Apache Spark研究报告

1. Spark简介

Apache Spark是一个围绕速度、易用性和复杂分析构建的大数据处理框架。最初在2009年由加州大学伯克利分校的AMPLab开发，并于2010年成为Apache的开源项目之一。

Spark扩展了广泛使用的MapReduce计算模型，而且高效地支持更多计算模式，包括交互式查询和流处理。在处理大规模数据集时，速度是非常重要的。速度快就意味着可以进行交互式的数据操作，否则每次操作就需要等待数分钟甚至数小时。Spark的一个主要特点就是能够在内存中进行计算，因而比MapReduce更快。不过即使是必须在磁盘上进行的复杂计算，Spark依然比MapReduce更加高效。

Spark适用于各种各样原先需要多种不同的分布式平台的场景，包括批处理、迭代算法、交互式查询、流处理。通过在一个统一的框架下支持这些不同的计算，Spark使我们可以简单且低耗地把各种处理流程整合在一起。这样的组合，在实际的数据分析过程中很有意义。不仅如此，Spark的这种特性还大大减轻了原先需要对各种平台分别管理的负担。

Spark提供的接口非常丰富。除了提供基于Scala、Java、Python和SQL的简单易用的API以及内建丰富的程序库外，Spark还能和其他大数据工作密切配合使用。例如，Spark可以运行在Hadoop集群上，访问包括Cassandra在内的任意Hadoop数据源。

2. Spark目前发展应用情况

对于一个具有相当技术门槛与复杂度的平台，Spark从诞生到正式版本的成熟，经历的时间如此之短，让人感到惊诧。2009年，Spark诞生于加州伯克利大学AMPLab，最开初属于加州伯克利大学的研究性项目。它于2010年正式开源，并于2013年成为了Aparch基金项目，并于2014年成为Aparch基金的顶级项目，整个过程不到五年时间。

从Spark的版本演化看，足以说明这个平台旺盛的生命力以及社区的活跃度。尤其在2013年来，Spark进入了一个高速发展期，代码库提交与社区活跃度都有显著增长。以活跃度论，Spark在所有Aparch基金会开源项目中，位列前三。相较于其他大数据平台或框架而言，Spark的代码库最为活跃。

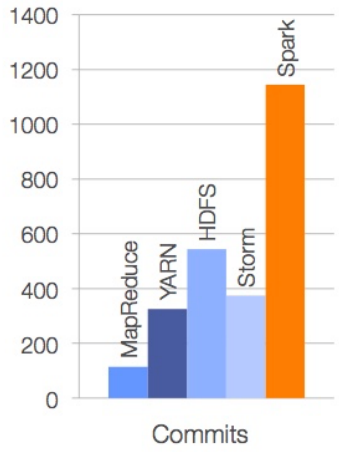

图 1 各大项目代码库提交数

目前，Spark的正式版本得到了部分Hadoop主流厂商的支持，如下企业或平台发布的Hadoop版本中，都包含了Spark：

Hortonworks IBM cloudera MAPR Pivotal

这说明业界已经认可了Spark，Spark也被许多企业尤其是互联网企业广泛应用到商业项目中。根据Spark的官方统计，目前参与Spark的贡献以及将Spark运用在商业项目的公司大约有80余家。在国内，投身Spark阵营的公司包括阿里、百度、腾讯、网易、搜狐等。在Spark Summit 大会上，参会的演讲嘉宾分享了在音乐推荐（Spotify）、实时审计的数据分析（Sharethrough）、流在高速率分析中的运用（Cassandra）、文本分析（IBM）、客户智能实时推荐（Graphflow）等诸多在应用层面的话题，这足以说明Spark的应用程度。

3. Spark体系结构

Spark项目包含多个紧密集成的组件，Spark的核心是由一个由很多计算任务组成的、运行在多个工作机器或者是一个计算集群上的应用进行调度、分发以及监控的计算引擎。由于Spark的核心引擎有着速度快和通用的的特点，因此Spark还支持为各种不同应用场景专门设计的高级组件，比如SQL和机器学习等。这些组件关系密切并且可以相互调用，这样就可以像在平常软件项目中使用程序库那样，组合使用这些的组件。

各组件间密切结合的设计原理有这样几个优点。首先，软件栈中所有的程序库和高级组件都可以从下层的改进中获益。比如，当Spark的核心引擎引入了一个优化时，SQL和机器学习程序库也都能自动获得性能提升。其次，运行整个软件栈的代价变小了。不需要运行多个独立

的软件系统了，一个机构只需要运行一套软件系统即可。这些代价包括系统的部署、维护、测试、支持等。这也意味着Spark软件栈中每添加一个新的组件，使用Spark的机构都能马上试用新加入的组件。这就把原先尝试一种新的数据分析系统所需要的下载、部署并学习一个新的软件项目的代价简化成了只需要升级Spark。

Spark主要包括Spark Core和在Spark Core基础之上建立的应用框架Spark SQL、Spark Streaming、MLlib和GraphX。

Core库中主要包括上下文（Spark Context）、抽象数据集（RDD、DataFrame和DataSet）、调度器（Scheduler）、洗牌（shuffle）和序列化器（Serializer）等。Spark系统中的计算、IO、调度和shuffle等系统基本功能都在其中。

在Core库之上就根据业务需求分为用于交互式查询的SQL、实时流处理Streaming、机器学习MIlib和图计算GraphX四大框架，除此外还有一些其他实验性项目如Tachyon、BlinkDB和Tungsten等。HDFS是Spark主要应用的持久化存储系统。

