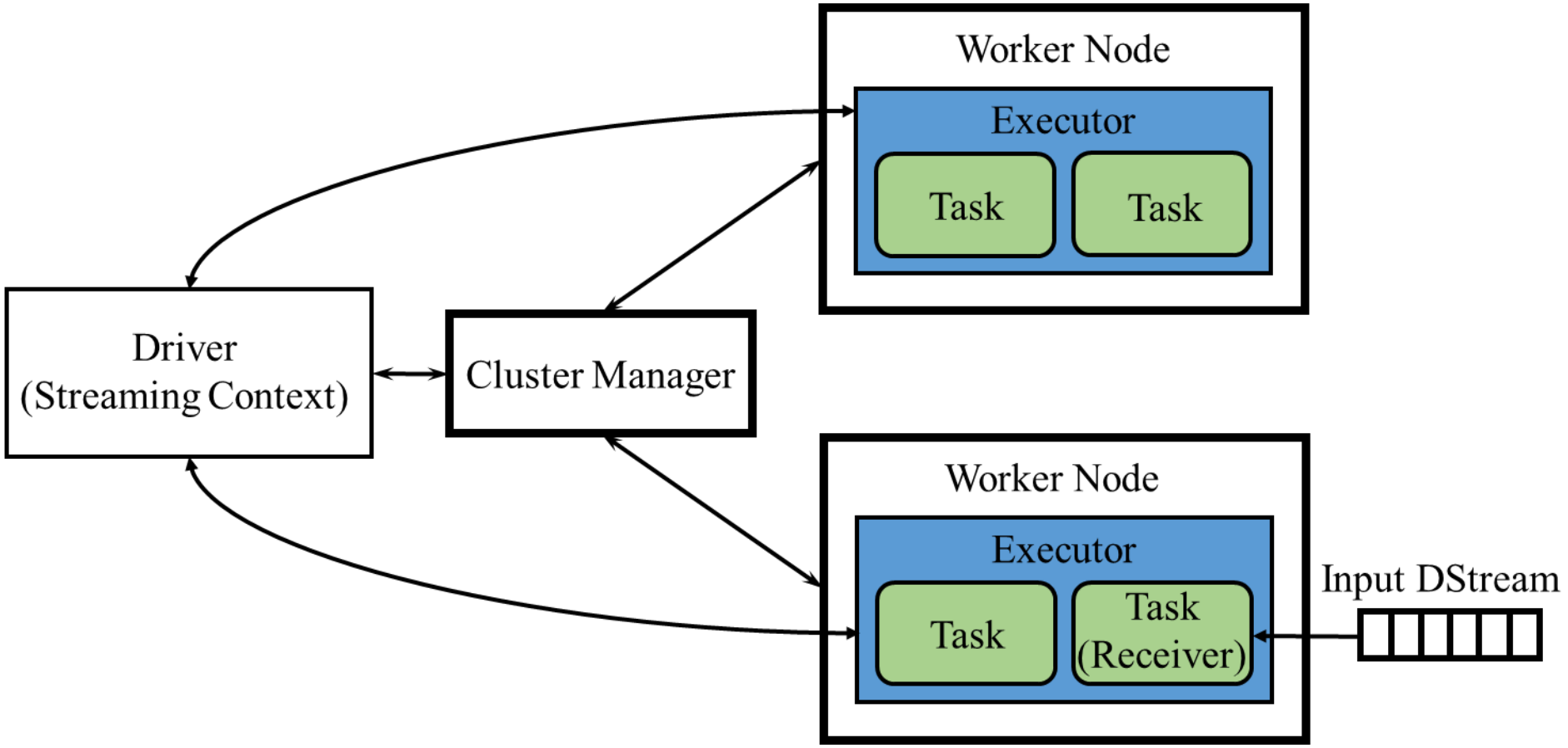
物理计算模型：DStream Lineage



大纲

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

架构图


Worker Node
Executor
Task
Task
Driver
(Streaming Context)
Cluster Manager
Worker Node
Executor
Task
Task
(Receiver)
Input DStream

工作部件

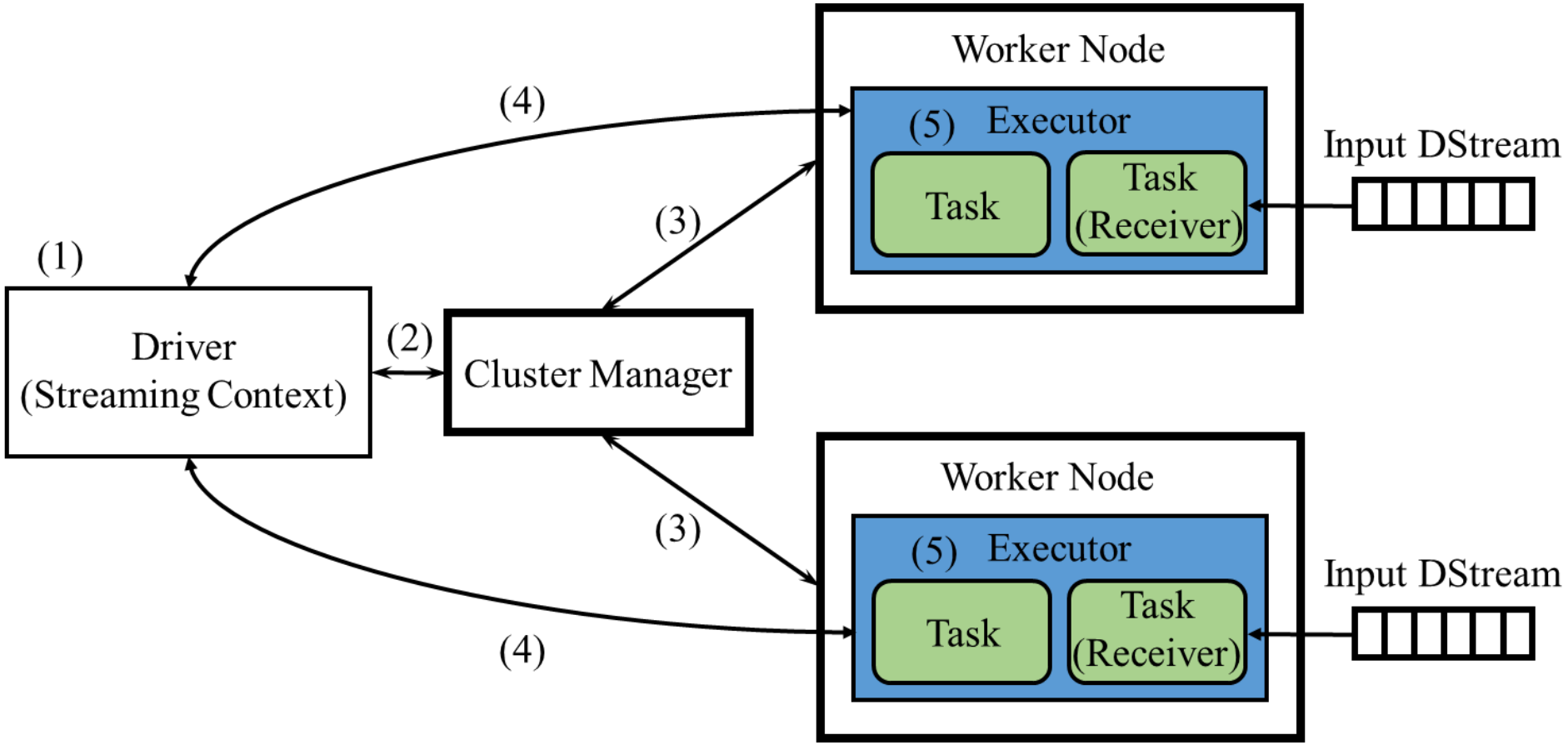
- Driver: Spark Streaming对SparkContext进行了扩充，构造了StreamingContext，用于管理流计算的元信息

- Executor: Executor中作为Receiver的某些task，负责从外部数据源源源不断的获取流数据，这和spark批处理读取数据的方式是不同的

大纲

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

应用程序执行流程


Worker Node
(5) Executor
Task
Task
(Receiver)
Input DStream
(4)
(3)
(1)
Driver
(Streaming Context)
(2)
Cluster Manager
Worker Node
(5) Executor
Task
Task
(Receiver)
Input DStream
(3)
(4)

应用程序执行流程

1. 启动Driver，以Standalone模式为例
   - 如果使用Client部署方式，客户端端直接启动Driver，并向Master注册
   - 如果使用Cluster部署方式，客户端将应用程序提交给Master，由Master选择一个Worker启动Driver进程(DriverWrapper)
2. 构建基本运行环境，即由Driver创建StreamingContext，向Cluster Manager进行资源申请，并由Driver进行任务分配和监控

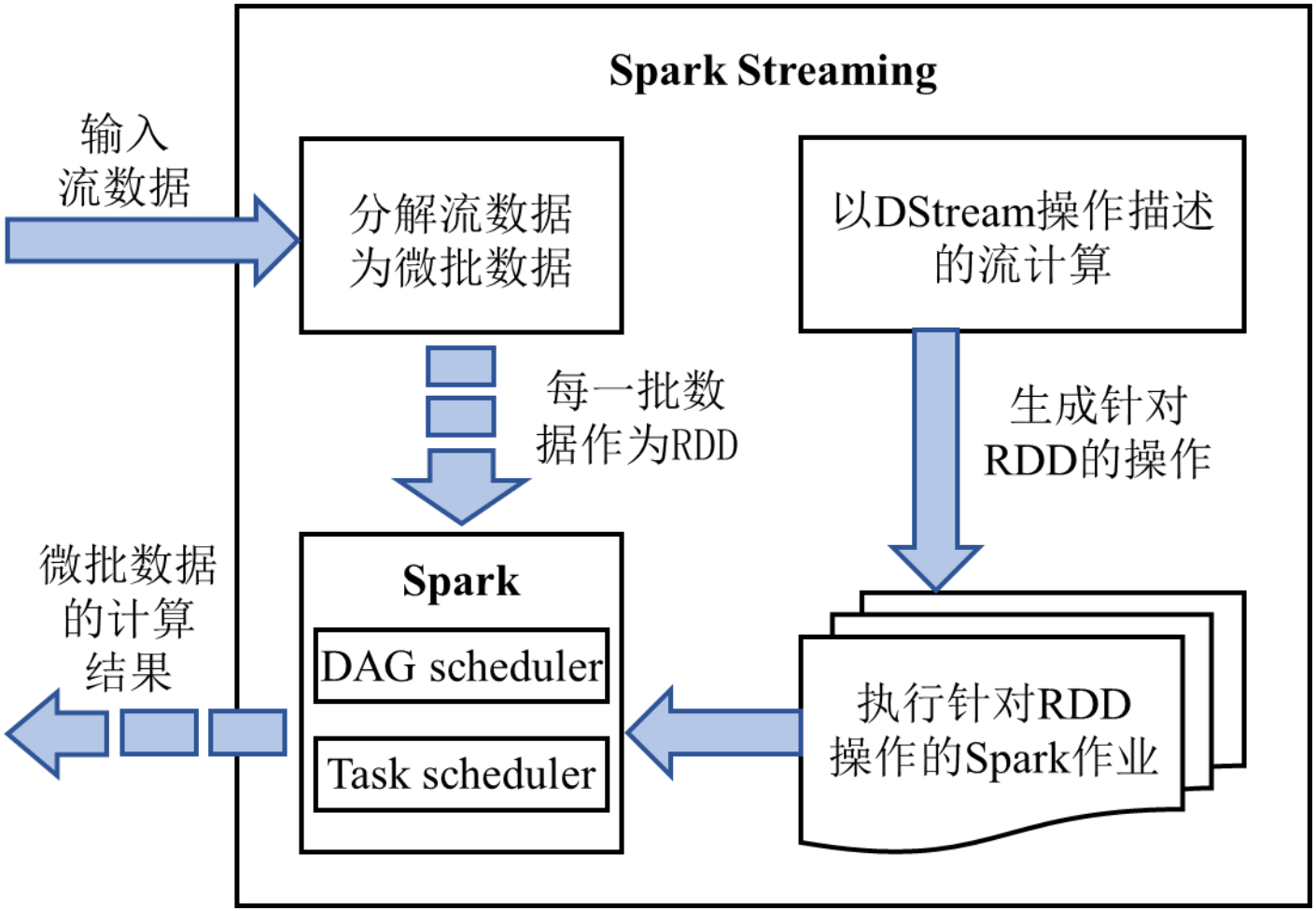
应用程序执行流程（续）

3. Cluster Manager通知工作节点启动Executor进程，该进程内部以多线程方式运行任务
4. Executor进程向Driver注册
5. StreamingContext构建关于RDD转换的DAG，从而交给Executor进程中的线程来执行任务

