Spark高级编程（II）

摘要

- Spark SQL
- Spark Streaming

摘要

- Spark SQL
- Spark Streaming

Spark SQL

Spark SQL is Apache Spark's module for working with structured data.

Integrated

Seamlessly mix SQL queries with Spark programs.

Spark SQL lets you query structured data inside Spark programs, using either SQL or a familiar DataFrame API. Usable in Java, Scala, Python and R.

```
results = spark.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Apply functions to results of SQL queries.

Uniform Data Access

Connect to any data source the same way.

DataFrames and SQL provide a common way to access a variety of data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC. You can even join data across these sources.

```
spark.read.json("s3n://...")
  .registerTempTable("json")
results = spark.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

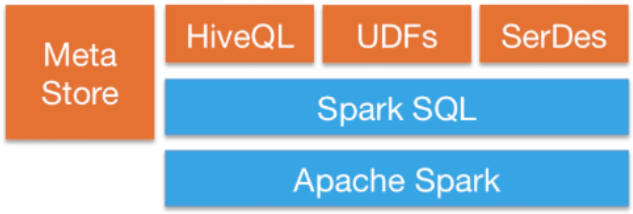
Query and join different data sources.

Spark SQL

Hive Integration

Run SQL or HiveQL queries on existing warehouses.

Spark SQL supports the HiveQL syntax as well as Hive SerDes and UDFs, allowing you to access existing Hive warehouses.

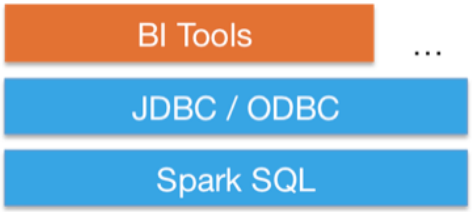


Spark SQL can use existing Hive metastores, SerDes, and UDFs.

Standard Connectivity

Connect through JDBC or ODBC.

A server mode provides industry standard JDBC and ODBC connectivity for business intelligence tools.

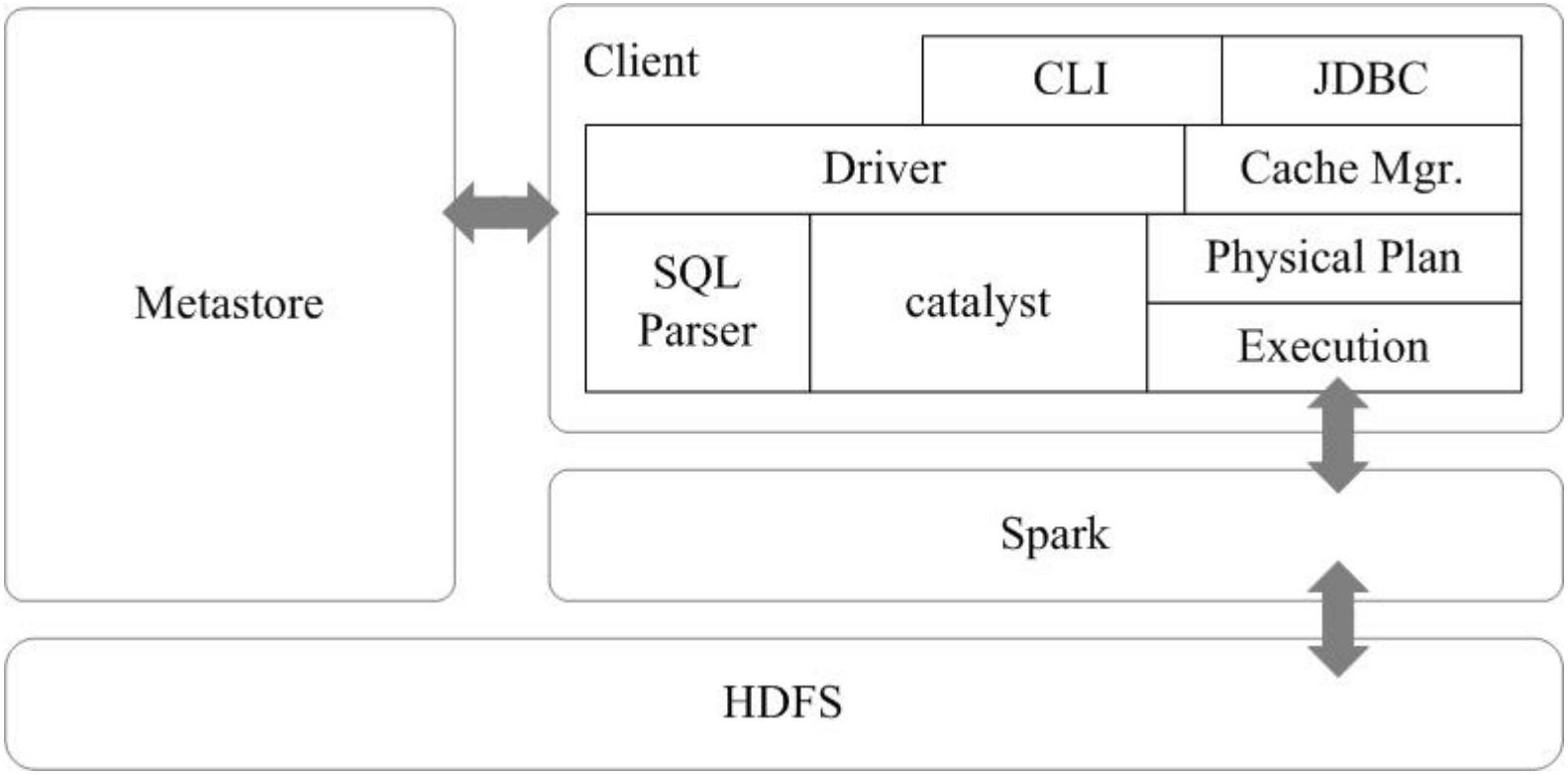


Use your existing BI tools to query big data.