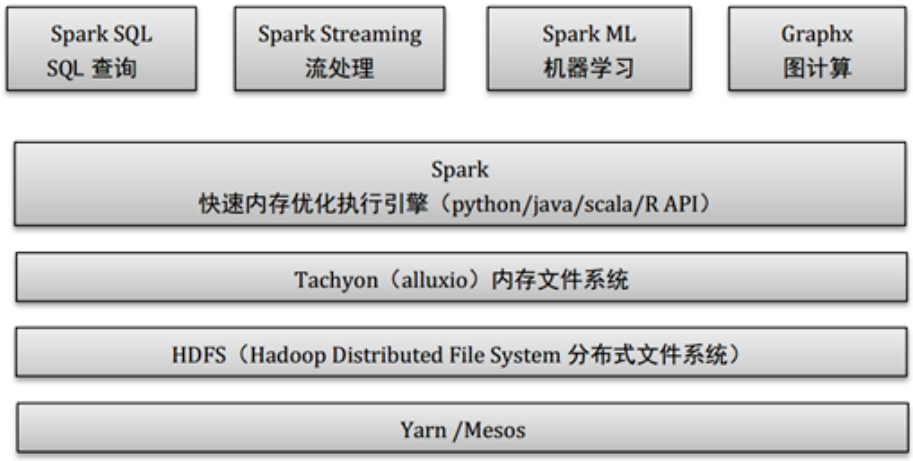

图 2 Spark体系结构

Spark Streaming:

Spark Streaming基于微批量方式的计算和处理，可以用于处理实时的流数据。它使用DStream，简单来说就是一个弹性分布式数据集（RDD）序列，处理实时数据。

Spark SQL:

Spark SQL可以通过JDBC API将Spark数据集暴露出去，而且还可以用传统的BI和可视化工具在Spark数据上执行类似SQL的查询。用户还可以用Spark SQL对不同格式的数据（如JSON，Parquet以及数据库等）执行ETL，将其转化，然后暴露给特定的查询。

Spark MLlib:

MLlib是一个可扩展的Spark机器学习库，由通用的学习算法和工具组成，包括二元分类、线性回归、聚类、协同过滤、梯度下降以及底层优化原语。

Spark GraphX:

GraphX是用于图计算和并行图计算的新的（alpha）Spark API。通过引入弹性分布式属性图（Resilient Distributed Property Graph），一种顶点和边都带有属性的有向多重图，扩展了Spark RDD。为了支持图计算，GraphX暴露了一个基础操作符集合（如subgraph，joinVertices和aggregateMessages）和一个经过优化的Pregel API变体。此外，GraphX还包括一个持续增长的用于简化图分析任务的图算法和构建器集合。

4. Spark相关概念介绍

（1）**Application**： 用户在 spark 上构建的程序，包含了 driver 程序以及在集群上运行的程序代码，物理机器上涉及了 driver，master，worker 三个节点。

（2）**SparkContext**：Spark应用程序的入口，负责调度各个运算资源，协调各个Worker Node上的Executor。

（3）**Driver Program**：运行Application的main()函数并且创建SparkContext，定义一个 spark 应用程序所需要的三大步骤的逻辑：加载数据集，处理数据，结果展示。

（4）**Executor**：是为Application运行在Worker node上的一个进程，该进程负责运行Task，并且负责将数据存在内存或者磁盘上。每个Application都会申请各自的Executor来处理任务。

（5）**Cluster Manager**：在集群上获取资源的外部服务 (例如：Standalone、Mesos、Yarn)。

（6）**Worker Node**：集群中任何可以运行Application代码的节点，运行一个或多个Executor进程。

（7）**Task**：运行在Executor上的工作单元。

（8）**Job**：包含很多 task 的并行计算，可以认为是 Spark RDD 里面的 action，每个 action 的触发会生成一个job。 用户提交的 Job 会提交给 DAGScheduler，Job 会被分解成 Stage，Stage 会被细化成 Task，Task 就是在一个数据 partition 上的单个数据处理流程。

（9）**Stage**：每个Job会被拆分很多组task，每组任务被称为Stage，也称TaskSet。

（10）**RDD**：是Resilient distributed datasets的简称，中文为弹性分布式数据集；是Spark最核心的模块和类。

（11）**DAGScheduler**：根据Job构建基于Stage的DAG，并提交Stage中的taskset给TaskScheduler。

（12）**TaskScheduler**：将task提交给Worker node集群运行并返回结果。

（13）**Transformations**：是Spark API的一种类型，Transformation返回值还是一个RDD，所有的Transformation采用的都是惰性策略，如果只是将Transformation提交是不会执行计算的。

（14）**Action**：是Spark API的一种类型，Action返回值不是一个RDD，而是一个scala集合；计算只有在Action被提交的时候计算才被触发。

5. Spark核心 — 弹性分布式数据集（RDD）

5.1. RDD是什么？

RDD，全程为Resilient Distributed Datasets，是一个容错的、并行的数据结构，可以让用户显式地将数据存储到磁盘和内存当中，并能控制数据的分区。RDD作为数据结构，本质上是一个只读的分区记录集合。可以通过两种方式来创建 RDD：一种是基于物理存储中的数据，比如说磁盘上的文件；另一种，也是大多数创建 RDD 的方式，即通过其他 RDD 转换(transformation)而成。

5.2. 窄依赖与宽依赖

一个RDD可以包含多个分区，每个分区就是一个dataset片段。RDD可以相互依赖。RDD经过转换操作后生成新的RDD，前一个RDD与新的RDD构成了谱系（lineage）关系，即两者之间存在一定的依赖关系。如果RDD的每个分区最多只能被一个Child RDD的一个分区使用，则称之为窄依赖(narrow dependency)；若多个Child RDD分区都可以依赖，则称之为宽依赖(wide dependency)。不同的操作依据其特性，可能会产生不同的依赖。例如map操作会产生narrow dependency，而join操作则产生wide dependency。计算窄依赖的子RDD：可以在某一个计算节点上直接通过父RDD的某几块数据（通常是一块）计算得到子RDD某一块的数据；计算宽依赖的子RDD：子RDD某一块数据的计算必须等到它的父RDD所有数据都计算完成之后才可以进行，而且需要对父RDD的计算结果进行hash并传递到对应的节点之上。