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

工作过程


Spark Streaming
输入
流数据
分解流数据
为微批数据
以DStream操作描述
的流计算
每一批数
据作为RDD
生成针对
RDD的操作
微批数据
的计算
结果
Spark
DAG scheduler
Task scheduler
执行针对RDD
操作的Spark作业

大纲

- 设计思想
- 体系架构
- 工作原理
  - 数据输入
  - 数据转换
  - 数据输出
- 容错机制
- 编程示例

输入操作

表 8.1　常见的DStream输入操作

转换	含义
socketTextStream(hostname, port)	从指定hostname:port的TCP连接创建一个DStream
textFileStream(directory)	监听HDFS兼容的文件系统下的文件夹中新产生的文件来创建DStream
fileStream[K, V, F](directory)	监听HDFS兼容的文件系统下的文件夹中新产生的文件，并指定InputFormat来创建DStream
queueStream(queue)	从RDD队列创建一个DStream

流数据读取

- 从外部数据源直接获取数据
  - 例如：从socket端口获取网络数据，或接收外部传感器产生的数据
  - 在两个工作节点进行备份
- 从外部存储系统周期性地读取数据
  - 例如，数据源将数据存入HDFS或Kafaka
  - 不进行备份

大纲

- 设计思想
- 体系架构
- 工作原理
  - 数据输入
  - 数据转换
  - 数据输出
- 容错机制
- 编程示例

转换操作

表 8.2　常见的DStream转换操作

序号	转换	含义
1	map(func)	对DStream中每个记录使用func转换，返回一个新的DStream
2	flatMap(func)	与map类似，但是对每个记录可以映射成0个或多个新的记录
3	filter(func)	过滤出对DStream中记录使用func后返回值为true的记录
4	reduceByKey(func)	将DStream中的键值对按键聚合，在每一个键的所有值上使用func，返回一个新的DStream
5	join(other)	对两个DStream中的RDD做join，返回一个新的DStream
6	cogroup(other)	[K, V1]和[K, V2]分别属于两个DStream，返回一个[K, (V1, V2)]组成的新的DStream
7	count()	得到当前批次中记录的个数，返回一个新的DStream
8	transform(func)	在DStream中每个RDD上应用一个RDD操作
9	window(windowDuration, slideDuration)	按指定窗口大小和滑动间隔划分，返回一个新的DStream
10	reduceByKeyAndWindow (func, windowDuration, slideDuration)	基于滑动窗口对[K, V]键值对的DStream中的值按键使用聚合函数func进行聚合操作，得到一个新的DStream
11	reduceByKeyAndWindow (func, invFunc, windowDuration, slideDuration)	使用逆函数的reduceByKeyAndWindow，实现对滑动窗口中的数据进行增量聚合
12	countByWindow	返回基于滑动窗口的DStream中的记录的个数
13	updateStateByKey (updateFunc)	通过在每个键的旧状态值和新数据值上使用updateFunc，更新每个键上的状态值

普通

transform

窗口

状态

Transform

- Transform
  - 允许用户在DStream的每个RDD上执行RDD操作
  - 例如：DStream与RDD之间的join

```
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...)
// RDD containing spam information

val cleanedDStream = wordCounts.transform { rdd =>
  rdd.join(spamInfoRDD).filter(...)
// join data stream with spam information to do data cleaning
  ...
}
```

https://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams

窗口操作

- 窗口(Window) 将流动的数据指定一定的计算范围，并且每隔一定间隔指定一次
  - 基于时间：Time-based window
  - 基于计数：Count-based window

- 窗口操作容许用户指定窗口的大小和间隔

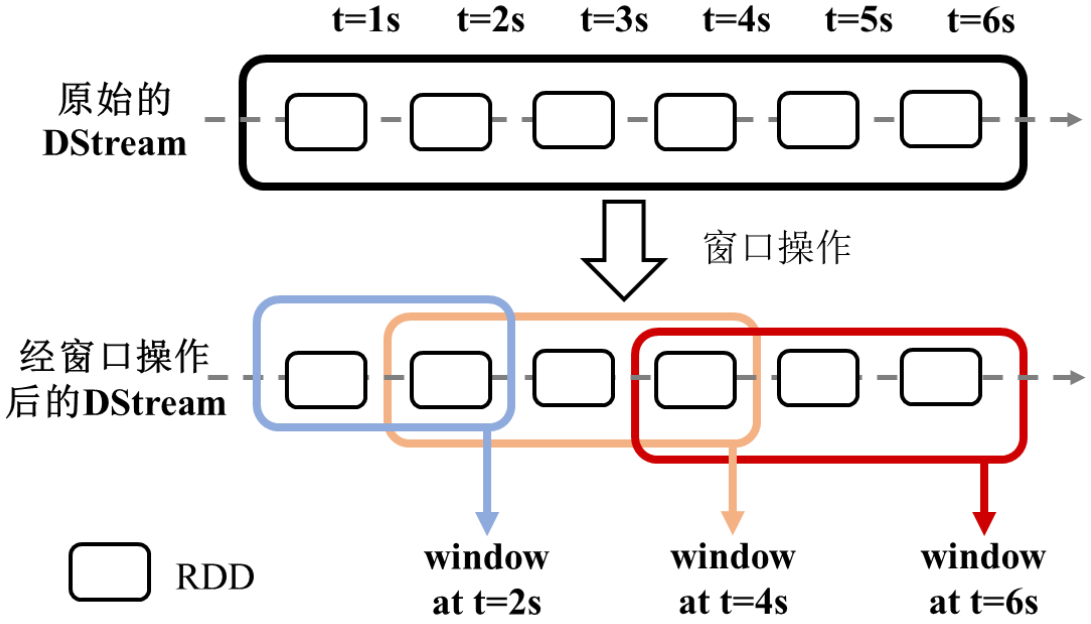
窗口操作

- Spark Streaming的微批处理是按照时间间隔进行划分的，因此Spark Streaming支持time-based window

- 例如，某一应用需要统计微博话题的讨论次数，统计的时间范围是最近1小时，并且每隔10分钟需要统计一次
  - 1小时：窗口的计算范围
  - 10分钟：是指窗口的间隔

滑动窗口

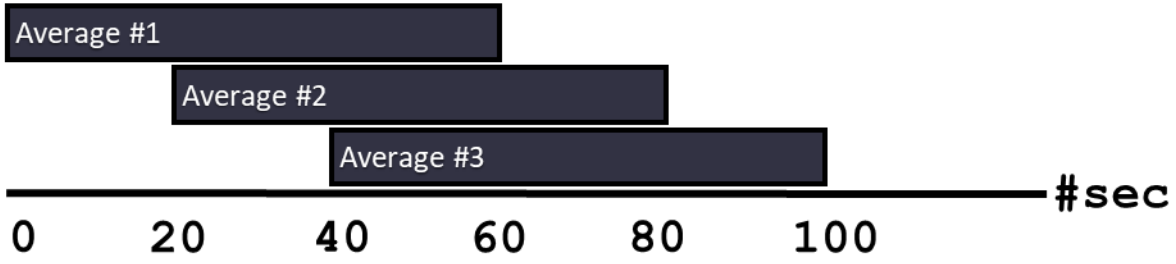
- 用户指定窗口时间间隔，窗口大小




```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Seconds(3), Seconds(2)).countByValue()
```

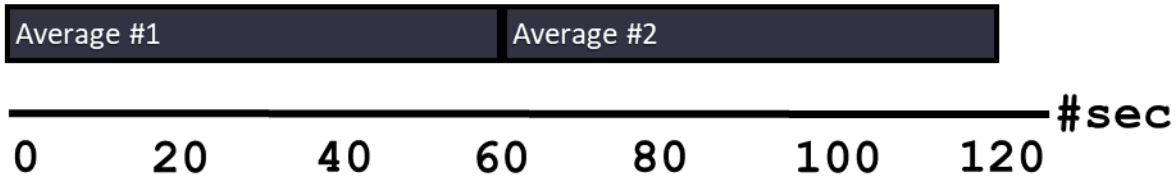
window operation　window length　sliding interval