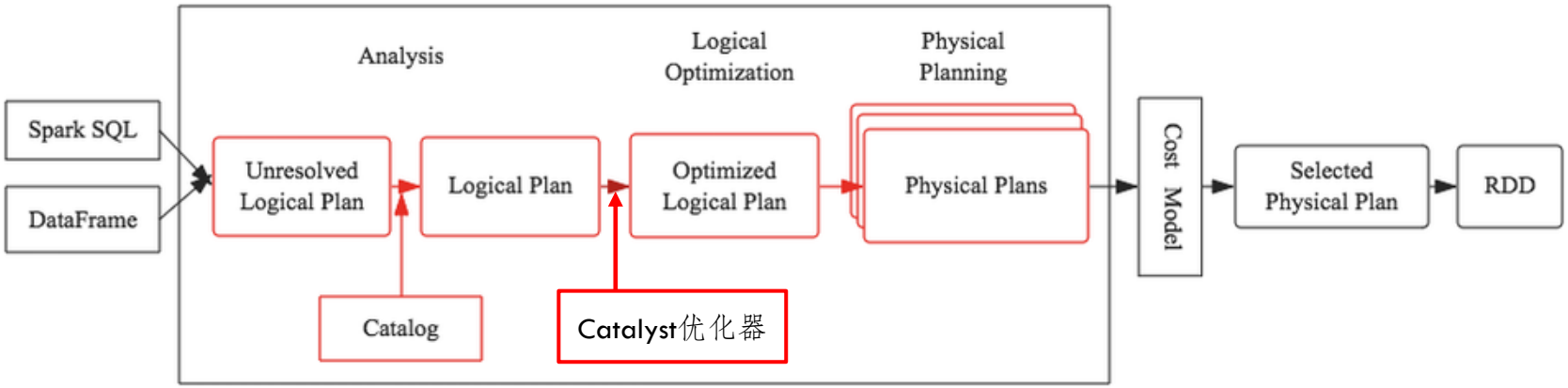
Spark SQL

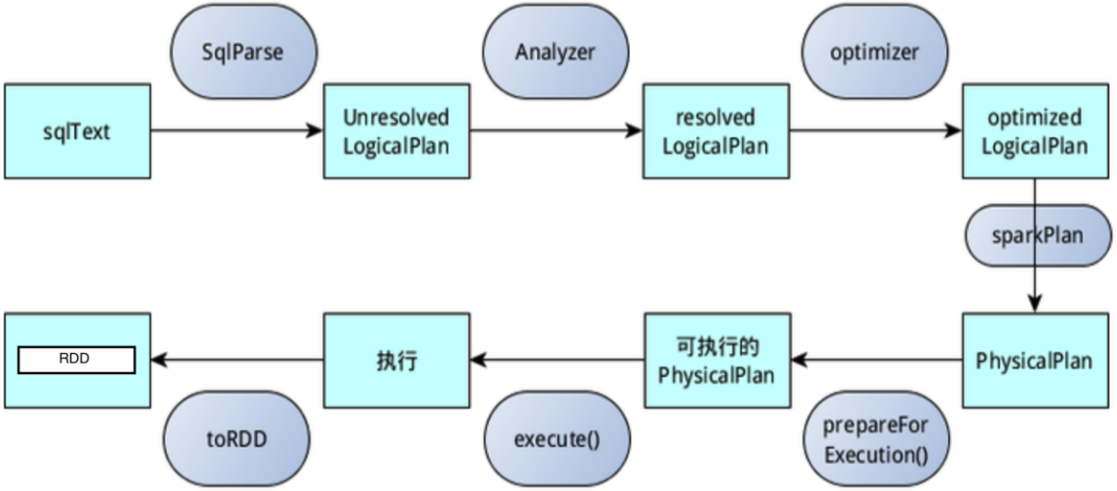
- Spark SQL：用来操作结构化和半结构化数据
  - 可以从各种结构化数据源中（例如JSON、Hive、Parquet等）读取数据；
  - 不仅支持在Spark程序内使用SQL语句进行数据查询，也支持从外部工具中通过JDBC/ODBC连接Spark SQL进行查询；
  - 支持SQL与常规的Python/Java/Scala代码高度整合，包括连接RDD与SQL表、公开的自定义SQL函数接口等。
- SchemaRDD → DataFrame/Dataset

Spark SQL架构

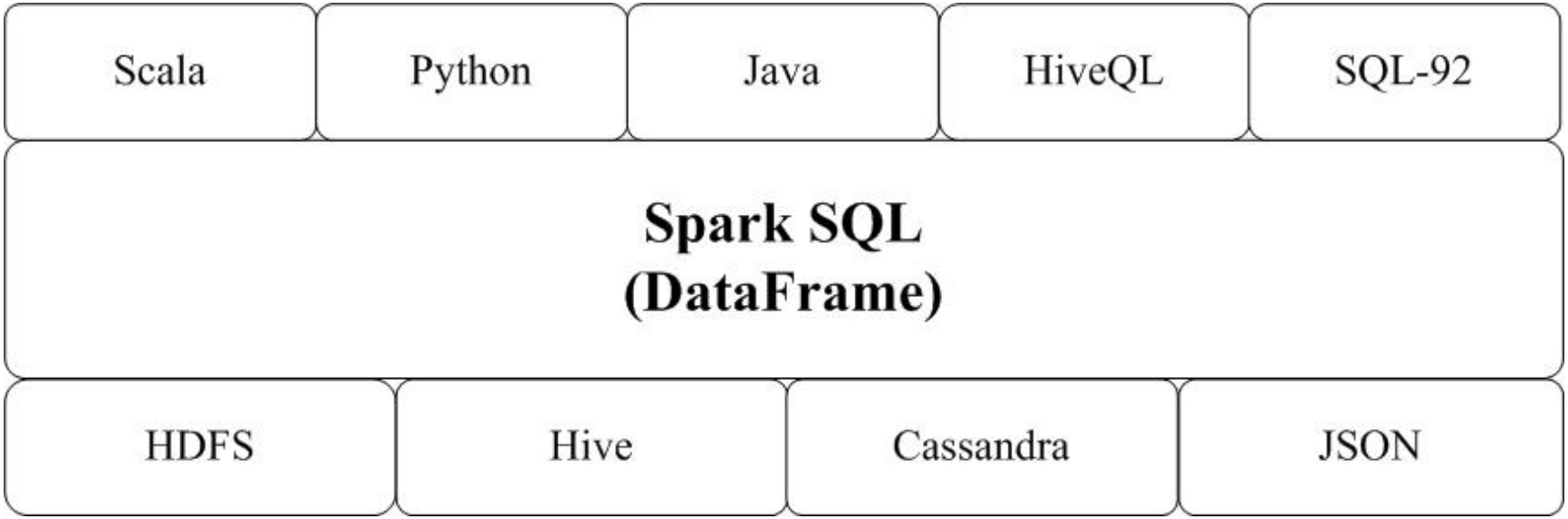

Metastore
Client
CLI
JDBC
Driver
Cache Mgr.
SQL Parser
catalyst
Physical Plan
Execution
Spark
HDFS

Spark SQL执行流程





Spark SQL支持的数据格式和编程语言



Spark SQL特点

- 数据兼容：兼容Hive，还可以从RDD、Parquet文件、JSON文件中获取数据，可以在Scala代码里访问Hive元数据，执行Hive语句，并且把结果取回作为RDD使用。支持Parquet文件读写。
- 组件扩展：语法解析器、分析器、优化器
- 性能优化：内存列存储、动态字节码生成、内存缓存数据
- 支持多种语言：Scala、Java、Python、R，还可以在Scala代码里写SQL，支持简单的SQL语法检查，能把RDD转化为DataFrame存储起来。

RDD

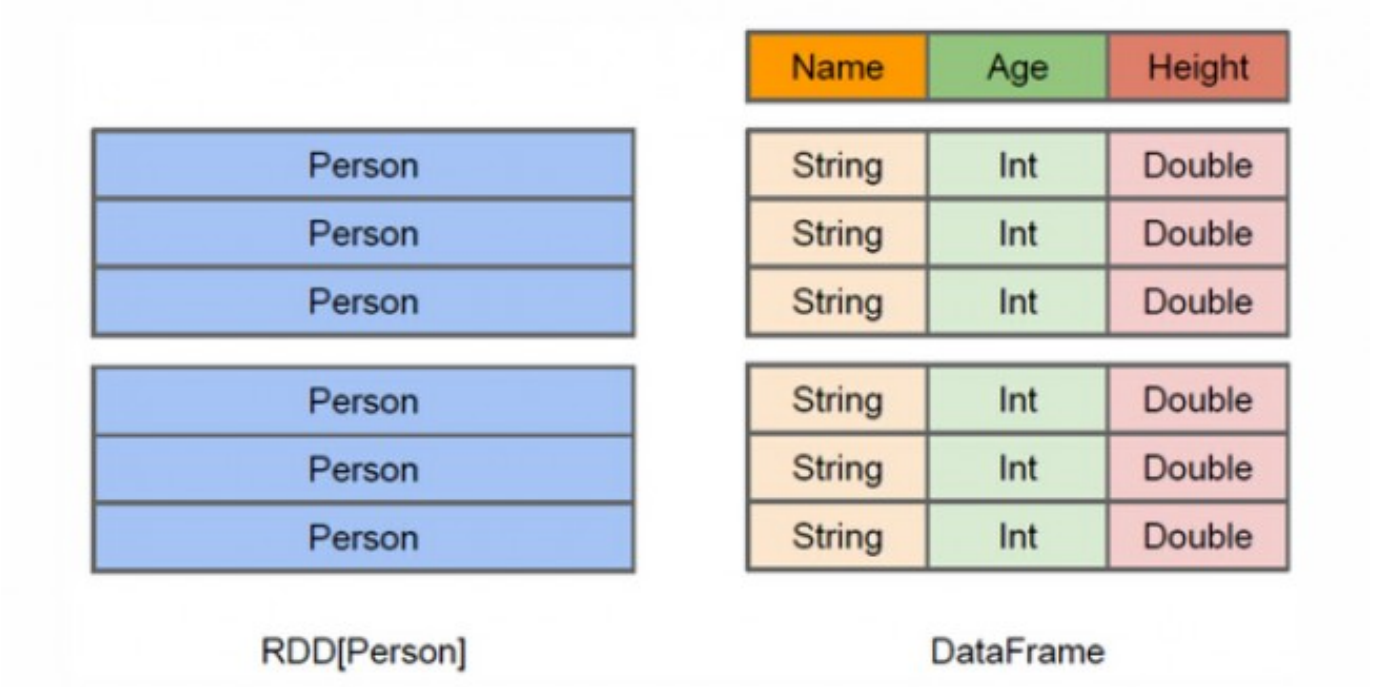
- 弹性
  - 数据可完全放内存或完全放磁盘，也可部分存放在内存，部分存放在磁盘，并可以自动切换
  - RDD出错后可自动重新计算（通过血缘自动容错）
  - 可checkpoint（设置检查点，用于容错），可persist或cache（缓存）
  - 里面的数据是分片的（也叫分区，partition），分片的大小可自由设置和细粒度调整
- 分布式
- 数据集