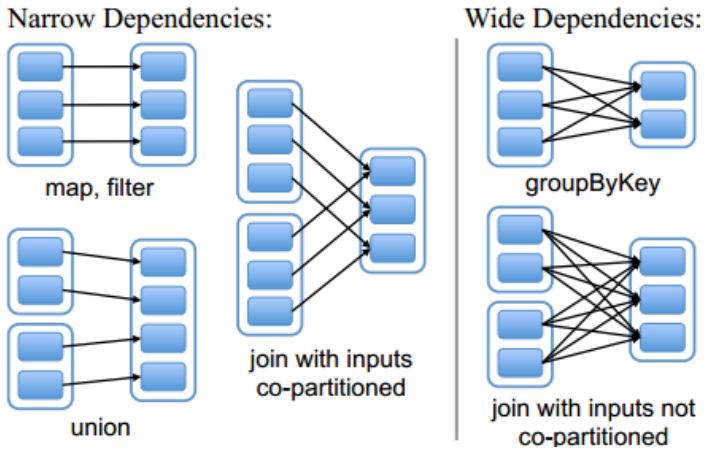

图 3 窄依赖与宽依赖

5.3. RDD支持的操作

RDD支持两种操作：转换操作（transformation）从现有的数据集创建一个新的数据集；而行动操作（actions）在数据集上运行计算后，返回一个值给驱动程序。 例如，map就是一

种转换操作，它将数据集每一个元素都传递给函数，并返回一个新的分布数据集表示结果。另一方面，reduce是一种行动操作，通过一些函数将所有的元素叠加起来，并将最终结果返回给Driver程序。

Spark中的所有转换操作都是惰性计算的，并不会直接计算结果。相反的，它们只是记住应用到基础数据集（例如一个文件）上的这些转换动作。只有当发生一个要求返回结果给Driver的行动操作时，这些转换才会真正运行。这个设计让Spark更加有效率的运行。例如，我们可以实现：通过map创建的一个新数据集，并在reduce中使用，最终只返回reduce的结果给driver，而不是整个大的新数据集。

默认情况下，每一个转换过的RDD都会在执行一个行动操作时被重新计算。不过，可以使用persist(或者cache)方法，持久化一个RDD在内存中。在这种情况下，Spark将会在集群中，保存相关元素，下次执行行动操作时，可以重用这个RDD，更快速访问。在磁盘上持久化数据集，或在集群间复制数据集也是支持的。

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

图 4 RDD支持的转换和行动操作

5.4. RDD对容错的支持

支持容错通常采用两种方式：数据复制或日志记录。对于以数据为中心的系统而言，这两种方式都非常昂贵，因为它需要跨集群网络拷贝大量数据。

RDD是天生支持容错的。第一，RDD是一个不可改变(immutable)的数据集，其次，它可利用谱系图来记录不同RDD之间的依赖关系，当执行任务的worker失败时，可以通过谱系图重新执行之前的操作，恢复丢失的数据。不同的依赖关系恢复的方式不同：

窄依赖：当某数据分片丢失时，只有丢失的那一块数据的父RDD需要被重新计算；

宽依赖：当某数据分片丢失时，需要把父RDD的所有分区数据重新计算一次，计算量明显比窄依赖情况下大很多。所以在长谱系链，特别是有宽依赖的时候，需要在适当的时机设置数据检查点。

6. Spark运行原理

6.1. Spark分布式部署

分布式集群上Spark应用程序的一般执行框架主要由SparkContext（spark上下文）、Cluster Manager（资源管理器）和Executor（单个节点的执行进程）。其中Cluster Manager负责整个集群的统一资源管理。Executor是应用执行的主要进程，内部含有多个task线程以及内存空间。

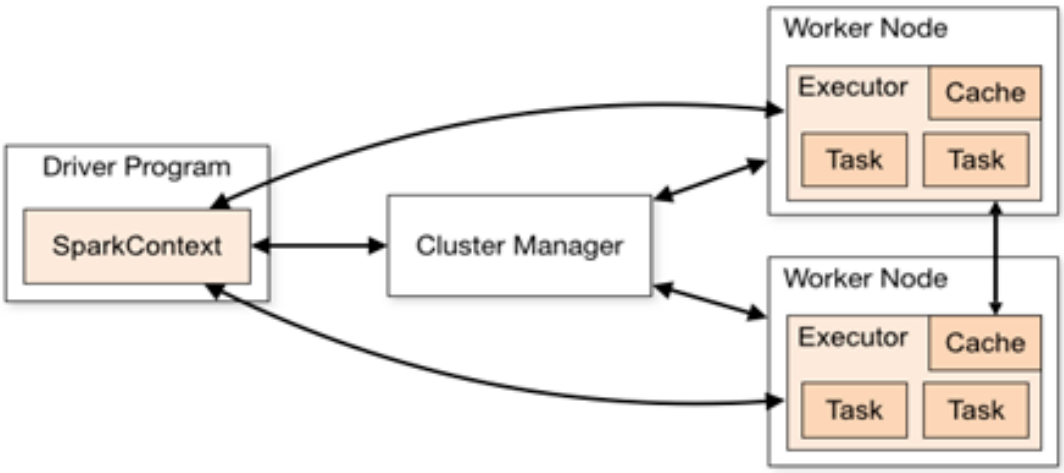

图 5 Spark分布式部署图

6.2. Spark内部执行流程

（1）提交Spark程序后，根据提交时指定（deploy-mode）的位置，创建driver进程，driver进程根据SparkConf中的配置，初始化SparkContext。SparkContext启动后，创建DAG Scheduler（将DAG图分解成stage）和Task Scheduler（提交和监控task）两个调度模块。

（2）Driver进程根据配置参数向Cluster manager申请资源（主要是用来执行任务的executor），Cluster Manager接到了Application的注册请求后，会使用自己的资源调度算法，在Spark集群的worker上，通知worker为Application启动多个executor。