窗口类型



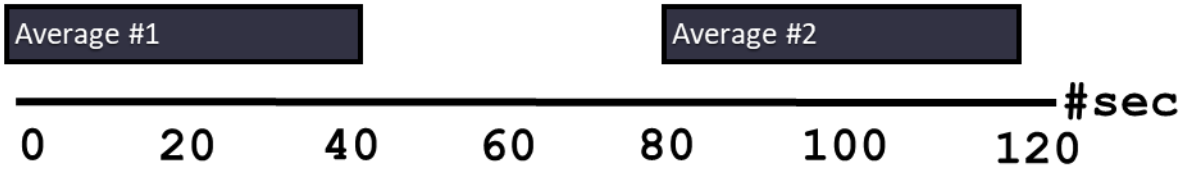
Sliding　　*range > slide*

滑动窗口



Tumbling　　*range = slide*

滚动窗口

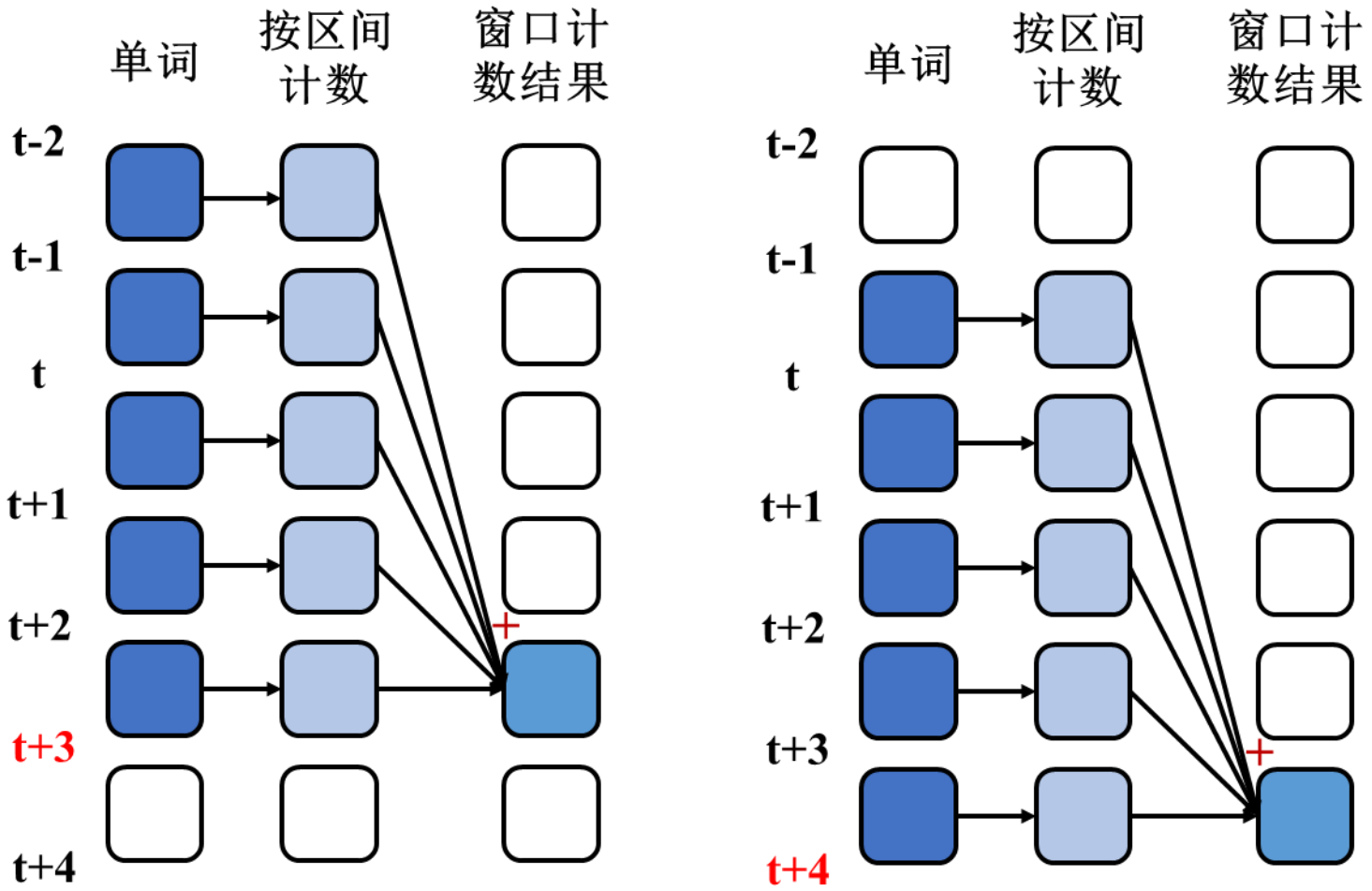


Jumping　　*range < slide*

跳跃窗口

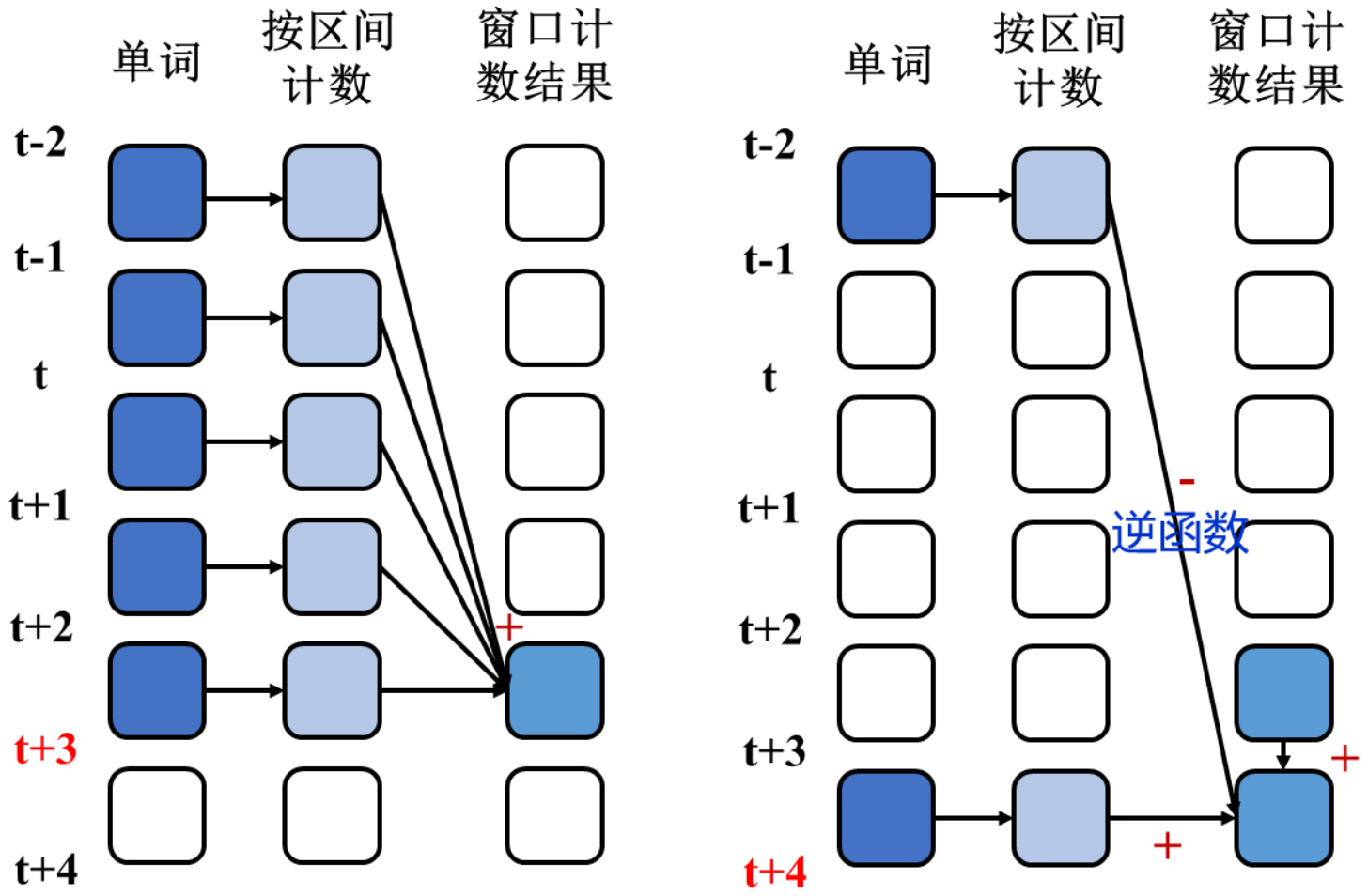
非增量式窗口操作

- 例子：统计过去5秒内的单词

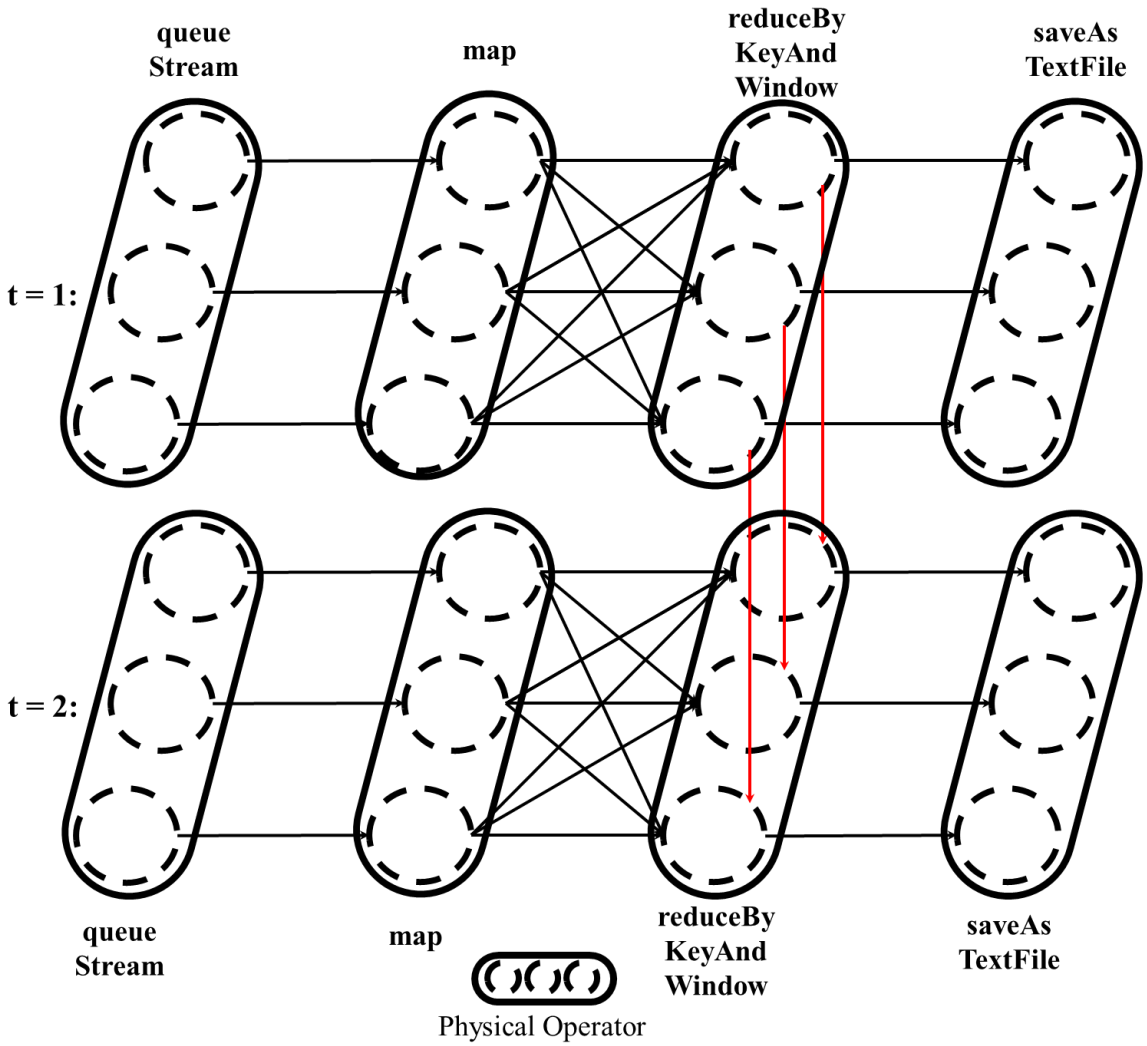


增量式窗口操作

- 例子：统计过去5秒内的单词

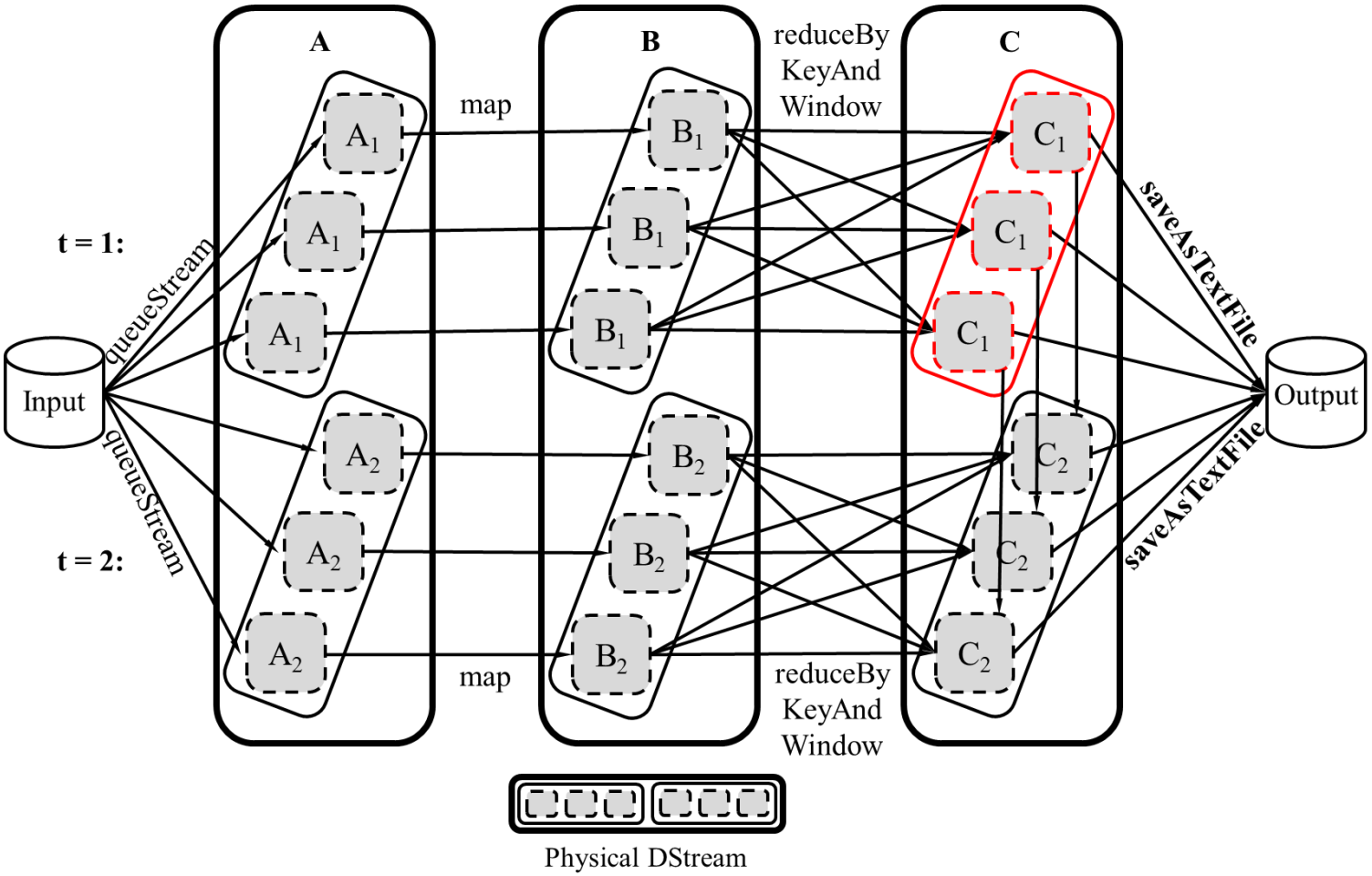


回顾：物理计算模型


queue Stream
map
reduceBy KeyAnd Window
saveAs TextFile
t = 1:
t = 2:
queue Stream
map
reduceBy KeyAnd Window
saveAs TextFile
Physical Operator

状态

- Window操作每隔2秒处理这2秒内的记录



状态操作

- 有状态的操作：需要涉及多个小批次数据

- 状态本质上是系统运行中产生的RDD，可以通过底层的Spark批处理引擎对RDD进行管理
  - UpdateStateByKey：针对状态进行转换的操作

大纲

- 设计思想
- 体系架构
- 工作原理
  - 数据输入
  - 数据转换
  - 数据输出
- 容错机制
- 编程示例

输出操作

- DStream中无action操作

表 8.3　常见的DStream输出操作

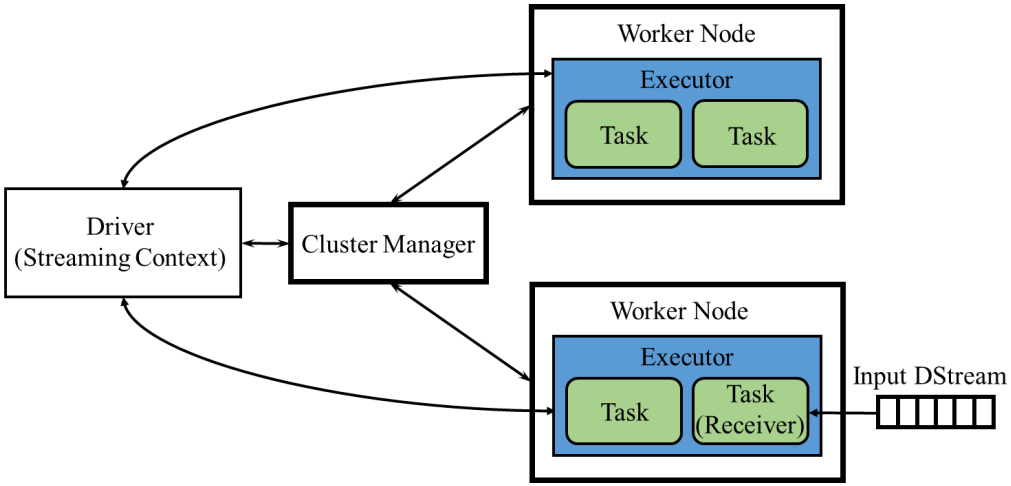
转换	含义