DataFrame

- DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.
- The DataFrame API is available in Scala, Java, Python, and R. In Scala and Java, a DataFrame is represented by a Dataset of Rows. In the Scala API, DataFrame is simply a type alias of Dataset[Row]. While, in Java API, users need to use Dataset<Row> to represent a DataFrame.

DataFrame vs. RDD

- DataFrame的推出，让Spark具备了处理大规模结构化数据的能力，不仅比原有的RDD转化方式更加简单易用，而且获得了更高的计算性能。Spark能够轻松实现从MySQL到DataFrame的转化，并且支持SQL查询。

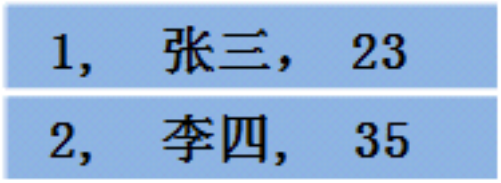


Dataset

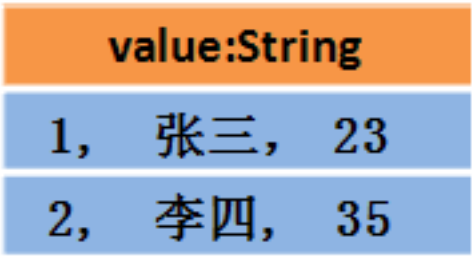
- A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine. A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). The Dataset API is available in Scala and Java.

Dataset vs. RDD

- 相对于RDD，Dataset提供了强类型支持，也是在RDD的每行数据加了类型约束。



RDD



Dataset

Dataset：每行数据是一个Object

DataFrame vs. Dataset

- 相比DataFrame，Dataset提供了编译时类型检查
- RDD转换DataFrame后不可逆，但RDD转换Dataset是可逆的。
- Dataset包含了DataFrame的功能，Spark2.0中两者统一，DataFrame表示为DataSet[Row]，即Dataset的子集。
- 使用API尽量使用Dataset，不行再选用DataFrame，其次选择RDD。

DataFrame vs. Dataset

- 编译时类型检查

```
val df1 = spark.read.json("/tmp/people.json")
//json文件中没有score字段，但是能编译通过
val df2 = df1.filter("score > 60")
df2.show()
```

```
val ds1 = spark.read.json("/tmp/people.json").as[People]

//使用dataSet这样写，在IDE中就能发现错误
val ds2 = ds1.filter(_.score < 60)
val ds3 = ds1.filter(_.age < 18)
ds3.show()
```

DataFrame vs. Dataset

- 可逆 vs. 不可逆

```
scala> case class People(id: Long, name: String)
defined class People

scala> val peopleRDD = sc.makeRDD(Seq(People(1,"zhangsan"),People(2,"lisi")))
peopleRDD: org.apache.spark.rdd.RDD[People] = ParallelCollectionRDD[0] at makeRDD at <console>:26
```

```
scala> val peopleDf = peopleRDD.toDF
peopleDf: org.apache.spark.sql.DataFrame = [id: bigint, name: string]

scala> peopleDf.rdd
res0: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[4] at rdd at <console>:31
```

```
scala> val peopleDs = peopleRDD.toDS
peopleDs: org.apache.spark.sql.Dataset[People] = [id: bigint, name: string]

scala> peopleDs.rdd
res1: org.apache.spark.rdd.RDD[People] = MapPartitionsRDD[6] at rdd at <console>:31
```

DataFrame

- DataFrame初始化

```scala
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

```scala
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

DataFrame

- Untyped Dataset Operations

```
// This import is needed to use the $-notation
import spark.implicits._
// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+
```

DataFrame

- 常用的DataFrame操作
  - df.printSchema()
  - df.select(df("name"),df("age")+1).show()
  - df.filter(df("age") > 20).show()
  - df.groupBy("age").count().show()
  - df.sort(df("age").desc).show()
  - df.sort(df("age").desc, df("name").asc).show()
  - df.select(df("name").as("username"),df("age")).show()

DataFrame

- 运行SQL

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

DataFrame

- 全局临时视图

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Dataset

- Dataset初始化

```scala
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
// +----+---+

// Encoders for most common types are automatically provided by importing spark.implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)

// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name
val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

RDD ←→ DataFrame

- 利用反射推断模式

```
scala> import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder

scala> import org.apache.spark.sql.Encoder
import org.apache.spark.sql.Encoder

scala> import spark.implicits._  //导入包，支持把一个RDD隐式转换为一个DataFrame
import spark.implicits._

scala> case class Person(name: String, age: Long) //定义一个case class
defined class Person

scala> val peopleDF = spark.sparkContext.textFile("file:///usr/local/spark/examples/src
/main/resources/people.txt").map(_.split(",")).map(attributes => Person(attributes(0),
attributes(1).trim.toInt).toDF()
peopleDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]

scala> peopleDF.createOrReplaceTempView("people") //必须注册为临时表才能供下面的查询使用

scala> val personsRDD = spark.sql("select name,age from people where age > 20")
//最终生成一个DataFrame
personsRDD: org.apache.spark.sql.DataFrame = [name: string, age: bigint]
scala> personsRDD.map(t => "Name:"+t(0)+","+"Age:"+t(1)).show()  //DataFrame中的每个元素
都是一行记录，包含name和age两个字段，分别用t(0)和t(1)来获取值
```

RDD ←→ DataFrame

- 编程指定模式

```
1.  scala> import org.apache.spark.sql.types._
2.  import org.apache.spark.sql.types._
3.
4.  scala> import org.apache.spark.sql.Row
5.  import org.apache.spark.sql.Row
6.
7.  //生成 RDD
8.  scala> val peopleRDD = spark.sparkContext.textFile("file:///usr/local/spark/examples/sr
    c/main/resources/people.txt")
9.  peopleRDD: org.apache.spark.rdd.RDD[String] = file:///usr/local/spark/examples/src/main
    /resources/people.txt MapPartitionsRDD[1] at textFile at <console>:26
10.
11. //定义一个模式字符串
12. scala> val schemaString = "name age"
13. schemaString: String = name age
14.
15. //根据模式字符串生成模式
16. scala> val fields = schemaString.split(" ").map(fieldName => StructField(fieldName, Str
    ingType, nullable = true))
17. fields: Array[org.apache.spark.sql.types.StructField] = Array(StructField(name,StringTy
    pe,true), StructField(age,StringType,true))
18.
19. scala> val schema = StructType(fields)
20. schema: org.apache.spark.sql.types.StructType = StructType(StructField(name,StringType,
    true), StructField(age,StringType,true))
21. //从上面信息可以看出，schema描述了模式信息，模式中包含name和age两个字段
```

RDD ←→ DataFrame

- 编程指定模式