（3）Executor创建后，会向Cluster Manager进行资源及状态反馈，以便Cluster Manager对executor进行状态监控，如监控到有失败的executor，则会立即重新创建。

（4）Executor会向taskScheduler反向注册，以便获取taskScheduler分配的task。

（5）Driver完成SparkContext初始化，继续执行Application程序，当执行到Action时，就会创建Job。并且由DAG Scheduler将Job划分为多个stage，每个stage由Taskset组成，并将Taskset提交给taskScheduler。TaskScheduler把TaskSet中的task依次提交给Executor，Executor在接受到task之后，会使用taskRunner（封装task的线程池）来封装task，然后从executor的线程池中取出一个线程来执行task。完成后释放相应的资源。

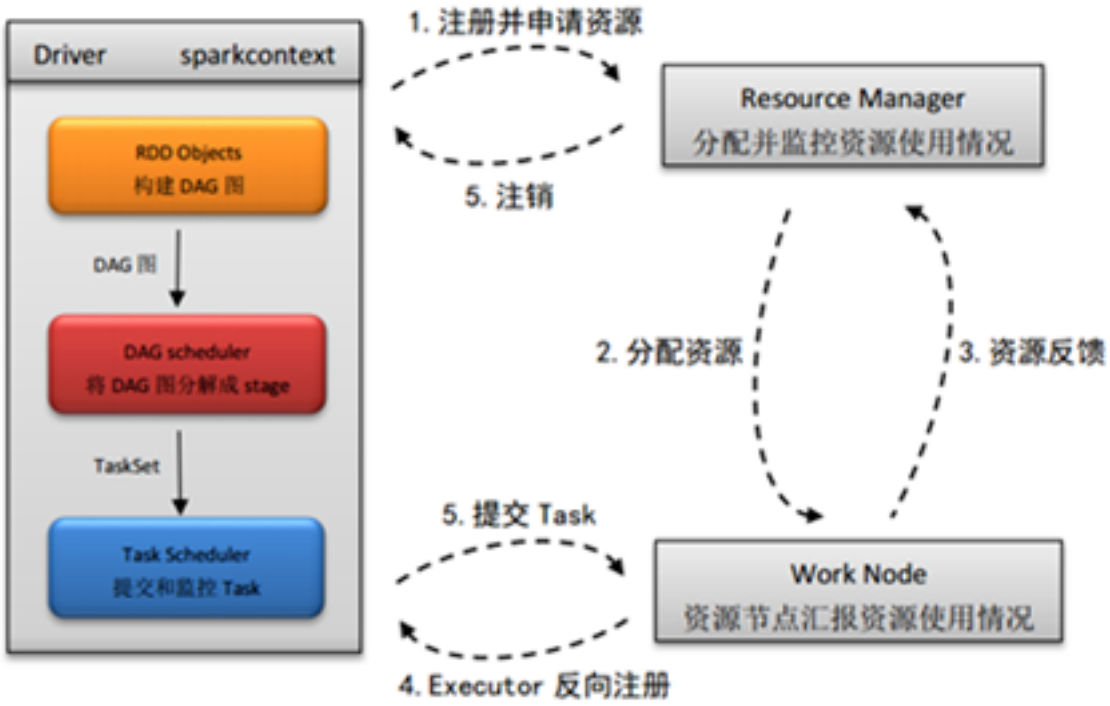

图 6 Spark应用执行流程图

6.3. Job、Stage、Task的关系

Job、Stage和Task是Spark任务执行流程中的三个基本单位。其中Job是最大的单位，Job是spark应用的action操作产生的；Stage是由Job拆分得到，在单个Job内是用shuffle算子来拆分stage，单个Stage内部可根据操作数据的分区数划分成多个task。

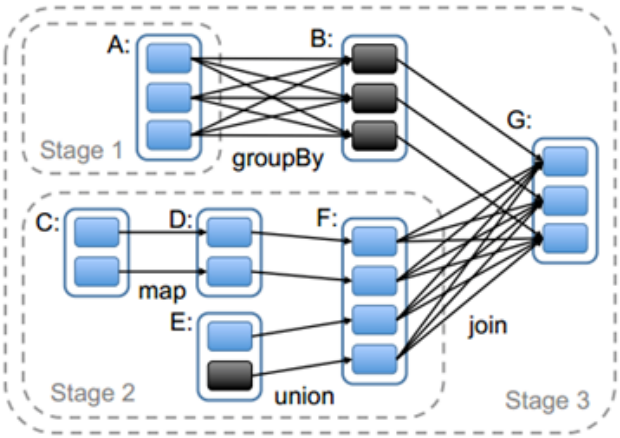

图 7 Job划分的示例

6.4. DAG Scheduler的工作流程

DAG Scheduler是一个高级的scheduler 层，他实现了基于stage的调度，他为每一个job划分stage，并将单个stage分成多个task，然后他会将stage作为taskSet提交给底层的Task Scheduler，由Task Scheduler执行。