print()	打印DStream中前10个记录
saveAsTextFiles(prefix, [suffix])	将DStream中的记录以文本的形式保存为文本文件
foreachRDD(foreachFunc)	在DStream中的RDD上应用一个foreachFunc

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

故障类型

- Cluster Manager故障：不考虑
- Executor(Worker)故障
  - 不含Receiver：利用RDD Lineage进行恢复
  - 含有Receiver：利用日志进行恢复
- Driver故障：利用检查点进行恢复



大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 基于RDD Lineage的容错
  - 基于日志的容错
  - 基于检查点的容错
  - 端到端的容错语义
- 编程示例

不含Receiver的Executor故障

- 只有负责数据处理的任务受到了影响，系统只需要重启这个Executor再次处理数据就可以了
- 由于Spark Streaming系统底层依赖的是Spark批处理引擎，那么Executor里运行的任务实际上是底层Spark批处理引擎的任务
- 因Executor故障受到影响的负责数据处理的任务可以使用Spark批处理引擎的容错机制进行恢复

回顾：Spark批处理容错

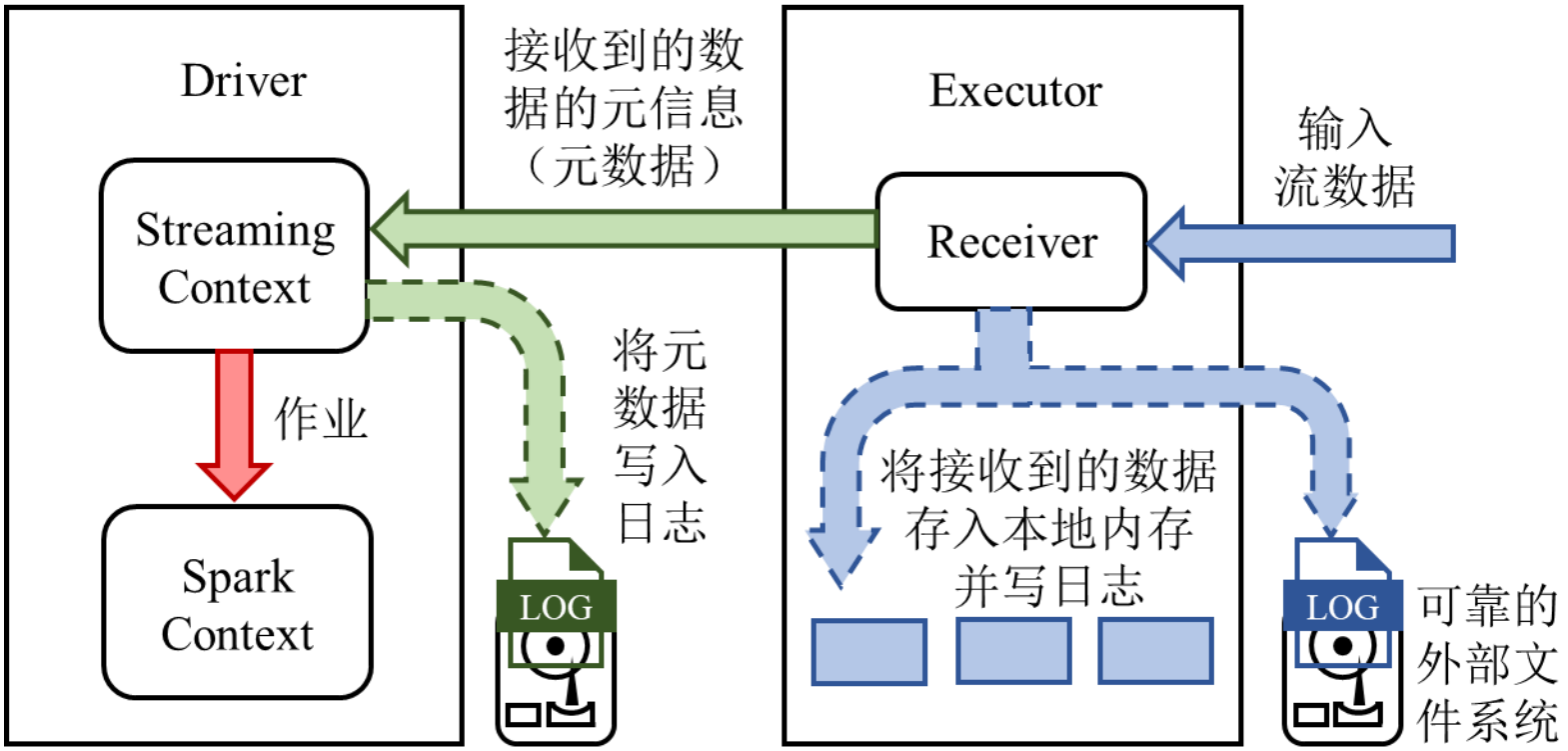
- RDD持久化
- RDD Lineage
- 检查点：数据检查点

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 基于RDD Lineage的容错
  - 基于日志的容错
  - 基于检查点的容错
  - 端到端的容错语义
- 编程示例

