High Performance Spark - June 3'rd, 2018

How Spark Works

https://github.com/rh01

- Spark 往往被看作是 Apache MapReduce 的替代版本，比传统的 MapReduce 范式要更加的方便和高效;
- Spark 不需要非得和 Apache Hadoop 一起使用，虽然在实践中常常这样;
  - 因为 Spark 自身已经继承了 Hadoop 的一些 API、设计和支持来自其他现有的计算框架的数据格式，比如 DryadLINQ;
- Spark 在处理故障方面，内部的实现以及工作原理和传统的系统是不一样的;
- Spark 在内存计算方面中采用了延迟执行（lazy evaluation）;
- Spark 可以看成快速处理和分析分布式数据的更高级的语言模型.

Microsoft Dryad

- Dryad 是对 MapReduce 模型的一种扩展
  - 组成单元不仅是 Map 和 Reduce，可以是多种节点
  - 节点之间形成一个有向无环图 DAG(Directed Acyclic Graph) ，以表达所需要的计算
  - 节点之间的数据传输模式更加多样
    * 可以是类似 Map/Reduce 中的 shuffle
    * 也可以是直接 1:1、1: 多、多:1 传输
  - 比 MapReduce 更加灵活，但也更复杂
    * 需要程序员规定计算的 DAG

Agenda

- Spark 的设计原则
- Spark 程序的执行方式
- Spark 的并行计算模型
- Spark 的调度和执行引擎（Schedual 和 Executor ）

Spark 如何适应大数据生态系统

- Spark 是 Apache 软件基金会的顶级开源项目，原本由伯克利大学开发, Scala 开发，先支持多种编程语言，比如：Java，Scala，Python, ...;
- Spark 提供了可以推广（泛化）的并行处理数据的方法 $\longrightarrow$ MapReduce;
  - 相同的高级 Spark 函数可以对不同大小和结构的数据执行不同的数据处理任务.
- Spark 本身并不是一种数据存储解决方案
- Spark 在 Spark JVM（Java 虚拟机）上执行计算，这些计算仅持续 Spark 应用程序运行时间.
- 如图 1 所示，Spark 和分布式存储系统 (e.g. HDFS, Cassandra, or S3) 、集群管理器以及用于协调 Spark 应用程序分发的集群管理器一起使用;（常见）
  - 集群管理器 $\longrightarrow$ The storage system to house the data processed with Spark.
  - Spark 目前支持三种集群管理器：Standalone Cluster Manager，Apache Mesos 和 Hadoop YARN.

* 独立集群管理器包含在 Spark 中，若使用独立管理器需要在集群的每个节点上安装 Spark.

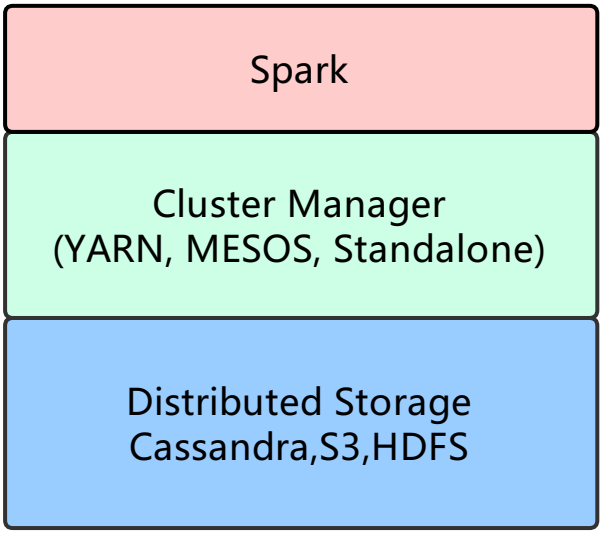

图 1: A diagram of data-processing ecosystem including Spark.