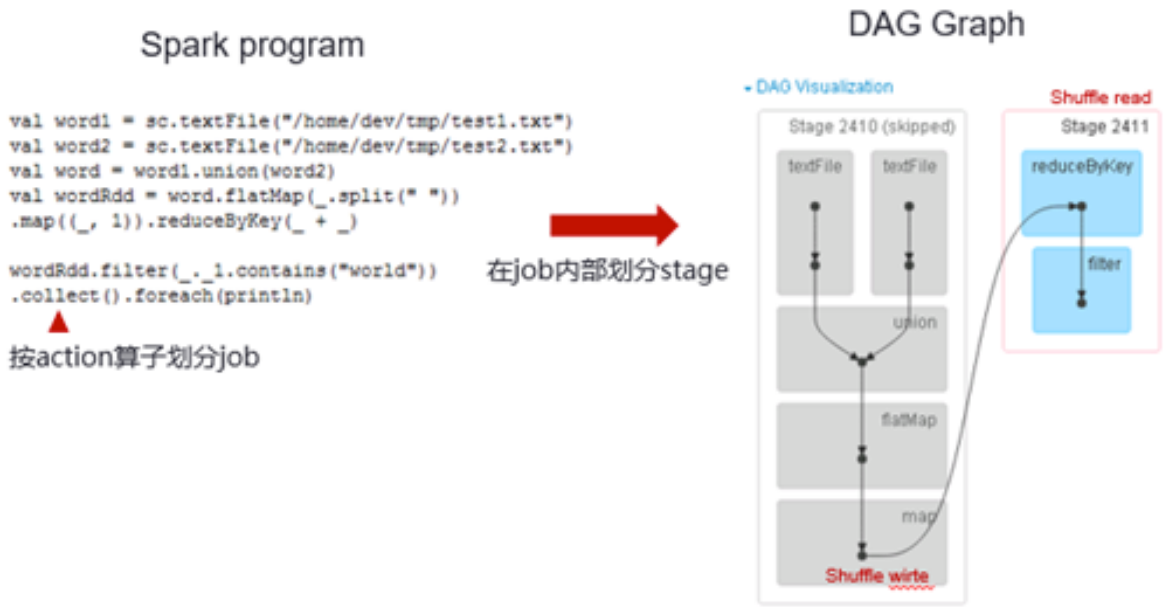

图 8 DAG Scheduler工作流程

针对左边的一段代码，DAG Scheduler根据collect（action算子）将其划分到一个job中，在此job内部，划分stage，如上右图所示。DAG Scheduler在DAG图中从末端开始查找shuffle算子，上图中将reduceByKey为stage的分界，shuffle算子只有一个，因此分成两个stage。前一个stage中，RDD在map完成以后执行shuffle write将结果写到内存或磁盘上，后一个stage首先执行shuffle read读取数据在执行reduceByKey，即shuffle操作。

6.5. TASK Scheduler的工作流程

Task Scheduler是sparkContext中除了DAG Scheduler的另一个非常重要的调度器，task Scheduler负责将DAGScheduer产生的task调度到executor中执行，一般的调度模式是FIFO（先进先出），也可以按照FAIR（公平调度）的调度模式，具体根据配置而定。其中FIFO：顾名思义是先进先出队列的调度模式，而FAIR则是根据权重来判断，权重可以根据资源的占用率来分，如可设占用较少资源的task的权重较高。这样就可以在资源较少时，调用后来的权重较高的task先执行了。至于每个executor中同时执行的task数则是由分配给每个executor中cpu的核数决定的。

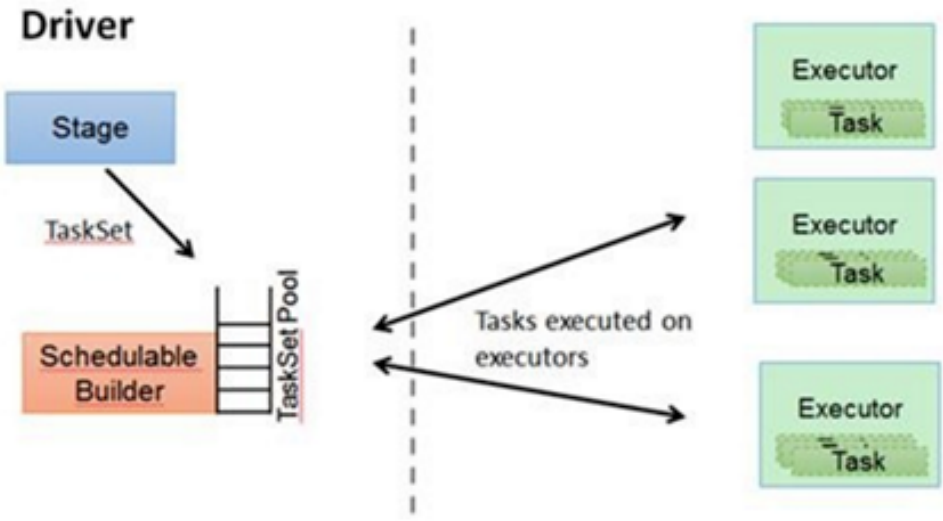

图 9 Task Scheduler工作流程

7. Spark SQL

Spark SQL是Spark用来操作结构化数据的程序包。通过Spark SQL，可以使用SQL或者Apache Hive版本的SQL方言来查询数据。Spark SQL支持多种数据源，比如Hive表、Parquet以及JSON等。除了为Spark提供了一个SQL接口，Spark SQL还支持开发者将SQL和传统的RDD变成的数据操作方式相结合，不论是使用Java、Python还是Scala。开发者都可以在单个的应用中同时使用SQL和复杂的数据分析。通过与Spark所提供的丰富的计算环境进行如此紧密的结合，Spark SQL得以从其他开源数据仓库工具中脱颖而出。

7.1. Spark SQL核心数据结构—Dataframe

在Spark中，DataFrame是一种以RDD为基础的分布式数据集，类似于传统数据库中的二维表格。DataFrame与RDD的主要区别在于，前者带有schema元信息，即DataFrame所表示的二维表数据集的每一列都带有名称和类型。这使得Spark SQL得以洞察更多的结构信息。