```
23. //对peopleRDD 这个RDD中的每一行元素都进行解析val peopleDF = spark.read.format("json").load
    ("examples/src/main/resources/people.json")
24.
25.
26. scala> val rowRDD = peopleRDD.map(_.split(",")).map(attributes => Row(attributes(0),
    attributes(1).trim))
27. rowRDD: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[3] at map
    at <console>:29
28.
29. scala> val peopleDF = spark.createDataFrame(rowRDD, schema)
30. peopleDF: org.apache.spark.sql.DataFrame = [name: string, age: string]
31.
32. //必须注册为临时表才能供下面查询使用
33. scala> peopleDF.createOrReplaceTempView("people")
34.
35. scala> val results = spark.sql("SELECT name,age FROM people")
36. results: org.apache.spark.sql.DataFrame = [name: string, age: string]
37.
38. scala> results.map(attributes => "name: " + attributes(0)+","+"age:"+attributes(1)).sho
    w()
```

Spark SQL数据源

- DataFrame提供统一接口加载和保存数据源中的数据，包括：结构化数据、Parquet文件(默认)、JSON文件、Hive表，以及通过JDBC连接外部数据源。

加载


```scala
val usersDF = spark.read.load("examples/src/main/resources/users.parquet")
usersDF.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

```scala
val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

```scala
val peopleDFCsv = spark.read.format("csv")
  .option("sep", ";")
  .option("inferSchema", "true")
  .option("header", "true")
  .load("examples/src/main/resources/people.csv")
```

```scala
usersDF.write.format("orc")
  .option("orc.bloom.filter.columns", "favorite_color")
  .option("orc.dictionary.key.threshold", "1.0")
  .save("users_with_options.orc")
```

保存

- 保存模式SaveMode

Scala/Java	Any Language	Meaning
SaveMode.ErrorIfExists (default)	"error" or "errorifexists"(default)	如果保存数据已经存在，抛出异常
SaveMode.Append	"append"	如果保存数据已经存在，追加DataFrame数据
SaveMode.Overwrite	"overwrite"	如果保存数据已经存在，重写DataFrame数据
SaveMode.Ignore	"ignore"	如果保存数据已经存在，忽略DataFrame数据

Parquet

- Parquet是一种支持多种数据处理系统的存储格式，Spark SQL提供了读写Parquet文件，并且自动保存原始数据的模式，优点：
  - 高效，Parquet采取列式存储避免读入不需要的数据
  - 方便的压缩和解压缩
  - 可以直接固化为Parquet文件，也可以直接读取Parquet文件，具有比磁盘更好的缓存效果

JSON

- Spark SQL可以自动推断出一个JSON数据集的Schema并作为一个DataFrame加载，通过SQLContext.read.json()方法使用JSON文件创建DataFrame，或者通过转换一个JSON对象的RDD[String]创建DataFrame。

Hive

- 若要把Spark SQL连接到一个部署好的Hive上，必须把hive-site.xml复制到Spark的配置文件目录中(conf/)。
- 如果没有部署好Hive，Spark SQL会在当前的工作目录中创建出自己的Hive元数据仓库，叫做metastore_db。
- 配置项 spark.sql.warehouse.dir，默认的数据仓库地址。

Hive

- Spark SQL支持任何Hive支持的数据格式

```scala
import java.io.File

import org.apache.spark.sql.{Row, SaveMode, SparkSession}

case class Record(key: Int, value: String)

// warehouseLocation points to the default location for managed databases and tables
val warehouseLocation = new File("spark-warehouse").getAbsolutePath

val spark = SparkSession
  .builder()
  .appName("Spark Hive Example")
  .config("spark.sql.warehouse.dir", warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()

import spark.implicits._
import spark.sql

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL
sql("SELECT * FROM src").show()
```

连接数据库

- JDBC/ODBC服务器作为一个独立的Spark驱动程序运行，可以在多用户之间共享。任意一个客户端都可以在内存中缓存数据表，对表进行查询。集群的资源以及缓存数据都在所有用户之间共享。
  - 启动Thriftserver
    - >sbin/start-thriftserver.sh –master sparkMaster
  - 连接JDBC服务器
    - >bin/beeline –u jdbc:hive2://localhost:10000

性能调优

- 缓存数据

属性名称	默认值	含义
spark.sql.inMemoryColumnarStorage.compressed	true	当设置为true，Spark SQL将基于数据统计为每列自动选择压缩编码
spark.sql.inMemoryColumnarStorage.batchSize	10000	控制列式缓存的批处理尺寸，大批量可以提升内存的使用率和压缩率，但是缓存数据时会有内存溢出的风险

- 调优参数
  - spark.sql.autoBroadcastJoinThreshold; spark.sql.tungsten.enabled; spark.sql.shuffle.partitions; spark.sql.planner.externalSort…
- 增加并行度

数据类型

- org.apache.spark.sql.types
- 数值类型
  - 字节，短整型，整型，长整型，浮点型，双精度型，数值型
- 字符串类型
- 二进制类型
- 布尔类型
- 时间类型
  - 时间戳类型，日期类型
- 复杂类型
  - 数组类型，Map类型，StructType，StructField

摘要

- Spark SQL
- Spark Streaming

Spark Streaming

Spark Streaming makes it easy to build scalable fault-tolerant streaming applications.

Ease of Use

Build applications through high-level operators.

Spark Streaming brings Apache Spark's language-integrated API to stream processing, letting you write streaming jobs the same way you write batch jobs. It supports Java, Scala and Python.

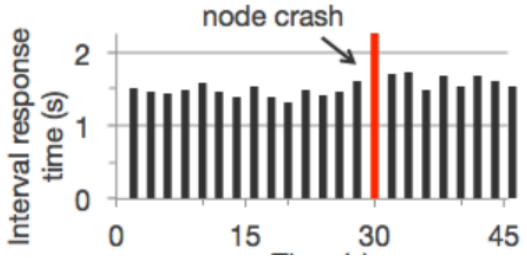
```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

Counting tweets on a sliding window

Fault Tolerance

Stateful exactly-once semantics out of the box.

Spark Streaming recovers both lost work and operator state (e.g. sliding windows) out of the box, without any extra code on your part.



Spark Integration

Combine streaming with batch and interactive queries.

By running on Spark, Spark Streaming lets you reuse the same code for batch processing, join streams against historical data, or run ad-hoc queries on stream state. Build powerful interactive applications, not just analytics.

```
stream.join(historicCounts).filter {
  case (word, (curCount, oldCount)) =>
    curCount > oldCount
}
```

Find words with higher frequency than historic data

架构


Kafka
Flume
HDFS
ZeroMQ
Twitter
Spark Streaming
HDFS
Databases
Dashboards

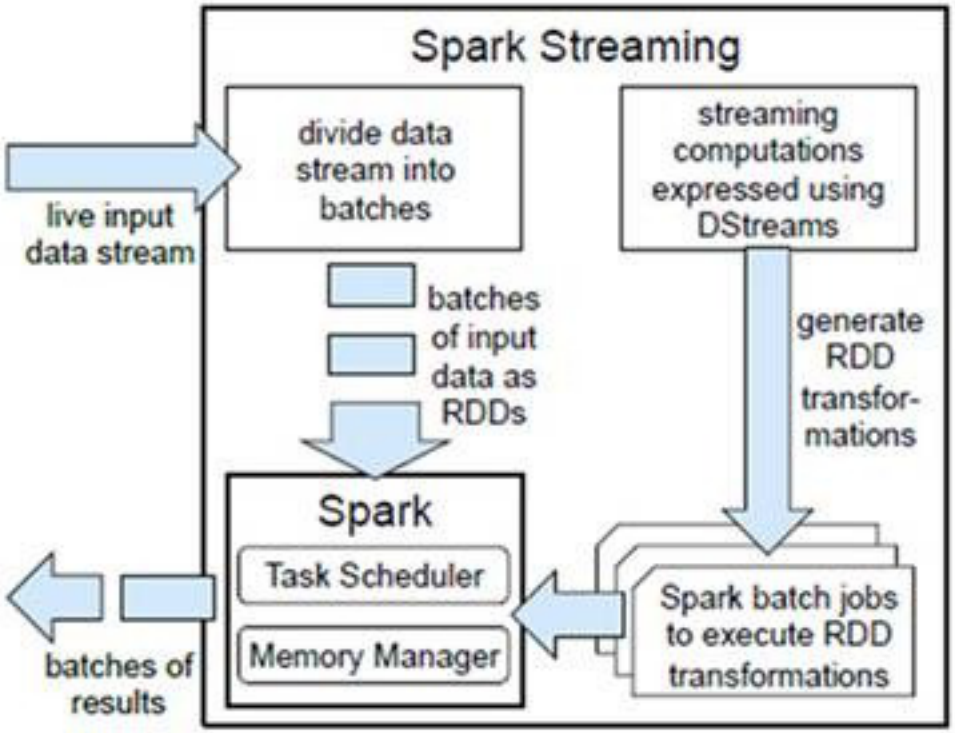

input data stream
Spark Streaming
batches of input data
Spark Engine
batches of processed data