Spark 的组件

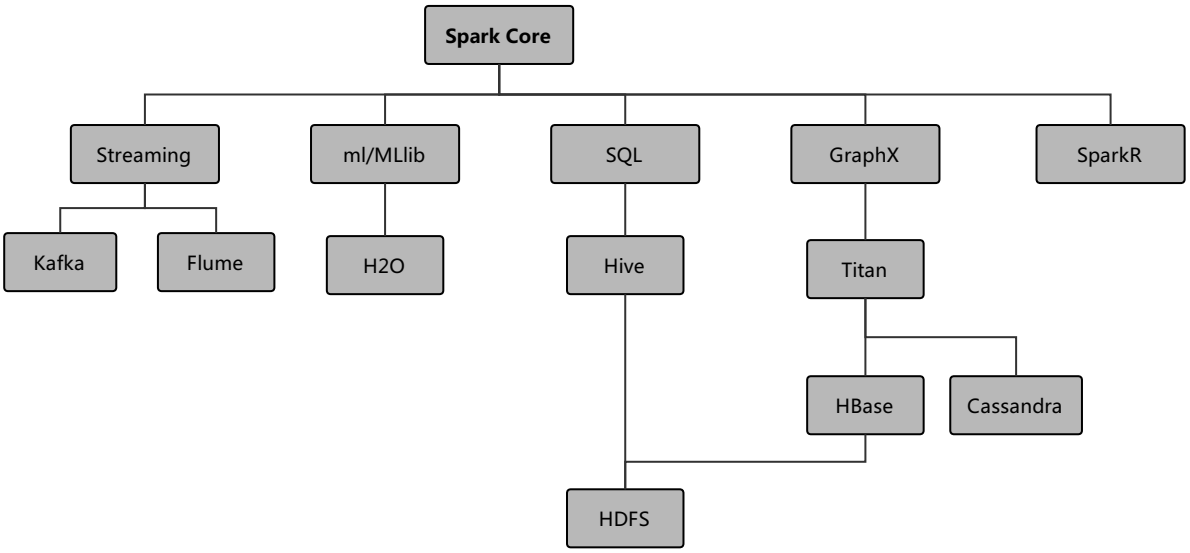

图 2: Spark 的模块.

- Spark Core, the main data processing framework in the Spark ecosystem, has APIs in Scala, Java, Python, and R.
  - Spark is built around a data abstraction called Resilient Distributed Datasets (RDDs)
    * RDDs are a representation of **lazily evaluated, statically typed, distributed collections**.

    * RDDs have a number of predefined "coarse-grained"(粗粒度) transformations (functions that are applied to the entire dataset), such as map, join, and reduce to manipulate the distributed datasets, as well as I/O functionality to read and write data between the distributed storage system and the Spark JVMs.

- Spark SQL is a component that can be used in tandem with Spark Core and has APIs in Scala, Java, Python, and R, and basic SQL queries.
  - Spark SQL 定义了一种称为 `DataFrames` 的半结构化数据类型的接口，以及在 Spark 1.6 版本中，Spark SQL 又定义了一种名为 `Datasets` 的半结构化类型的 RDD.
  - Spark SQL 对 Spark 性能有很大的影响
  - Spark SQL 可以完成和 Spark Core 同样的功能.
- Spark ML 和 Spark MLlib 是机器学习包. MLlib 是用机器学习与统计算法的工具包，用 Spark 写的，而 Spark ML 仍处于早期阶段，自 Spark 1.2 版本中才开始存在.
  - Spark ML 提供了比 MLlib 更高级别的 API，其目标是允许用户更轻松地创建实用的机器学习流程（pipeline）.
  - Spark MLlib 主要构建在 RDD 之上并使用 Spark Core 的功能，而 ML 则构建在 Spark SQL `DataFrames` 之上. 最终，Spark 社区计划转移到 ML 并弃用 MLlib.
  - Spark ML 和 MLlib 都有 Spark Core 和 Spark SQL 的其他性能考虑因素.
- Spark Streaming 使用 Spark Core 的调度来对小型数据进行流式分析.
  - Spark Streaming has a number of unique considerations, such as the window sizes used for batches(批次的窗口大小).
- GraphX 是 Spark 中最不成熟的组件之一.

Spark Model of Parallel Computing: RDDs

- Spark 的基础数据结构：RDD
  - Resilient Distributed Data sets
    * 一个数据集
    * 只读，整个数据集创建后不能修改
    * lazily, computing RDD transformations only when the final RDD data needs to be computed (often by writing out to storage or collecting an aggregate to the driver).
    * 通常进行整个数据集的运算
    * 可以加载到执行程序节点上的内存中，以便在重复计算中更快地访问.（可重用）
  - 优点
    * 并发控制被简化了
    * 可以记录 lineage（数据集上的运算序列），可以重新计算
      · 并不需要把 RDD 存储在 stable storage 上

lazy evaluation

- 普通内存计算是基于可变对象的"fine-grained"细粒度更新.
  - i.e., Calls to a particular cell in a table by storing intermediate results.
- RDD 的计算（evaluation）是完全懒惰的