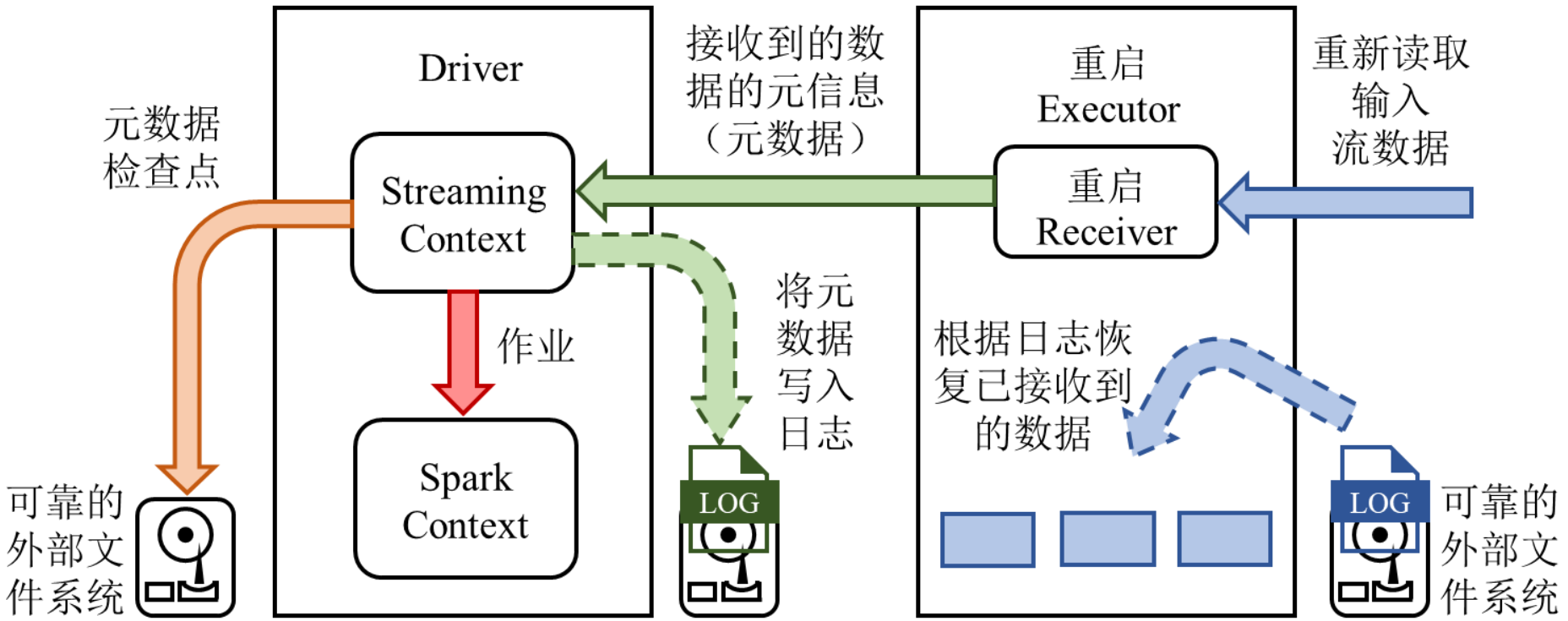
日志

- Receiver日志：哪些输入数据已做备份
- Driver日志：哪些输入数据已被处理



故障恢复

Executor(Worker)故障恢复



大纲

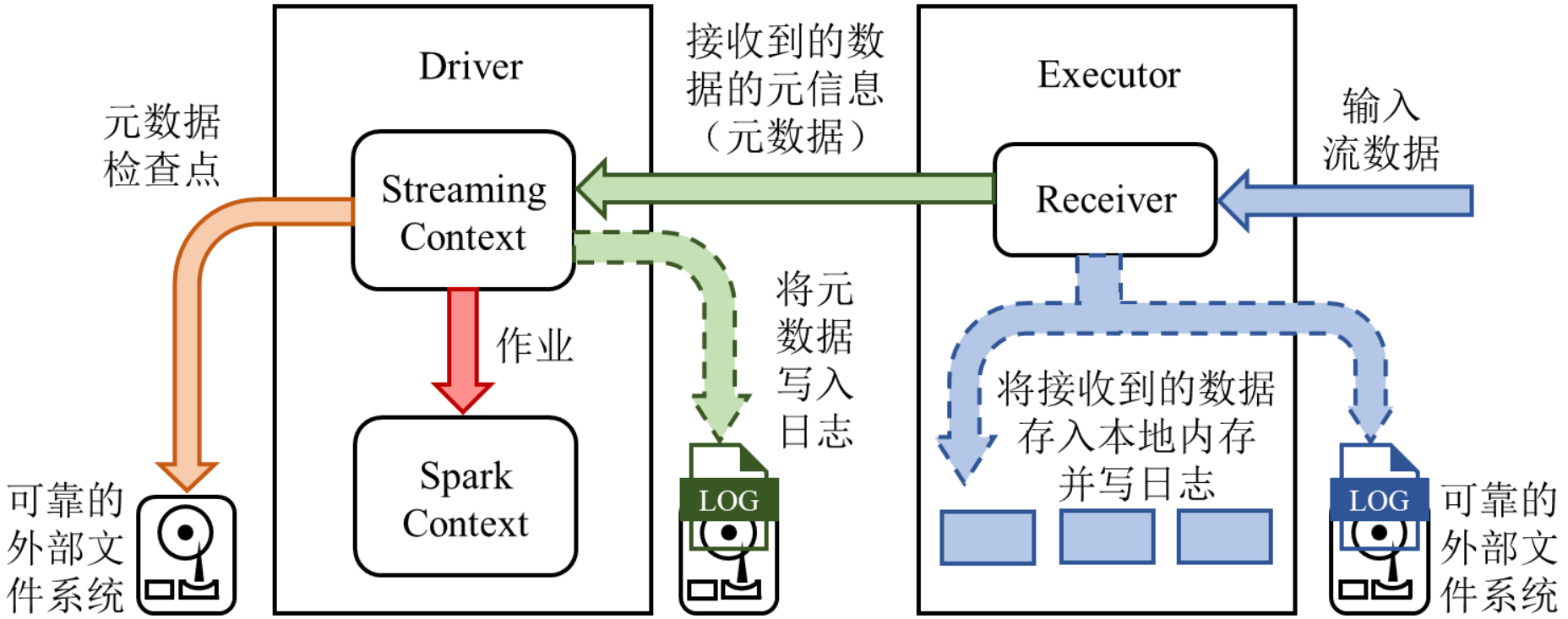
- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 基于RDD Lineage的容错
  - 基于日志的容错
  - 基于检查点的容错
  - 端到端的容错语义
- 编程示例

检查点

- 数据检查点：RDD检查点
- 元数据检查点
  - 配置信息：创建spark streaming应用程序的配置信息
  - DStream操作信息：定义了应用程序计算逻辑的DStream操作的信息
  - 未处理的batch信息：那些正在排队的作业中还没处理的batch信息

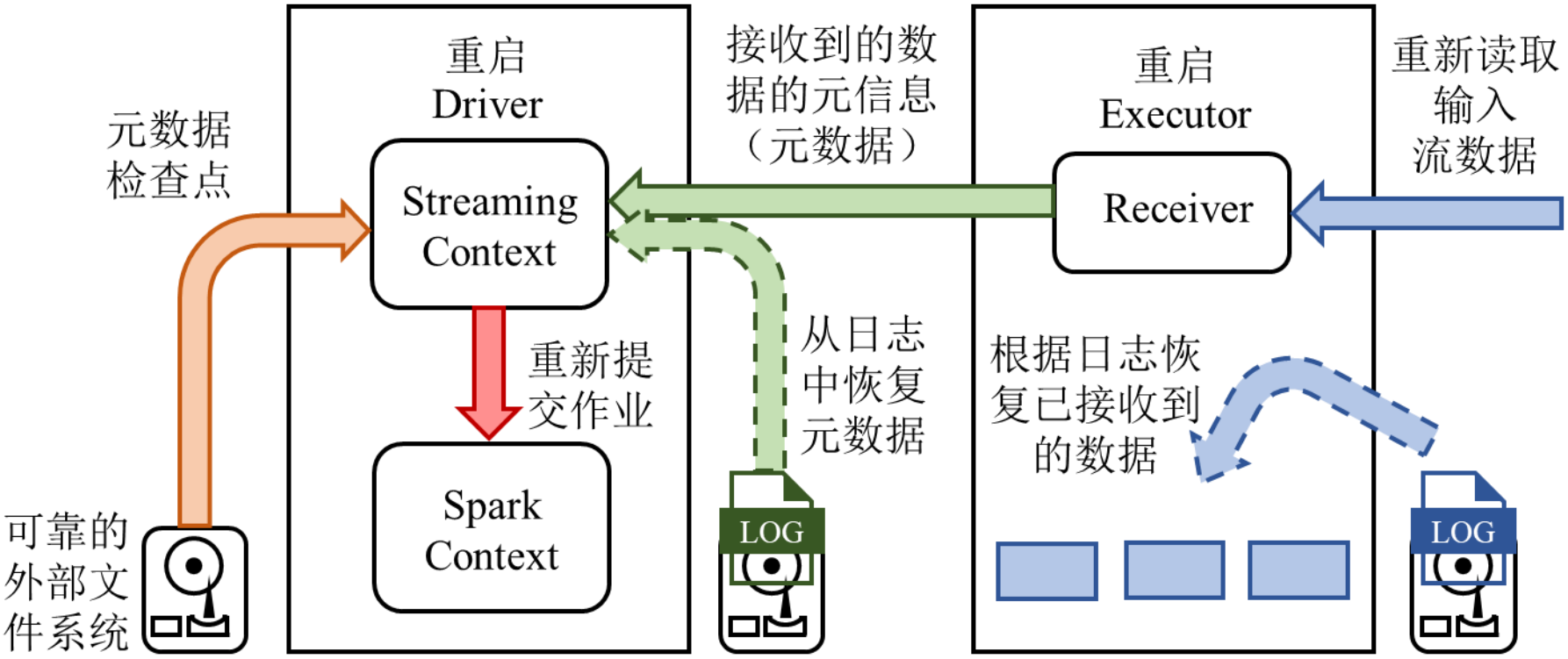
检查点

- 与写日志是同时进行的



故障恢复

Driver故障恢复



大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 基于RDD Lineage的容错
  - 基于日志的容错
  - 基于检查点的容错
  - 端到端的容错语义
- 编程示例

容错语义

- 流计算系统本身的容错语义

- 端到端的容错语义
  - 一个完整的流计算处理流程不仅涉及流计算系统本身，还涉及提供数据源和接收处理结果的系统，即端到端的过程

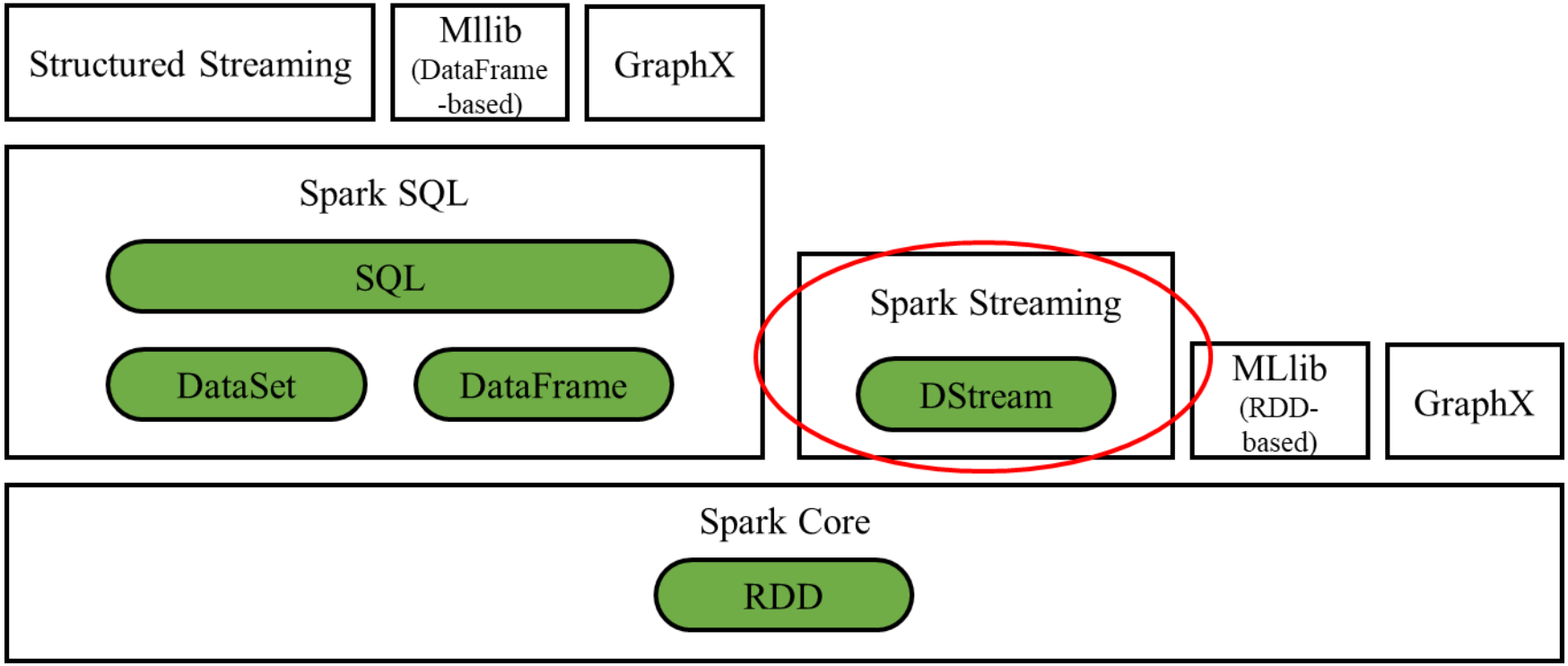
端到端的容错语义

- 接收数据: *at-least or exactly once*
  - 取决于数据是使用Receiver或其它方式从数据源接收的
- 转换数据: *exactly once*
  - 接收到的数据是用Dstream和RDD做转换的
- 输出数据: *at-least or exactly once*
  - 取决于最终的转换结果被推出到外部系统如文件系统，数据库，仪表盘等