工作原理

- Spark Streaming将流式计算分解成一系列短小的批处理作业，具有如下特性：
  - 能线性扩展至超过数百个节点；
  - 实现亚秒级延迟处理；
  - 可与Spark批处理和交互式处理无缝集成；
  - 提供了一个简单的API实现复杂的算法；
  - 更多的网络流方式支持，包括Kafka、Flume、Kinesis、Twitter、ZeroMQ等。

DStream抽象

- DStream (Discretized Stream, 离散流)：连续的数据流，由一系列RDDs组成。



DStream抽象

- DStream的核心思想是将计算作为一系列较小时间间隔的、状态无关的、确定批次的任务，每个时间间隔内接收到的输入数据被可靠地存储在集群中，作为它的一个输入数据集。当某个时间间隔完成，将对相应的数据集并行地进行Map、Reduce和groupBy等操作，产生中间数据或输出新的数据集，并存储在RDD中。任务间的状态可以通过RDD重新计算，得益于计算任务被分解成一系列的小任务，用户可以在合适的粒度上呈现任务间的依赖关系。

DStream抽象

- 两类操作：
  - 转化操作：生成一个新的DStream
  - 输出操作：把数据写入外部系统中
- 增加了与时间相关的新操作，比如滑动窗口

简单的例子

- 例：从监听TCP套接字的数据服务器获取文本数据，然后计算文本中包含的单词数。

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
// Create a local StreamingContext with two working thread and batch interval of 1 sec
ond
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

```
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

简单的例子

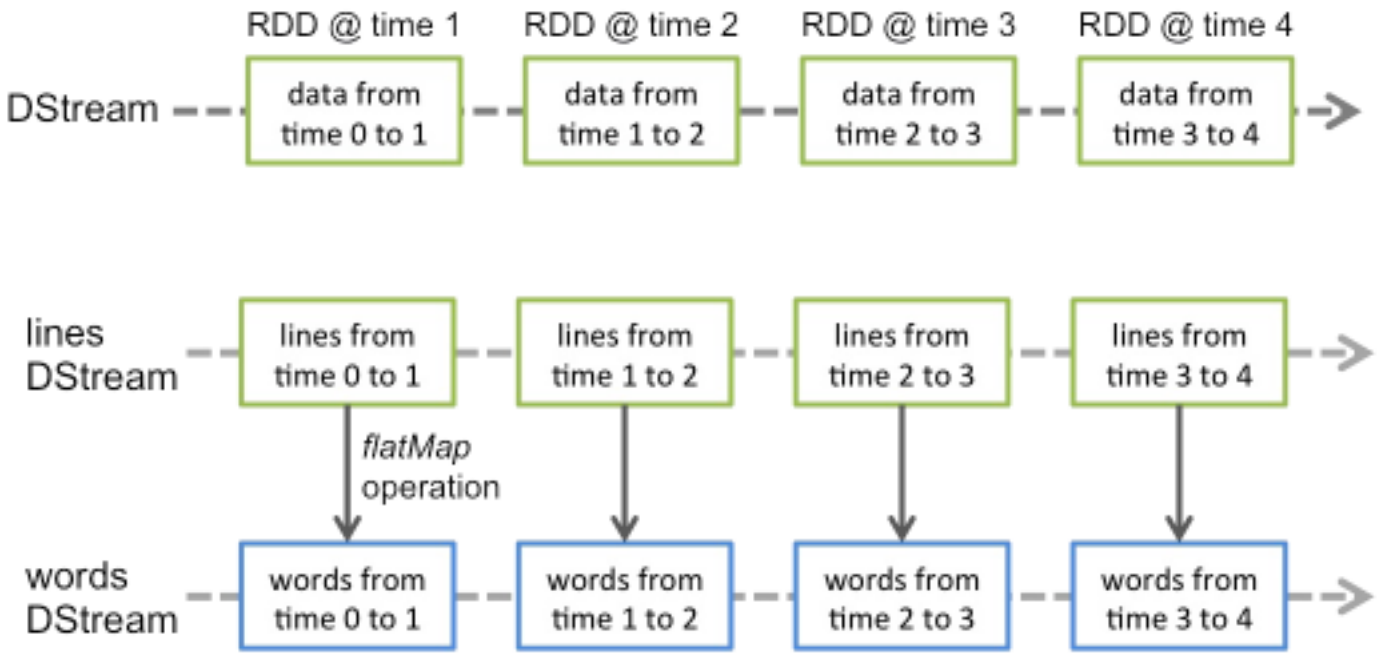
- 例：从监听TCP套接字的数据服务器获取文本数据，然后计算文本中包含的单词数。

```
import org.apache.spark.streaming.StreamingContext._
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print()
```

```
ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

抽象

- DStream抽象



输入源

- 每一个输入流DStream和一个Receiver对象关联，这个Receiver从源中获取数据，并将数据存入内存中进行处理。
- 输入源
  - 基本源：在StreamingContext API中直接使用，例如文件系统、套接字连接、Akka的actor等。
  - 高级源：包括Kafka，Flume，Twitter等，需要通过额外的类来使用。
- 多个数据流→多个Receiver，要分配足够的核（如果是本地运行，那么是线程）用以处理接收的数据并且运行receiver是非常重要的

DStream操作

- 转换操作：允许在DStream运行任何RDD-to-RDD函数，比如map, flatMap, filter, reduce, join等
- 状态操作
  - updateStateByKey：不断用新信息更新它的同时保持任意状态
  - 窗口操作：允许在一个滑动窗口数据上应用transformation算子，需制定两个参数：窗口长度（窗口的持续时间）和滑动的时间间隔（窗口操作执行的时间间隔）

```
// Reduce last 30 seconds of data, every 10 seconds
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b), Seconds(
30), Seconds(10))
```

DStream操作

Transformation	Meaning
map	对传入的每个元素，返回一个新的元素
flatMap	对传入的每个元素，返回一个或多个元素
filter	对传入的元素返回true或false，返回的false的元素被过滤掉
union	将两个DStream进行合并
count	返回元素的个数
reduce	对所有values进行聚合
countByValue	对元素按照值进行分组，对每个组进行计数，最后返回<K, V>的格式
reduceByKey	对key对应的values进行聚合
cogroup	对两个DStream进行连接操作，一个key连接起来的两个RDD的数据，都会以Iterable<V>的形式，出现在一个Tuple中。

DStream操作

Transformation	Meaning
join	对两个DStream进行join操作，每个连接起来的pair，作为新DStream的RDD的一个元素
transform	对数据进行转换操作
updateStateByKey	为每个key维护一份state，并进行更新（这个，我认为，是在普通的实时计算中，最有用的一种操作）
window	对滑动窗口数据执行操作（实时计算中最有特色的一种操作）