- Spark 只有调用了 `Action`，Spark 才开始计算分区.
  - An action is a Spark operation that **returns something other than an RDD**, **triggering evaluation of partitions and possibly returning some output to a non-Spark system** (outside of the Spark executors);
  - i.e., Bringing data back to the driver (with operations like `count` or `collect`) or writing data to an external storage storage system (such as `copyToHadoop`).
  - Action 触发调度程序 (Schedualer)，该调度程序基于 RDD 转换 (Transformations) 之间的依赖关系构建有向非循环图（称为 DAG）；
  - Spark 反向来执行操作，定义一个 DAG. 然后调用执行计划（Execute Plans），通过执行计划来执行之前上面定义的一系列步骤（DAG），调度程序计算每个阶段的缺失分区，直到计算结果为止.

lazy evaluation 的性能和优点

- 延迟执行允许 Spark 将不需要与驱动程序通信的操作（也可以称为具有一对一依赖关系的转换）组合在一起，以避免对数据进行多次传递.
  - 例如，假设 Spark 程序在同一个 RDD 上调用 `map` 和 `reduce` 函数. Spark 可以将 `map` 和 `reduce` 的指令发送给每个执行程序. 然后 Spark 可以在每个分区上执行 `map` 和 `reduce`，这只需要访问记录一次，而不是发送两组指令并访问每个分区两次. 理论上，这将计算复杂度降低了一半.
- 相比 MapReduce 来说，易于实现.
  - Java Version: **15 lines**, Scala Version: **5 lines**.
- 易于修改和改进.
  - Suppose that we now want to modify this function to filter out some "stop words" and punctuation from each document before computing the word count.
  - Java Version: **Much lines**, Scala Version: **7 lines**.

```
1  public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
       output, Reporter reporter) throws IOException {
2      String line = value.toString();
3      StringTokenizer tokenizer = new StringTokenizer(line);
4      while (tokenizer.hasMoreTokens()) {
5          word.set(tokenizer.nextToken());
6          output.collect(word, one);
7      }
8  }
9  public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
       IntWritable> output, Reporter reporter) throws IOException {
10     int sum = 0;
11     while (values.hasNext()) {
12         sum += values.next().get();
13     }
14     output.collect(key, new IntWritable(sum));
15 }
```

Listing 1: Simple Java word count example

```scala
def simpleWordCount(rdd: RDD[String]): RDD[(String, Int)] = {
    val words = rdd.flatMap(_.split(" "))
    val wordPairs = words.map((_, 1))
    val wordCounts = wordPairs.reduceByKey(_ + _)
    wordCounts
}
```

Listing 2: Simple Scala word count example

```java
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    private boolean caseSensitive = false;
    private long numRecords = 0;
    private String input;
    private Set<String> patternsToSkip = new HashSet<String>();
  private static final Pattern WORD_BOUNDARY = Pattern.compile("\\s*\\b\\s*");

    protected void setup(Mapper.Context context)
        throws IOException,
        InterruptedException {
      if (context.getInputSplit() instanceof FileSplit) {
        this.input = ((FileSplit) context.getInputSplit()).getPath().toString();
      } else {
        this.input = context.getInputSplit().toString();
      }
      Configuration config = context.getConfiguration();
      this.caseSensitive = config.getBoolean("wordcount.case.sensitive", false);
      if (config.getBoolean("wordcount.skip.patterns", false)) {
        URI[] localPaths = context.getCacheFiles();
        parseSkipFile(localPaths[0]);
      }
    }

    private void parseSkipFile(URI patternsURI) {
      try {
        BufferedReader fis = new BufferedReader(new FileReader(new File(
            patternsURI.getPath()).getName()));
        String pattern;
        while ((pattern = fis.readLine()) != null) {
          patternsToSkip.add(pattern);
        }
      } catch (IOException ioe) {
        System.err.println("Caught exception while parsing the cached file '"
            + patternsURI + "' : " + StringUtils.stringifyException(ioe));
      }
    }

    public void map(LongWritable offset, Text lineText, Context context)
        throws IOException, InterruptedException {
```

```
41          String line = lineText.toString();
42          if (!caseSensitive) {
43            line = line.toLowerCase();
44          }
45          Text currentWord = new Text();
46        for (String word : WORD_BOUNDARY.split(line)) {
47        if (word.isEmpty() || patternsToSkip.contains(word)) {
48          continue;
49        }
50          currentWord = new Text(word);
51          context.write(currentWord,one);
52        }
53        }
54      }
55
56      public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable>
            {
57        @Override
58        public void reduce(Text word, Iterable<IntWritable> counts, Context context)
59            throws IOException, InterruptedException {
60          int sum = 0;
61          for (IntWritable count : counts) {
62            sum += count.get();
63          }
64          context.write(word, new IntWritable(sum));
65        }
66      }
67    }
```