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Spark API


Structured Streaming
Mllib
(DataFrame
-based)
GraphX
Spark SQL
SQL
DataSet
DataFrame
Spark Streaming
DStream
MLlib
(RDD-
based)
GraphX
Spark Core
RDD

Spark Streaming程序框架


```
object CustomApp {
  def run(args: Array[String]): Unit = {
    /* 步骤1：通过SparkConf设置配置信息，并创建StreamingContext */
    val conf = new SparkConf
      .setAppName("appName")
      .setMaster("local[*]") // 仅用于本地进行调试，如在集群中运行则删除该行
      ......
    val ssc = new StreamingContext(conf, batchDuration) //batchDuration表示批次间隔

    /* 步骤2：按应用逻辑使用操作算子编写DAG，包括DStream的输入、转换和输出等 */
    ......

    /* 步骤3：开启计算并等待计算结束 */
    ssc.start()
    ssc.awaitTermination()
  }
```

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 按批词频统计
  - 全局词频统计
  - 窗口操作
  - 异常检测

按批词频统计

- 每隔5秒统计一次该5秒内输入的文本中每个单词出现的次数。

时间	输入
1s	An An
2s	My
4s	Me
6s	An An
8s	My
9s	He
11s	My My
12s	An
14s	My
…	…

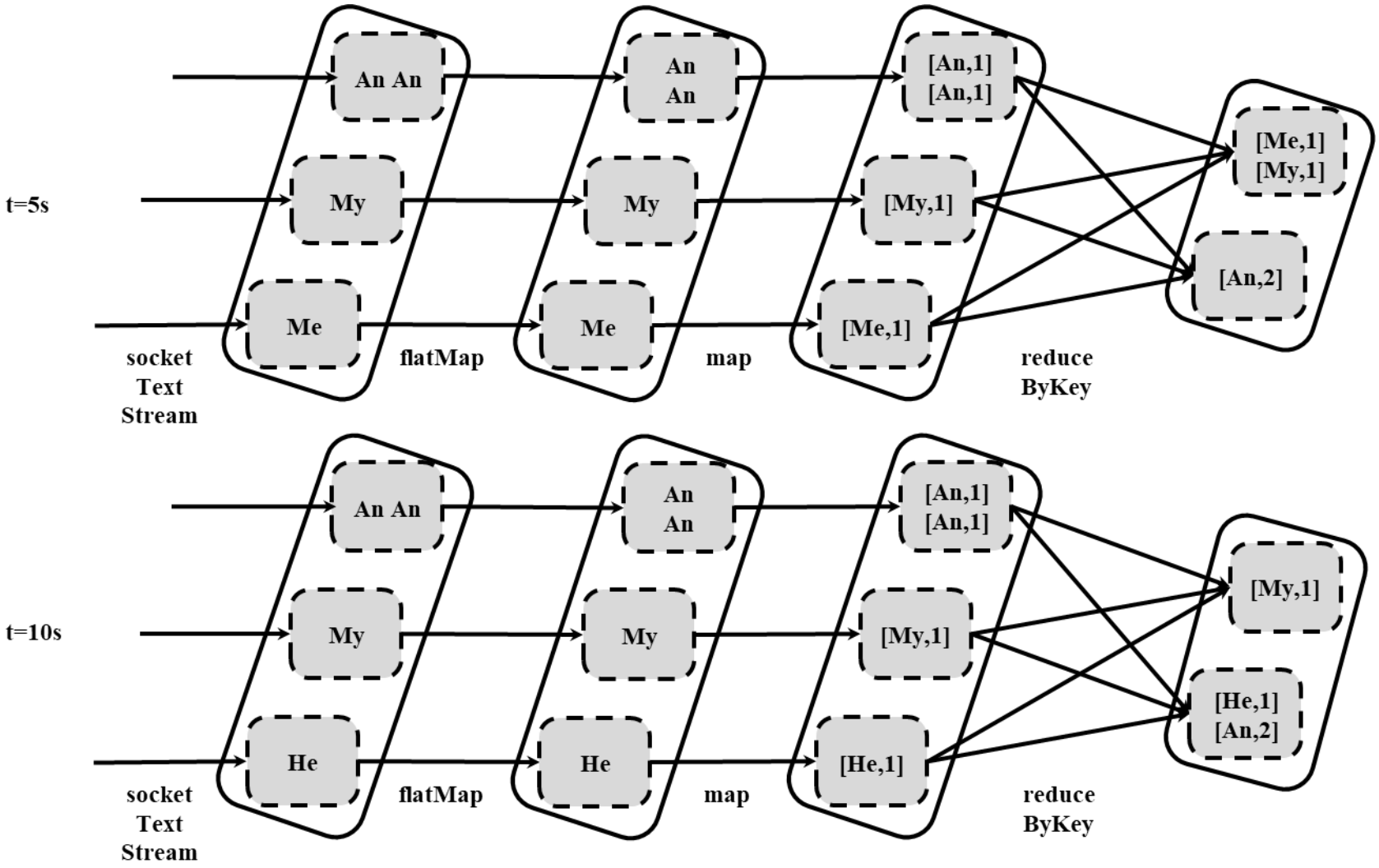
时间	输出
5s	An 2 My 1 Me 1
10s	An 2 My 1 He 1
15s	My 3 An 1
…	…

解决方案

- 因为要求每隔5秒对该5秒内的输入进行一次词频统计，因此创建StreamingContext时将批次间隔指定位5秒

- 对于每个微批数据上的词频统计，与Spark RDD的方式一样，用flatMap, map, reduceByKey算子来完成

运行过程


t=5s
An An
My
Me
An
An
My
Me
[An,1]
[An,1]
[My,1]
[Me,1]
[Me,1]
[My,1]
[An,2]
socket
Text
Stream
flatMap
map
reduce
ByKey
t=10s
An An
My
He
An
An
My
He
[An,1]
[An,1]
[My,1]
[He,1]
[My,1]
[He,1]
[An,2]
socket
Text
Stream
flatMap
map
reduce
ByKey

代码


```
object BatchWordCount {
  def run(args: Array[String]): Unit = {
    /* 步骤1：通过SparkConf设置配置信息，并创建StreamingContext*/
    val conf = new SparkConf
      .setAppName("BatchWordCount")
      .setMaster("local[*]") // 仅用于本地进行调试，如在集群中运行则删除该行
    val ssc = new StreamingContext(conf, Seconds(5))
    /* 步骤2：按应用逻辑使用操作算子编写DAG，包括DStream的输入、转换和输出等 */
    // 从指定的主机名和端口号接收数据
    val inputDStream = ssc.socketTextStream("hostname", port)
    // 将接收到的文本行数据按空格分割，并将每个单词映射为[word, 1]键值对
    val pairsDStream = inputDStream.flatMap(_.split(" ")).map(x => (x, 1))
    // 按单词聚合，对相同单词的频数进行累计
    val wordCounts = pairsDStream.reduceByKey((t1 : Int, t2: Int) => t1 + t2)
    // 打印结果
    wordCounts.print()

    /* 步骤3：开启计算并等待计算结束 */
    ssc.start()
    ssc.awaitTermination()
  }

  def main(args: Array[String]): Unit = {
    run(args)
  }
}
```

创建 StreamingContext 时将批次间隔指定为5秒

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 按批词频统计
  - 全局词频统计
  - 窗口操作
  - 异常检测

全局词频统计

- 每隔5秒统计到目前为止已经接收到的所有输入数据中每个单词出现的次数

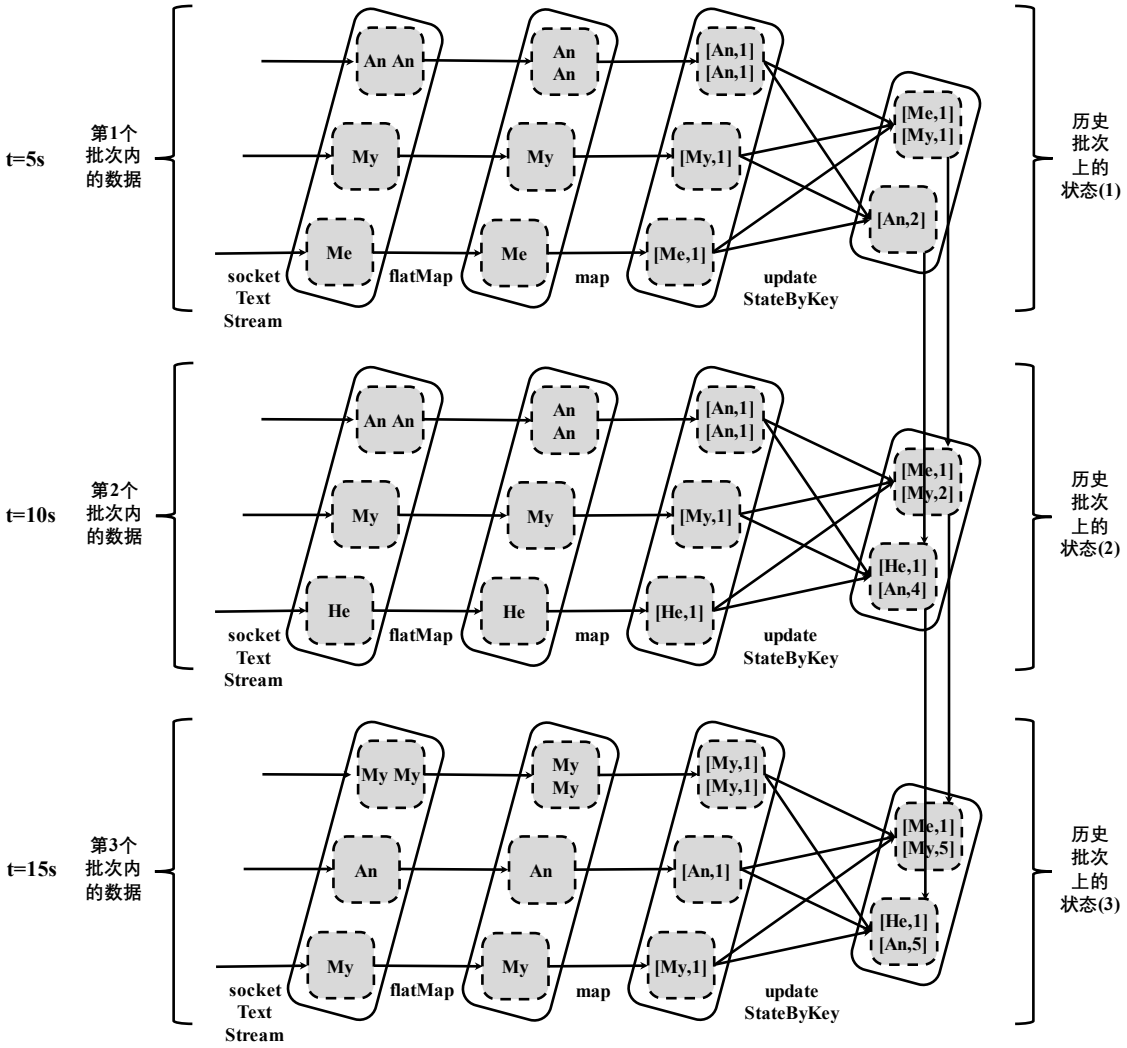
时间	输入
1s	An An
2s	My
4s	Me
6s	An An
8s	My
9s	He
11s	My My
12s	An
14s	My
…	…