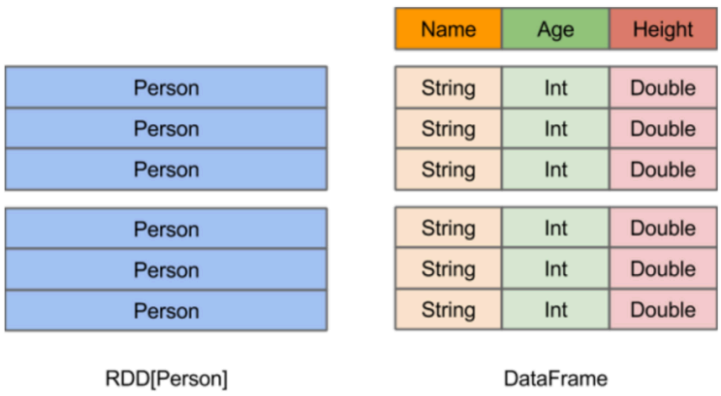

图 10 RDD与DataFrame比较

DataFrame与RDD相同之处，都是不可变分布式弹性数据集。不同之处在于，DataFrame的数据集都是按指定列存储，即结构化数据。类似于传统数据库中的表。 DataFrame的设计是为了让大数据处理起来更容易。DataFrame允许开发者把结构化数据集导入DataFrame，并做了higher-level的抽象; DataFrame提供特定领域的语言（DSL）API来操作你的数据集。图10直观地体现了DataFrame和RDD的区别。左侧的RDD[Person]虽然以Person为类型参数，但Spark框架本身不了解Person类的内部结构。而右侧的DataFrame却提供了详细的结构信息，使得Spark SQL可以清楚地知道该数据集中包含哪些列，每列的名称和类型各是什么。DataFrame多了数据的结构信息，即schema。RDD是分布式的Java对象的集合。DataFrame是分布式的Row对象的集合。DataFrame除了提供了比RDD更丰富的算子以外，更重要的特点是提升执行效率、减少数据读取以及执行计划的优化。

7.2 Spark SQL 语句执行的查询优化框架—Catalyst

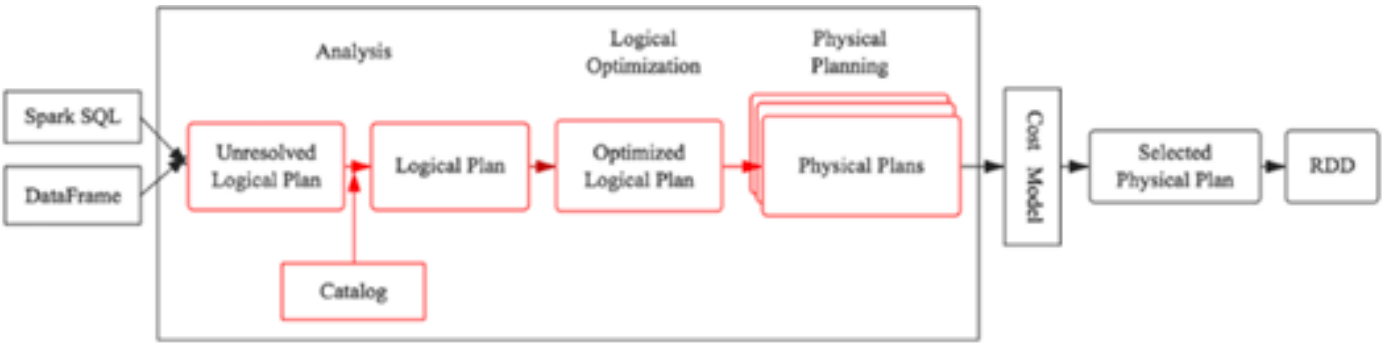

图 11 Spark sql执行流程图

Catalyst是Spark SQL的核心，是一套针对Spark SQL语句执行过程中的查询优化框架。图11的长方形框内为Catalyst的工作流程。SQL语句首先通过Parser模块被解析为语法树，此棵树称为Unresolved Logical Plan; Unresolved Logical Plan通过Analyzer模块借助于Catalog中的表信息解析为Logical Plan; 此时，Optimizer再通过各种基于规则的优化策略进行深入优化，得到Optimized Logical Plan; 优化后的逻辑执行计划并不能被Spark系统理解，需要将此逻辑执行计划转换为Physical Plan。

8. Spark Streaming

Spark Streaming是Spark体系中的一个流式处理框架，它能够与Spark SQL、Mlib和GraphX无缝衔接。Spark流可以从多种数据源获取数据，同时能够输出到多种不同的数据平台，包括文件系统、数据库和实时数据展示平台dashboards，如下图所示：

图 11 Spark Streaming的输入输出

Spark Streaming接受实时数据流输入的数据流后，将数据划分为一个个小批次的数据流(batch)，将每个小批次数据视为一个RDD，供后续Spark engine处理，所以实际上，Spark Streaming是按一个个batch来处理数据流的。

图 12 Spark Streaming数据处理流程

8.1. DStream

Spark Streaming的核心数据结构是DStream。DStream代表了一系列连续的RDD，DStream中每个RDD包含特定时间间隔的数据，存储方式为HashMap<Time, RDD>。其中，Time为时间序列，RDD为对应时间的弹性分布式数据集。

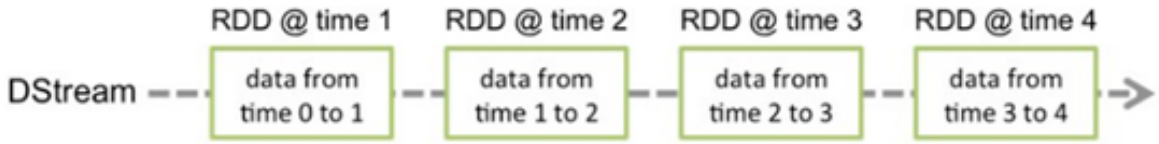

图 13 DStream结构

对连续不断的Streaming data流的多次切片，就会将流分成多个batch，单个batch内有一套针对多个DStream的处理逻辑，每个batch的处理逻辑相同。这个处理逻辑相当于Spark Core对RDD的处理逻辑。针对RDD的处理中，DAGScheduler将DAGGraph按照宽窄依赖划分成stage。同样的，每个batch内部也存在DStreamGraph，对DStream的处理也类似于对RDD的处理。例如下图所示，针对一段代码，在单个batch内部也会生成DStreamGraph和DStream依赖。