DStream操作

- 输出操作

Output Operation	Meaning
print()	在DStream的每个批数据中打印前10条元素，这个操作在开发和调试中都非常有用。在Python API中调用 `pprint()` 。
saveAsObjectFiles(prefix, [suffix])	保存DStream的内容为一个序列化的文件 `SequenceFile` 。每一个批间隔的文件的文件名基于 `prefix` 和 `suffix` 生成。"prefix-TIME_IN_MS[.suffix]"，在Python API中不可用。
saveAsTextFiles(prefix, [suffix])	保存DStream的内容为一个文本文件。每一个批间隔的文件的文件名基于 `prefix` 和 `suffix` 生成。"prefix-TIME_IN_MS[.suffix]"
saveAsHadoopFiles(prefix, [suffix])	保存DStream的内容为一个hadoop文件。每一个批间隔的文件的文件名基于 `prefix` 和 `suffix` 生成。"prefix-TIME_IN_MS[.suffix]"，在Python API中不可用。
foreachRDD(func)	在从流中生成的每个RDD上应用函数 `func` 的最通用的输出操作。这个函数应该推送每个RDD的数据到外部系统，例如保存RDD到文件或者通过网络写到数据库中。需要注意的是， `func` 函数在驱动程序中执行，并且通常都有RDD action在里面推动RDD流的计算。

DStream操作

- 缓存及持久化
  - persist()
  - Dstream的持久化策略是将数据序列化在内存中。
  - 基于窗口或状态的操作，如reduceByWindow、reduceByKeyAndWindow和updateStateByKey，Dstream都会自动持久化在内存中，无须显式调用persist()方法。
  - 通过网络接收的流数据默认采取保存两份序列化后的数据在两个不同的节点上的持久化策略，从而实现容错。

Checkpointing机制

- Metadata checkpointing：保存流计算的定义信息到容错存储系统如HDFS中。这用来恢复应用程序中运行worker的节点的故障。元数据包括：
  - Configuration
  - DStream operations
  - Incomplete batches
- Data checkpointing：保存生成的RDD到可靠的存储系统中，这在有状态transformation（如结合跨多个批次的数据）中是必须的。有状态的transformation的中间RDD会定时存储到可靠存储系统中。

Checkpointing配置

- 在容错、可靠的文件系统（HDFS、S3等）中设置一个目录用于保存checkpoint信息。
  - streamingContext.checkpoint(checkpointDirectory)

```
// Function to create and setup a new StreamingContext
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
val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContex
t _)

// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted
context. ...

// Start the context
context.start()
context.awaitTermination()
```

Spark Streaming编程

- 首先创建StreamingContext：

```
#方法一：
val conf = new SparkConf().setAppName(appName).setMaster(master);
val ssc = new StreamingContext(conf, Seconds(1));
#方法二：可以使用已有的SparkContext来创建
val sc = new SparkContext(conf);
val ssc = new StreamingContext(sc, Seconds(1));
```

- 注：appName，是用来在Spark UI上显示的应用名称；master，是一个Spark、Mesos或者Yarn集群的URL，或者是local[*]；batch interval可以根据你的应用程序的延迟要求以及可用的集群资源情况来设置。

Spark Streaming编程

- 接下来的流程：
  1. 通过创建输入DStream来创建输入数据源。
  2. 通过对DStream定义transformation和output算子操作，来定义实时计算逻辑。
  3. 调用StreamingContext的start()方法，来开始实时处理数据。
  4. 调用StreamingContext的awaitTermination()方法，来等待应用程序的终止。可以使用CTRL+C手动停止，或者就是让它持续不断的运行进行计算。
  5. 也可以通过调用StreamingContext的stop()方法，来停止应用程序。

Spark Streaming编程

- 注意事项：
  1. 只要一个StreamingContext启动之后，就不能再往其中添加任何计算逻辑了。比如执行start()方法之后，还给某个DStream执行一个算子。
  2. 一个StreamingContext停止之后，是肯定不能够重启的，调用stop()之后，不能再调用start()。
  3. 一个JVM同时只能有一个StreamingContext启动，在你的应用程序中，不能创建两个StreamingContext。
  4. 调用stop()方法时，会同时停止内部的SparkContext，如果不希望如此，还希望后面继续使用SparkContext创建其他类型的Context，比如SQLContext，那么就用stop(false)。
  5. 一个SparkContext可以创建多个StreamingContext，只要上一个先用stop(false)停止，再创建下一个即可。

输入源

- 套接字

```
val lines = ssc.socketTextStream("localhost", 9999)
```

- 文件流

```
val logData = ssc.textFileStream(logDirectory)
```

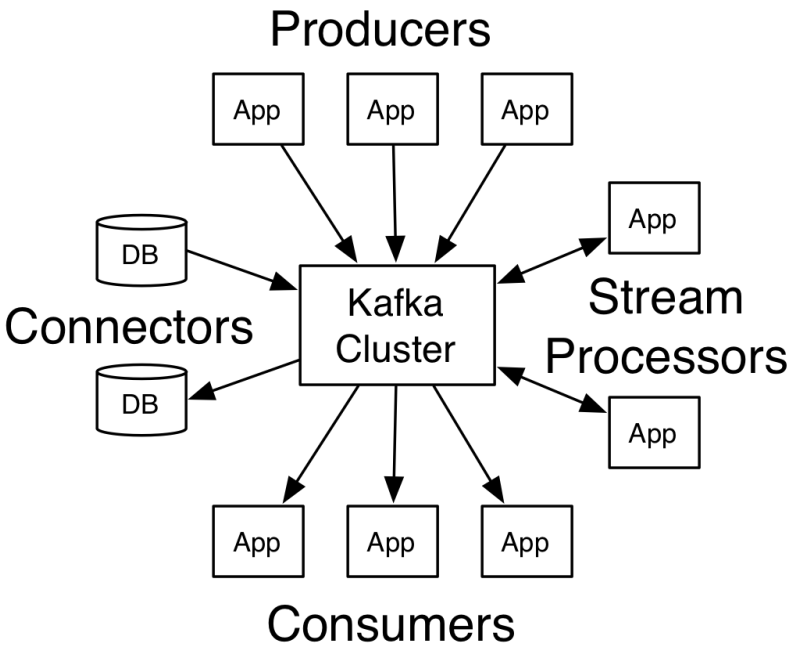
```
val data = ssc.fileStream[KeyClass, ValueClass, InputFormatClass]
(dataDirectory)
```

- 附加数据源
  - Apache Kafka
  - Twitter
  - Amazon Kinesis
  - Apache Flume

Apache Kafka

- Apache Kafka是一个分布式流处理平台。用于构建实时的数据管道和流式的app。它可以水平扩展，高可用，速度快，并且已经运行在数千家公司的生产环境。
- 流处理平台的三种特性
  - 可以让你发布和订阅流式的记录。这一方面与消息队列或者企业消息系统类似。
  - 可以储存流式的记录，并且有较好的容错性。
  - 可以在流式记录产生时就进行处理。

Apache Kafka



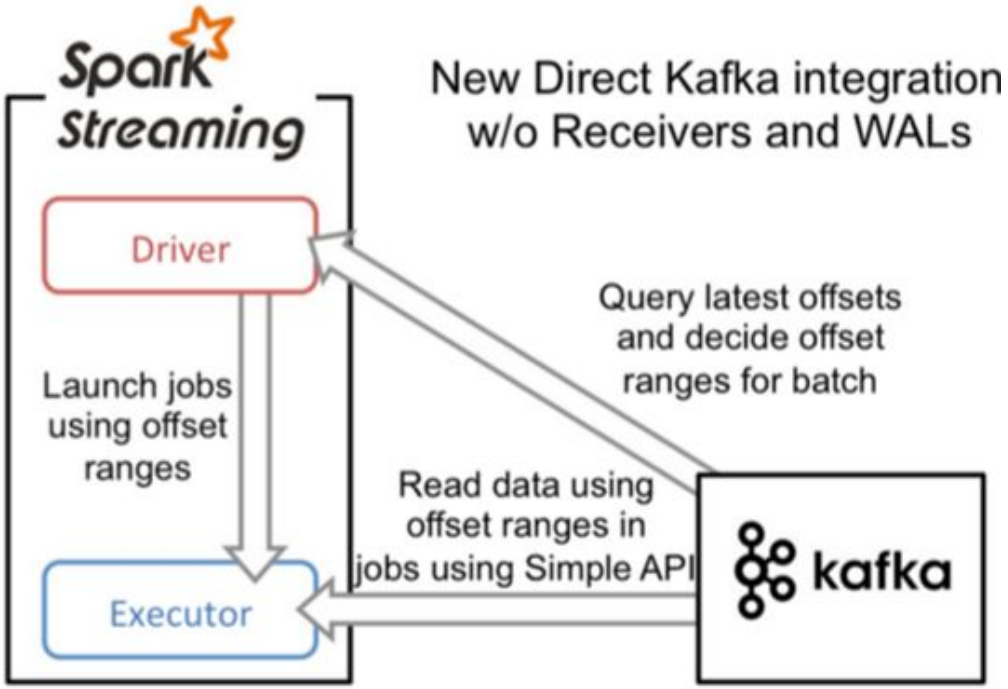
Apache Kafka

- Kafka维护按类区分的消息，称为主题（topic）
- 生产者（producer）向Kafka的主题发布消息
- 消费者（consumer）向主题注册，并且接收发布到这些主题的消息
- Kafka以一个拥有一台或多台服务器的集群运行着，每一台服务器称为broker
- 从高层来说，生产者（producer）通过网络发消息到Kafka集群，而Kafka集群则以下面这种方式对消费者进行服务。