时间	输出
5s	An 2 My 1 Me 1
10s	An 4 My 2 Me 1 He 1
15s	An 5 My 5 Me 1 He 1
…	…

解决方案

- 对历史批次中每个单词出现的次数累加上当前批次中每个单词出现的次数，从而得到所有输入数据上的词频结果
- 实现有状态词频统计的关键是为所有历史批次中的数据维护一个状态，这可以借助updateStateByKey 状态操作来实现

运行过程


t=5s
第1个批次内的数据
An An
My
Me
An An
My
Me
[An,1] [An,1]
[My,1]
[Me,1]
[Me,1] [My,1]
[An,2]
历史批次上的状态(1)
socket Text Stream
flatMap
map
update StateByKey
t=10s
第2个批次内的数据
An An
My
He
An An
My
He
[An,1] [An,1]
[My,1]
[He,1]
[Me,1] [My,2]
[He,1] [An,4]
历史批次上的状态(2)
socket Text Stream
flatMap
map
update StateByKey
t=15s
第3个批次内的数据
My My
An
My
My My
An
My
[My,1] [My,1]
[An,1]
[My,1]
[Me,1] [My,5]
[He,1] [An,5]
历史批次上的状态(3)
socket Text Stream
flatMap
map
update StateByKey

代码


```scala
object GlobalWordCount {
  def run(args: Array[String]): Unit = {
    /* 步骤1：通过SparkConf设置配置信息，并创建StreamingContext*/
    val conf = new SparkConf
      .setAppName("GlobalWordCount")
      .setMaster("local[*]") // 仅用于本地进行调试，如在集群中运行则删除该行
    val ssc = new StreamingContext(conf, Seconds(5))

    // 若使用了updateStateByKey状态算子，则必须设置检查点路径
    ssc.checkpoint("hdfs://...")

    /* 步骤2：按应用逻辑使用操作算子编写DAG，包括DStream的输入、转换和输出等 */
    val inputDStream = ssc.socketTextStream("hostname", port)
    val pairsDStream = inputDStream.flatMap(_.split(" ")).map(x => (x, 1))

    // 使用updateStateByKey根据状态值和新到达数据统计词频
    val wordCounts = pairDStream.updateStateByKey(
      (curValues:Seq[Int], preValue:Option[Int]) => {
        val curValue = curValues.sum
        Some(curValue + preValue.getOrElse(0))
      })
      .checkpoint(Seconds(25)) // 设置检查点间隔，最佳实践为批次间隔的5～10倍

    wordCounts.print()// 打印结果

    /* 步骤3：开启计算并等待计算结束 */
    ssc.start()
    ssc.awaitTermination()
  }

  def main(args: Array[String]): Unit = {
    run(args)
  }
}
```

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 按批词频统计
  - 全局词频统计
  - 窗口操作
  - 异常检测

窗口操作

- 每隔5秒统计前10秒内输入的文本中每个单词出现的次数

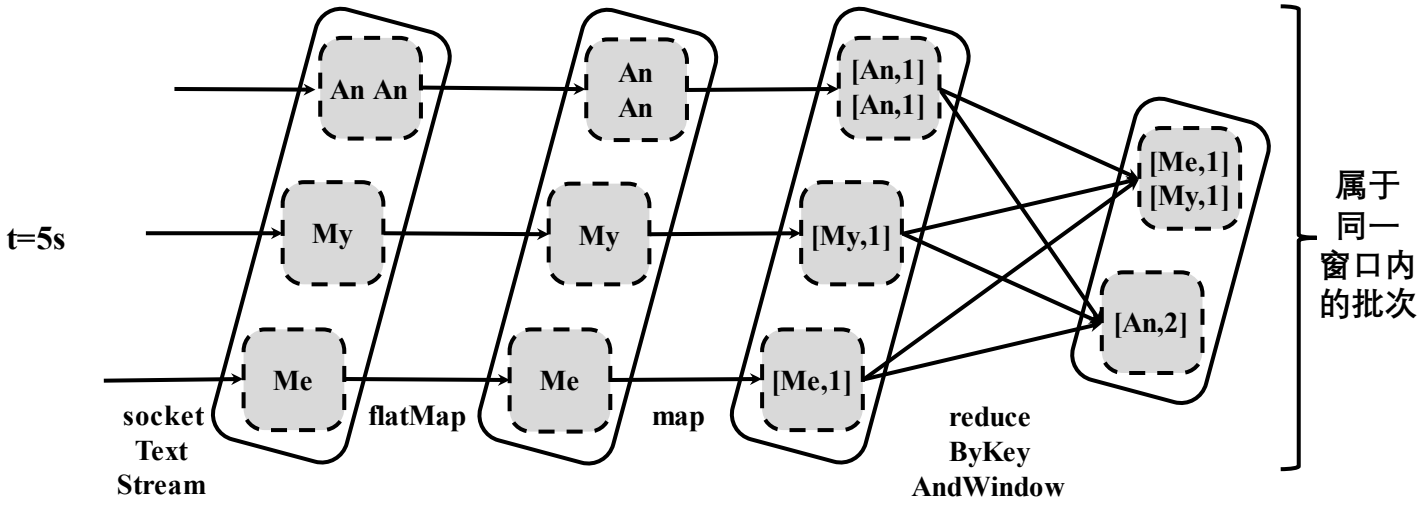
时间	输入
1s	An An
2s	My
4s	Me
6s	An An
8s	My
9s	He
11s	My My
12s	An
14s	My
…	…