图 14 单个batch内部处理流程

针对一个Spark Streaming的处理流中的多个batch，处理逻辑如下图所示，图中用虚线将左侧的Streaming data流分成三个batch，每个batch的处理逻辑如右侧所示：

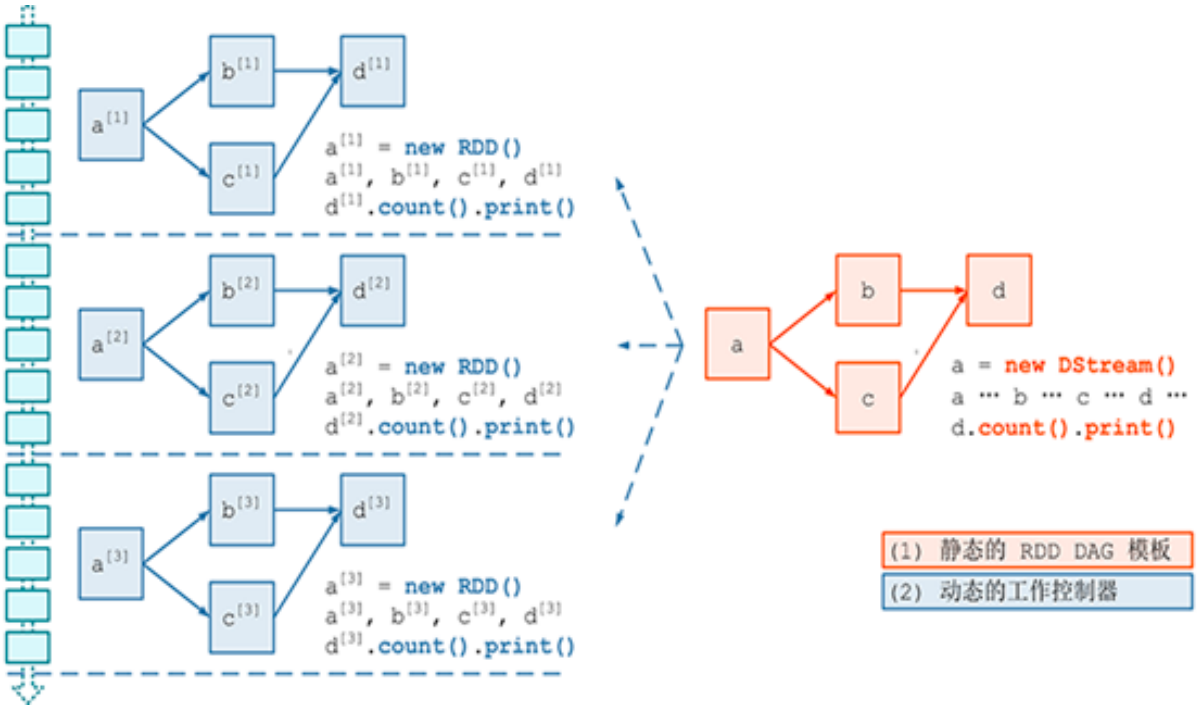

图 15 Streaming批量处理流程

8.2. Spark Streaming工作原理

Spark Streaming的需求大致有以下几点：

1. 需要一个DAG的静态模板来定义batch内的执行逻辑。
2. 针对实时的数据流来说，还需要有控制器，不间断地将数据流分成多个batch，同时在每个batch内部应用DAG静态模板执行处理逻辑。
3. 要生成DStream，并不能像一般的数据源那样从存储介质中去读取，而是要从多种数据源推送过来的数据获取，包括kafka、flume以及twitter等等。
4. 由于流式处理要不断地循环执行，保证任务的稳定性就显得尤其重要了。