Apache Kafka

- 步骤
  1. 运行zookeeper服务器
  2. 运行kafka服务器
  3. 创建topic
  4. 查看topic是否存在
  5. 创建producer
  6. 创建consumer（测试）
  7. 提交Spark Streaming作业

Spark Streaming + Kafka



Spark Streaming + Kafka

- Linking
  - groupId = org.apache.spark
  - artifactId = spark-streaming-kafka-0-10_2.12
  - version = 3.0.1
- 编程
  - 通过KafkaUtils对象创建出Dstream；
  - 由于KafkaUtils可以订阅多个主题，因此它创建出的Dstream由成对的主题和消息组成；
  - 要创建一个流数据，需要使用StreamingContext实例，一个由逗号隔开的Zookeeper主机列表字符串、消费者组的名字（唯一名字），以及一个从主题到针对这个主题的接收器线程数的映射表来调用createStream()方法。

Spark Streaming + Kafka


```
…
Map<String, Integer> topicMap = new HashMap<>();
String[] topics = args[2].split(",");
for (String topic: topics) {  topicMap.put(topic, numThreads);  }

JavaPairReceiverInputDStream<String, String> messages =
        KafkaUtils.createStream(jssc, args[0], args[1], topicMap);

JavaDStream<String> lines = messages.map(Tuple2::_2);
JavaDStream<String> words = lines.flatMap(x ->  Arrays.asList
(SPACE.split(x)).iterator());
JavaPairDStream<String, Integer> wordCounts = words.mapToPair(s -> new
Tuple2<>(s, 1)).reduceByKey((i1, i2) -> i1 + i2);
```

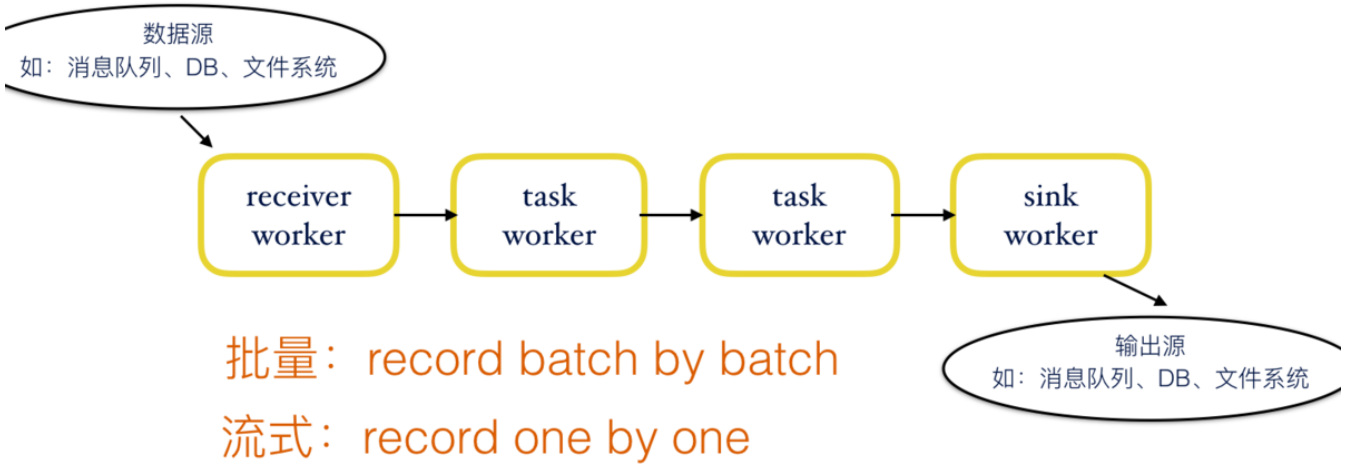
Spark Streaming + Kafka

- Usage: JavaKafkaWordCount <zkQuorum> <group> <topics> <numThreads>
  - <zkQuorum> is a list of one or more zookeeper servers that make quorum
  - <group> is the name of kafka consumer group
  - <topics> is a list of one or more kafka topics to consume from
  - <numThreads> is the number of threads the kafka consumer should use

- $ bin/run-example org.apache.spark.examples.streaming.JavaKafkaWordCount zoo01,zoo02, zoo03 my-consumer-group topic1,topic2 1

批量计算 vs 流式计算

- 批量和流式处理数据粒度不一样，批量每次处理一定大小的数据块（输入一般采用文件系统），一个task处理完一个数据块之后，才将处理好的中间数据发送给下游。流式计算则是以record为单位，task在处理完一条记录之后，立马发送给下游。



批量计算 vs 流式计算

- 区别
  - 数据处理单位
    - 批量计算按数据块来处理数据，每一个task接收一定大小的数据块
    - 流式计算的上游算子处理完一条数据后，会立马发送给下游算子，所以一条数据从进入流式系统到输出结果的时间间隔较短
  - 数据源
    - 批量计算通常处理的是有限数据（bound data），数据源一般采用文件系统，而流式计算通常处理无限数据（unbound data），一般采用消息队列作为数据源。
  - 任务类型
    - 批量计算中的每个任务都是短任务，任务在处理完其负责的数据后关闭，而流式计算往往是长任务，每个work一直运行，持续接受数据源传过来的数据。

批量计算 vs 流式计算

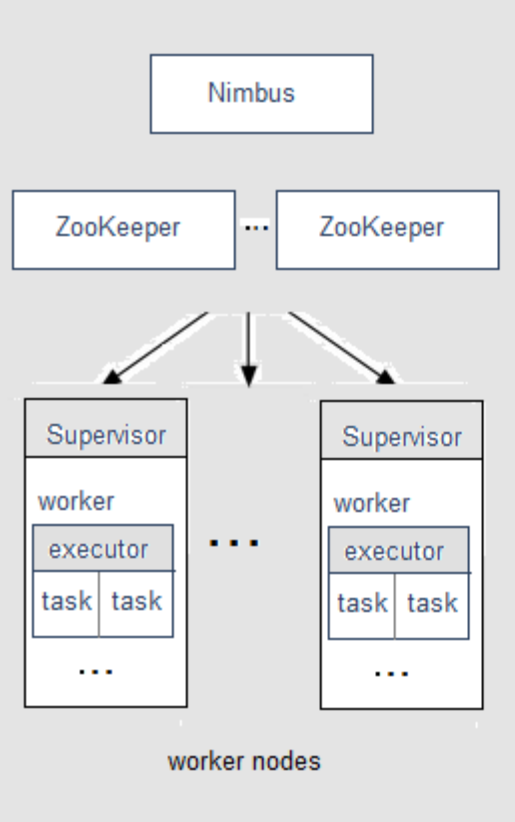
- 离线=批量？实时=流式？
  - **离线和实时应该指的是：数据处理的延迟；批量和流式指的是：数据处理的方式。**两者并没有必然的关系。事实上Spark streaming就是采用小批量（batch）的方式来实现实时计算。