时间	输出
5s	An 2 My 1 Me 1
10s	An 4 My 2 Me 1 He 1
15s	An 3 My 4 He 1
…	…

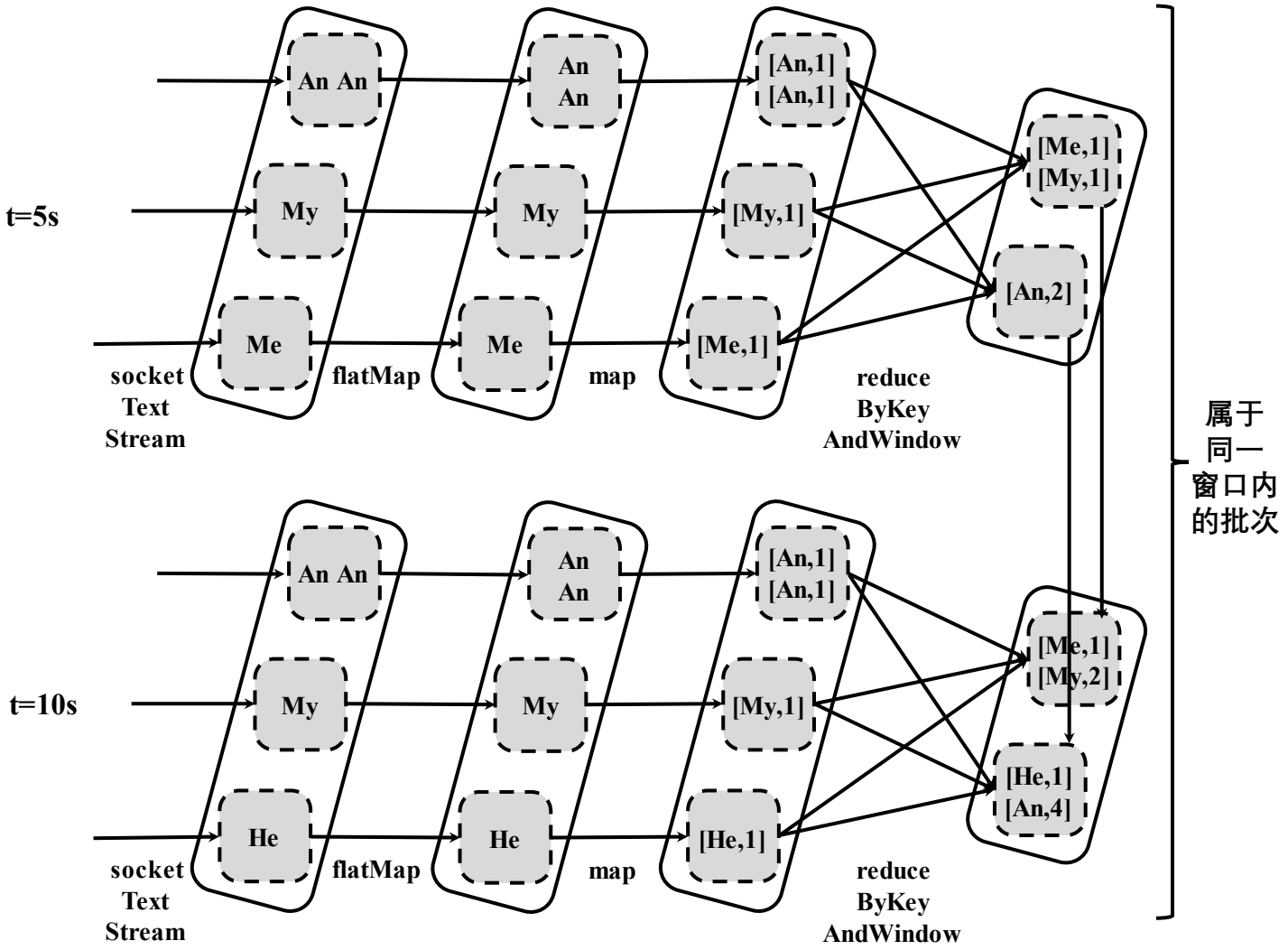
解决方案

- 对于跨若干个批次上数据的计算，可以结合 reduceByKeyAndWindow 窗口操作来完成

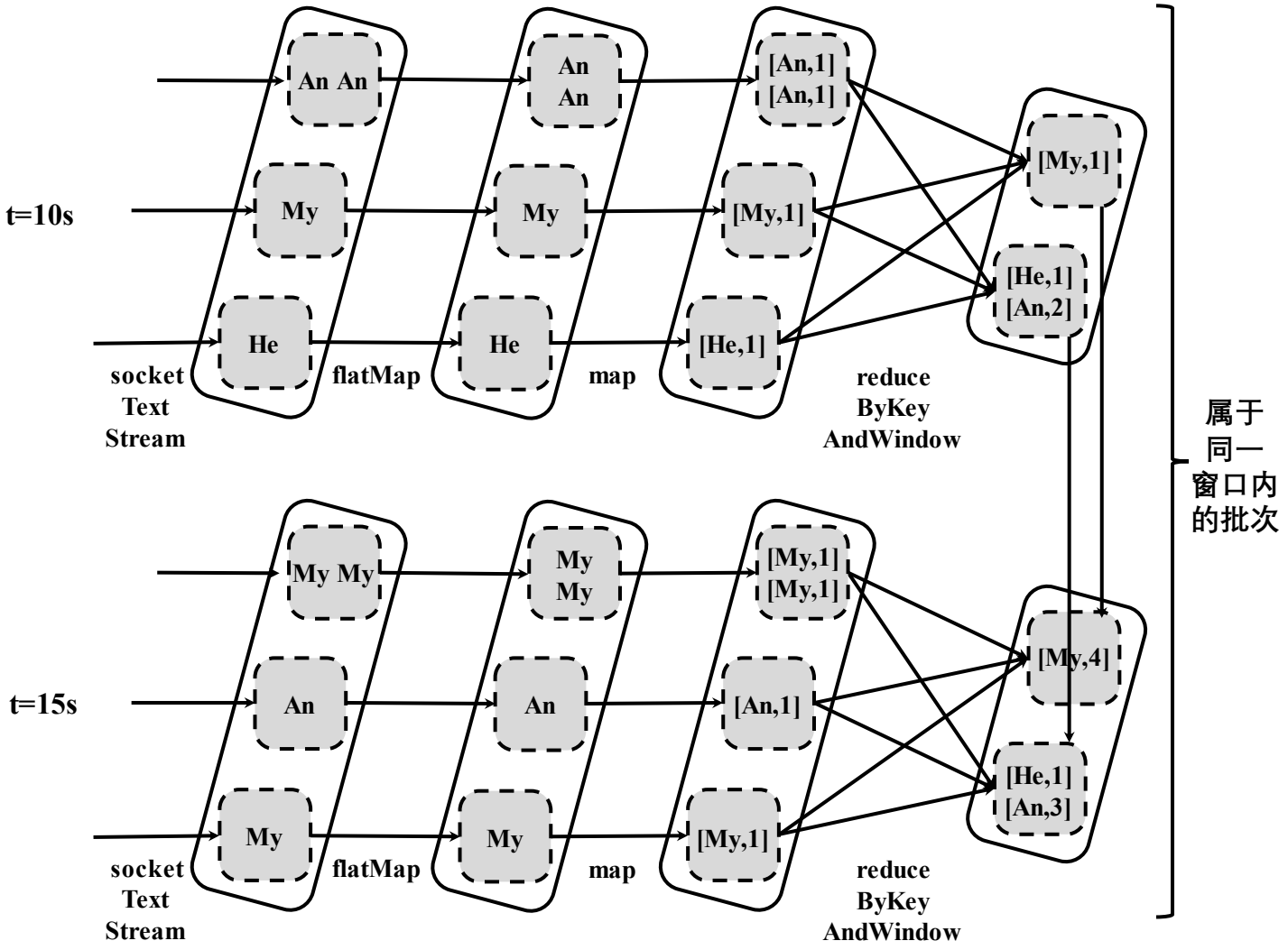
运行过程


t=5s
An An
My
Me
An
An
My
Me
[An,1]
[An,1]
[My,1]
[Me,1]
[Me,1]
[My,1]
[An,2]
socket
Text
Stream
flatMap
map
reduce
ByKey
AndWindow
属于
同一
窗口内
的批次

运行过程


t=5s
t=10s
An An
My
Me
He
An
An
My
Me
He
[An,1]
[An,1]
[My,1]
[Me,1]
[He,1]
[Me,1]
[My,1]
[An,2]
[Me,1]
[My,2]
[He,1]
[An,4]
socket
Text
Stream
flatMap
map
reduce
ByKey
AndWindow
属于
同一
窗口内
的批次

运行过程


t=10s
t=15s
socket Text Stream
flatMap
map
reduce ByKey AndWindow
An An
My
He
My My
An
[An,1] [An,1]
[My,1]
[He,1]
[My,1] [My,1]
[An,1]
[My,1]
[My,1]
[He,1] [An,2]
[My,4]
[He,1] [An,3]
属于同一窗口内的批次

代码


```scala
object Window {
  def run(args: Array[String]): Unit = {
    /* 步骤1：通过SparkConf设置配置信息，并创建StreamingContext*/
    val conf = new SparkConf
      .setAppName("Window")
      .setMaster("local[*]") // 仅用于本地进行调试，如在集群中运行则删除该行
    val ssc = new StreamingContext(conf, Seconds(5))
    // 如需使用增量式窗口操作则必须设置检查点路径
    // ssc.checkpoint("hdfs://...")

    /* 步骤2：按应用逻辑使用操作算子编写DAG，包括DStream的输入、转换和输出等 */
    val inputDStream = ssc.socketTextStream("hostname", port)
    val pairsDStream = inputDStream.flatMap(_.split(" ")).map(x => (x, 1))

    // 窗口操作，窗口长度是10秒，滑动间隔是5秒
    val wordCounts = pairsDStream.reduceByKeyAndWindow((a: Int, b: Int) => (a+b),
        Seconds(10), Seconds(5))
    // 如需使用增量式窗口操作则将第21行替换为第23行
    // val wordCounts = pairsDStream.reduceByKeyAndWindow(_+_, _-_, Seconds(10),
        Seconds(5)).checkpoint(Seconds(25))

    // 打印结果
    wordCounts.print()

    /* 步骤3：开启计算并等待计算结束 */
    ssc.start()
    ssc.awaitTermination()
  }

  def main(args: Array[String]): Unit = {
    run(args)
  }
}
```

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 按批词频统计
  - 全局词频统计
  - 窗口操作
  - 异常检测

异常检测

- 根据给定的线性模型和阈值过滤异常值，输入记录是一个向量，输出记录是异常的向量

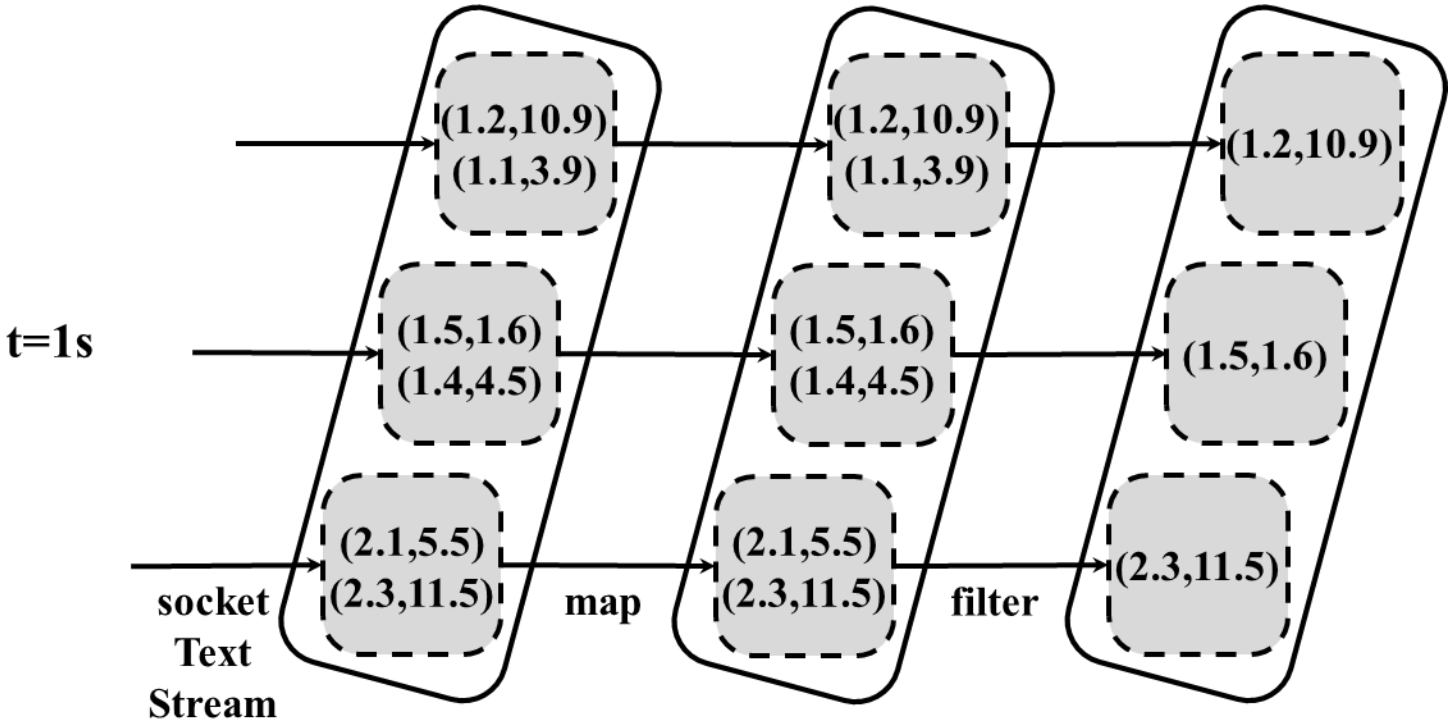
输入	输出
1.2 10.9 1.1 3.9 1.5 1.6 1.4 4.5 2.1 5.5 2.3 11.5 …	1.2 10.9 1.5 1.6 2.3 11.5 …

解决方案

- 对于接收到的输入记录，使用map方法按逗号分隔符进行解析，并转换成Double数据类型，得到(x’,y’)向量

- 在filter方法中对(x’,y’)向量应用线性模型进行检测，过滤出异常值后使用print方法输出异常值

运行过程


t=1s
(1.2,10.9)
(1.1,3.9)
(1.5,1.6)
(1.4,4.5)
(2.1,5.5)
(2.3,11.5)
socket
Text
Stream
map
(1.2,10.9)
(1.1,3.9)
(1.5,1.6)
(1.4,4.5)
(2.1,5.5)
(2.3,11.5)
filter
(1.2,10.9)
(1.5,1.6)
(2.3,11.5)

代码


```scala
object AnomalyDetection {
  def run(args: Array[String]): Unit = {
    /* 步骤1：通过SparkConf设置配置信息，并创建StreamingContext*/
    val conf = new SparkConf
      .setAppName("AnomalyDetection")
      .setMaster("local[*]") // 仅用于本地进行调试，如在集群中运行则删除该行
    val ssc = new StreamingContext(conf, Seconds(1))

    /* 步骤2：按应用逻辑使用操作算子编写DAG，包括DStream的输入、转换和输出等 */
    // 模型参数
    val w = 1.5
    val b = 2.5
    val delta = 0.5

    val inputDStream = ssc.socketTextStream("hostname", port)

    // 按逗号分割解析每行数据，并转化为Double类型
    val anomaly = inputDStream
      .map(line => {
        val tokens = line.split(",")
        (tokens(0).toDouble, tokens(1).toDouble)
      })
      // 使用线性模型检测异常
      .filter(t => {
        Math.abs(w * t._1 + b - t._2) > delta
      })

    // 输出异常
    anomaly.print()
    /* 步骤3：开启计算并等待计算结束 */
    ssc.start()
    ssc.awaitTermination()
  }

  def main(args: Array[String]): Unit = {
    run(args)
  }
}
```

课后阅读

- 论文
  - Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., & Stoica, I. (2013). Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In SOSP (pp. 423–438).

本章小结

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例