Spark Streaming的整体执行流程就是围绕上述四个需求而设置的，其总体工作流程如下图：

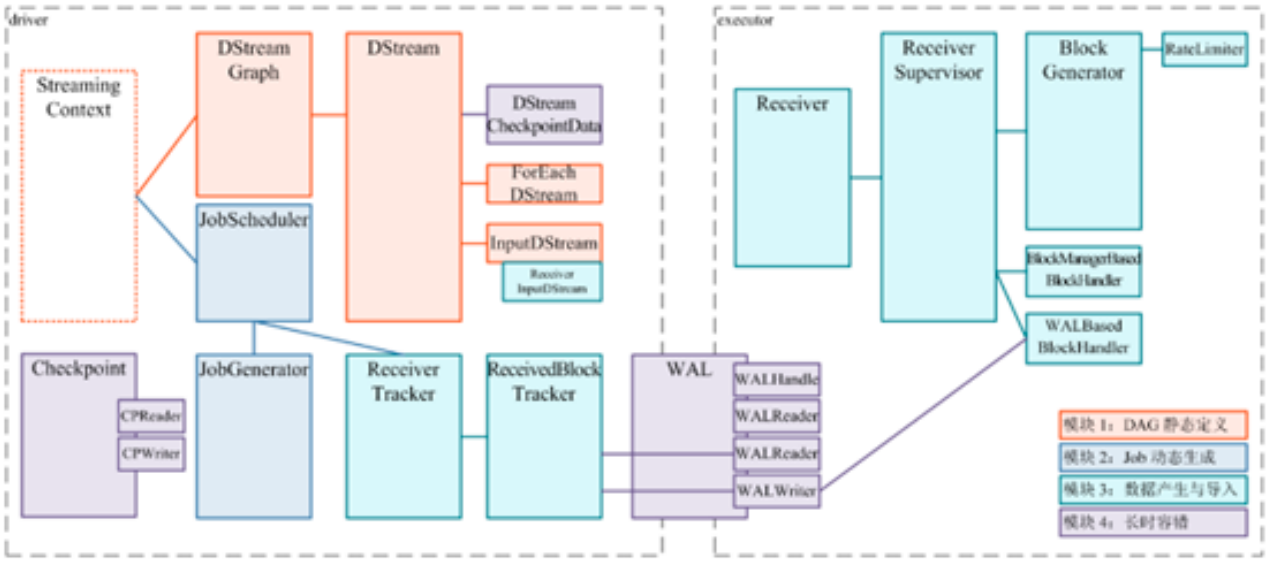

图 16 Spark Streaming工作原理图

各部分的主要职责为：

1. 橙色部分为DAG的静态定义部分，如图14、图15所示步骤生成DStreamGraph和DStream。
2. 淡蓝色为控制器部分，负责流的拆分，同时执行橙色部分定义的静态模板。JobScheduler是主要的控制器，负责动态任务的调度，包括JobGenerator和ReceiveTracker两个主要的成员。其中，JobGenerator主要负责将data streaming按照程序中设置的时间间隔切分成多个batch，并按照静态的DStreamGraph为以后的每个batch生成DStreamGraph。而Receive Tracker则负责数据流的接受跟踪和控制。
3. 绿色部分为driver和executor的数据接收部分。ReceiverTracker启动多个job，并分发到多个executor上。Executor启动Receiver supervisor, Receive supervisor启动Receiver来接受数据，Receiver supervisor接受到数据后，按块的形式存储，并将块上的meta信息上报给Receiver Tracker。
4. 紫色的部分负责稳定性的保障，即checkpoint。Receiver Tracker接收到块的meta信息后交给ReceiverBlock Tracker去管理块信息。ReceiverBlock Tracker进行备份，在driver失效后，由新的ReceiverBlock Tracker读取并恢复block的meta信息。这部分主要是出于容错率的考虑，设置checkpoint机制。因此，checkpoint需要将整个处理流程中的关键节点都做checkpoint，包括DStreamGraph，JobScheduler，数据快的meta信息以及块数据。

9. Spark MLib

MLib是Spark中可扩展的机器学习库，它由一系列机器学习算法和实用程序组成，包括分类、回归、聚类、协同过滤、降维，还包括一些底层的优化方法。MLib的底层实现采用数值计算库Breeze和基础的线性代数库BLAS。

1. 优化计算：MLib目前支持随机梯度下降法、少内存拟牛顿法、最小二乘法等。

2. 回归：MLib目前支持线性回归、逻辑回归、岭回归、保序回归和与之相关的L1和L2正则化的变体。MLib中回归算法的优化计算方法采用随机梯度下降法。
3. 分类：MLib目前支持贝叶斯分类、决策树分类、线性SVM和逻辑回归，同时也包括L1和L2正则化的变体。优化计算方法也采用随机梯度下降法。
4. 聚类：MLib目前支持KMeans聚类算法、LDA主题模型算法。
5. 推荐：MLib目前支持ALS推荐，采用交替最小二乘求解的协同推荐算法。
6. 关联规则：MLib目前支持FPGrowth关联规则挖掘算法。

10. Spark GraphX

GraphX是Apache Spark用于图和图并行计算的API。它扩展了Spark RDD，引入了一个新的图抽象：有向多图（directed multigraph），每个节点和边都有自己的属性。

GraphX库提供了图算子（operator）来转换图数据，如subgraph、joinVertices和aggregateMessages。它提供了几种方法来从RDD或硬盘上的一堆节点和边中来构建一个图。它也提供了许多图算法和构造方法来进行图分析。通过内嵌的算子和算法，GraphX使得在图数据上运行分析变得更加容易。它还允许用户cache和uncache图数据，以在多次调用图的时候避免出现重复计算。

算子类型	算子
基本算子	• numEdges • numVertices • inDegrees • outDegrees • degrees
属性算子	• mapVertices • mapEdges • mapTriplets
结构算子	• reverse • subgraph • mask • groupEdges
关联算子	• joinVertices • outerJoinVertices

图 17 Spark GraphX的图算子

10.1. GraphFrames

GraphFrames是Spark图数据处理工具集的一个新工具，它将模式匹配和图算法等特征与Spark SQL整合在一起。节点和边被表示为DataFrames，而不是RDD对象。

GraphFrames简化了图数据分析管道，优化了对图数据和关系数据的查询。与基于RDD的图处理相比，GraphFrames有下列优势：

1. 在Scala API之外，还支持Python和Java。我们现在可以在这三门语言中使用GraphX算法。

2. 用Spark SQL和DataFrames获得更高级的查询能力。Graph-aware query planner使用物化视图来提高查询性能。我们也可以用Parquet、JSON和CSV等格式来存储和导入图。

11. 总结

Spark基于map reduce算法实现的分布式计算，拥有Hadoop MapReduce所具有的优点；但不同于MapReduce的是Job中间输出和结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的map reduce的算法。

1. Spark的特点：

（1）运行速度快：Spark拥有DAG执行引擎，支持在内存中对数据进行迭代计算。官方提供的数据表明，如果数据由磁盘读取，速度是Hadoop MapReduce的10倍以上，如果数据从内存中读取，速度可以高达100多倍。

（2） 使用场景广泛：可应用于大数据分析统计，实时数据处理，图计算以及机器学习。

（3）易用性：编写简单，支持80种以上的高级算子，支持多种语言，数据源丰富，可部署在多种集群。

（4）容错性高：Spark引进了弹性分布式数据集RDD的概念，它是分布在一组节点中的只读对象集合，这些集合是弹性的，如果一部分数据丢失，可以根据谱系图（lineage graph）对它们进行重新计算。此外，RDD计算时可以通过CheckPoint来实现容错，而CheckPoint有两种方式：数据复制或日志记录，用户可以控制采用哪种方式来实现容错。

2. Spark的适用场景：

（1）复杂的批处理，偏重点在于处理海量数据的能力，通常处理时间在数十分钟到数小时。

（2）基于历史数据的交互式查询，通常延迟在数十秒到数十分钟之间。

（3）基于实时数据流的数据处理，通常延迟在数百毫秒到数秒之间。

同时，Spark也存在一些不足之处，JVM的内存开销较大；在不同的Spark Application之间缺乏有效的数据共享机制。