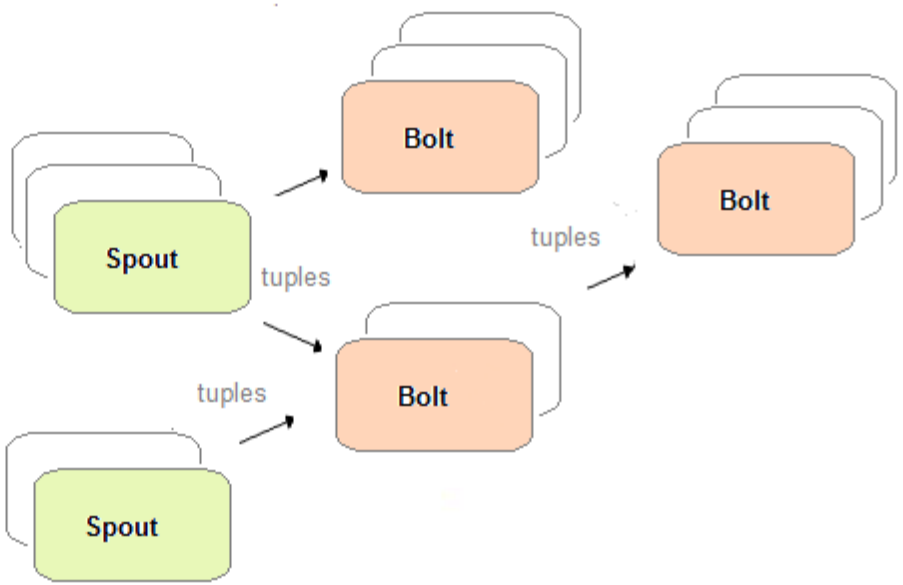
流式计算框架

- Apache Storm
- Apache Spark Streaming
- Apache Flink

流式计算框架

- Apache Storm

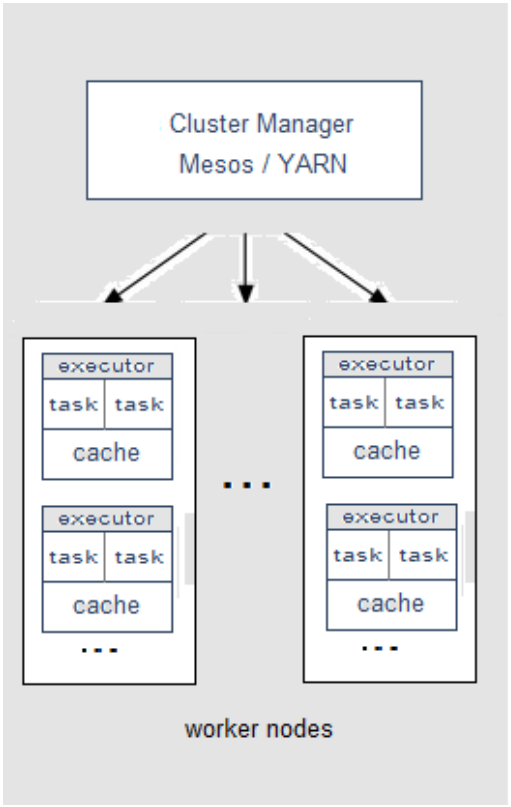


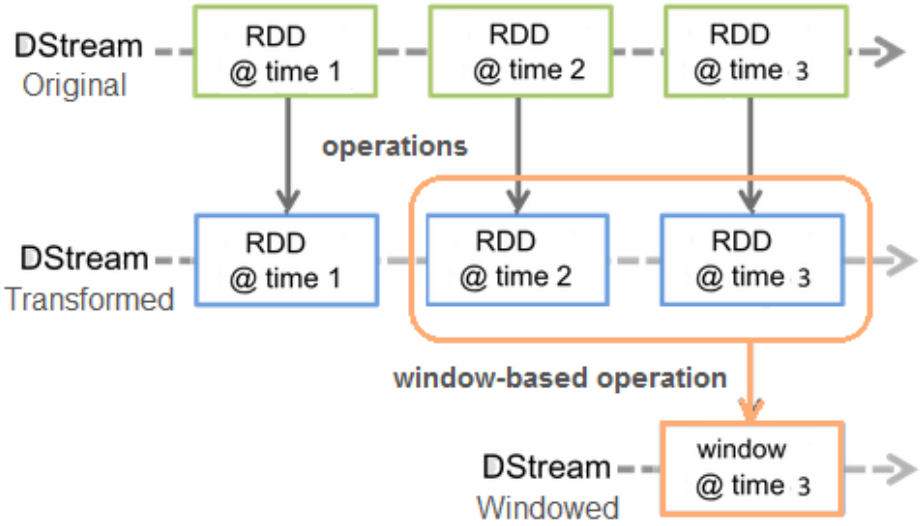


APACHE STORM™

流式计算框架

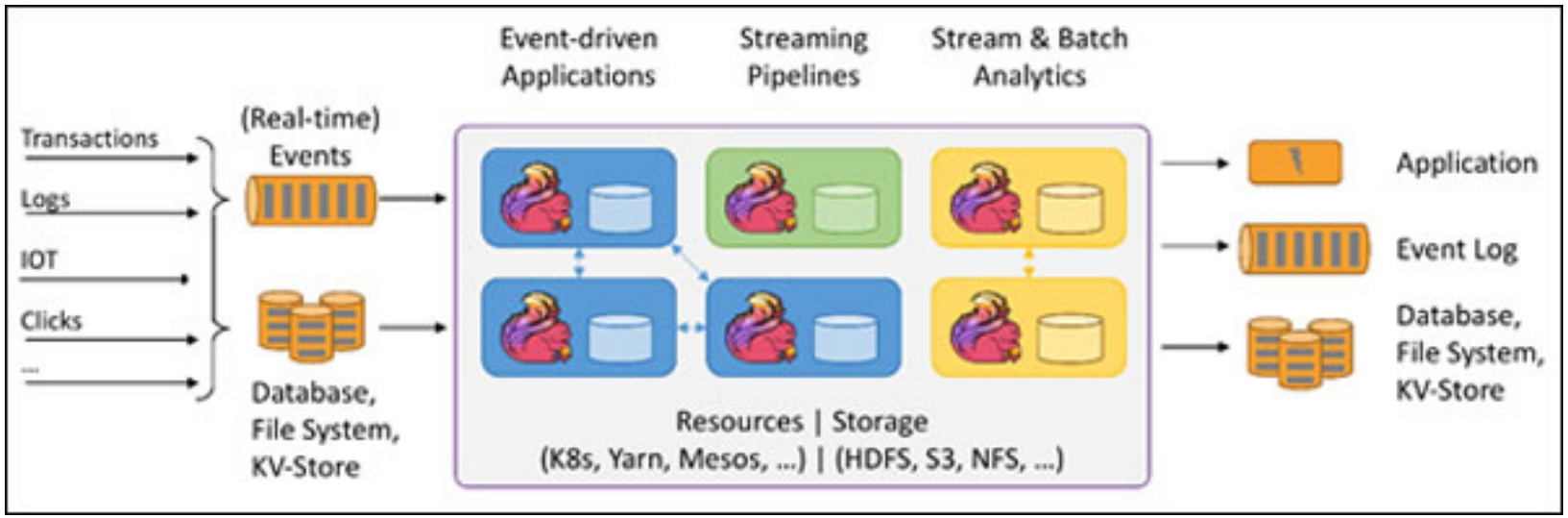
- Apache Spark Streaming





流式计算框架

- Apache Flink



Comparison

	Strom	Spark Streaming	Flink
Streaming Model	Native	Micro-batching	Native
Guarantees	At-Least-Once	Exactly-Once	Exactly-Once
Back Pressure	No	Yes	Yes
Latency	Very Low	Medium	Low
Throughput	Low	High	High
Fault Tolerance	Record ACKs	RDD Based Check Pointing	Check Pointing
Stateful	No	Yes (DStream)	Yes (Operators)