Listing 3: Java Word count example with stop words filtered

```
1   def withStopWordsFiltered(rdd : RDD[String], illegalTokens : Array[Char],
2       stopWords : Set[String]): RDD[(String, Int)] = {
3       val separators = illegalTokens ++ Array[Char](' ')
4       val tokens: RDD[String] = rdd.flatMap(_.split(separators).
5           map(_.trim.toLowerCase))
6       val words = tokens.filter(token =>
7           !stopWords.contains(token) && (token.length > 0) )
8       val wordPairs = words.map((_, 1))
9       val wordCounts = wordPairs.reduceByKey(_ + _)
10      wordCounts
11  }
```

Listing 4: Scala Word count example with stop words filtered

Lazy evaluation and fault tolerance

- Spark 具有容错能力
  - 当主机或者网络发生故障时，Spark 不会失败、丢失数据或者返回不正确的结果.
  - 数据的每个分区都包含着依赖信息, 可以根据以来信息进行重新计算.

- 大多数分布式计算范式/框架使用可变对象作为数据结构，通过记录更新日志或跨机器复制数据来保证容错.
- 相反，Spark 不需要维护每个 RDD 的更新日志或者记录实际的中间结果.
  - RDD 自身包括了复制每个分区所需的所有依赖性信息.
  - 如果分区丢失，RDD 具有足够的信息来重新计算，并且可以并行化使得恢复更快.

Lazy evaluation and debugging

- 延迟计算对调试具有重要影响，意味着 Spark 程序仅在 `action` 处失败.

In-Memory Persistence and Memory Management

- 在重复计算场景中，具有很大优势.
- Spark 使用了内存持久化
  - Spark 并不是在每次传递数据时写入磁盘，而是将执行程序上的数据保存在内存上.
- 提供了三种内存管理的方式
  - In memory as deserialized Java objects
    * 在 RDD 中存储对象的最直观方式是使用驱动器类，将原始数据反序列化为 Java 对象. 这种形式的内存存储是最快的，因为它减少了序列化时间; 但是，它可能不是最有效的内存存储，因为它需要将数据存储为对象.
  - In memory
    * 使用标准 Java 序列化库，Spark 对象在网络中传递时会转换为字节流. 这种方法可能会慢，因为序列化数据比反序列化数据更难以读取 CPU; 但是，它通常更节省内存，因为它允许用户选择更有效的表示. 虽然 Java 序列化比完整对象更有效，但 Kryo 序列化可以更加节省空间.
  - On disk
    * 由于 RDD 的分区太大而无法存储在每个执行程序的 RAM 中，这时可以写入磁盘. 对于重复计算，此策略显然较慢，但对于很长的变换序列可能更具容错性，并且可能是进行大规模计算的唯一可行选项.
- The `persist()` function in the RDD class lets the user control how the RDD is stored. By default, `persist()` stores an RDD as deserialized objects in memory, but the user can pass one of numerous storage options to the `persist()` function to control how the RDD is stored.

Immutability

Spark 定义了一个 RDD 接口（`trait`），其中包含每一种 RDD 必须实现的属性. 这些属性包括 RDD 的依赖关系以及执行引擎计算该 RDD 所需的有关数据局部性的信息. 由于 RDD 是静态类型且不可变的，因此在一个 RDD 上调用 `transform` 是不会修改原始 RDD，而是返回具有 RDD 属性的新 RDD 对象.

有以下三种方式可以创建 RDD：

- 在现有的 RDD 上进行 `transform` 操作.
- 通过 `SparkContext` 创建.
  - The `SparkContext` can be used to create an RDD from a local Scala object (using the `makeRDD` or `parallelize` methods) or by reading from stable storage (text files, binary files, a Hadoop

Context, or a Hadoop file).

- 将 `DataFrame` 或者 `Dataset` 对象转化为 RDD.
  - `DataFrame` and `Dataset` can be read using the Spark SQL equivalent to a `SparkContext`, the `SparkSession`.

The RDDs Interface

- 在内部，Spark 有以下五个主要的属性/函数
  - `partition()` $\longrightarrow$ A list of partitions
  - `iterator(p, parentIters)` $\longrightarrow$ A function for computing each split
  - `dependencies()` $\longrightarrow$ A list of dependencies on other RDDs
  - `Partitioner` $\leftrightarrow$ Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
  - `preferredLocations(p)` $\longrightarrow$ Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)
- Spark API 还包含 RDD 类的实现，它们通过覆盖 RDD 的核心属性来定义更具体的行为. 这些包括前面讨论过的 `NewHadoopRDD` 类 $\rightarrow$ 代表从 HDFS 文件系统创建的 RDD 和 `ShuffledRDD`$\rightarrow$ 它代表已经分区的 RDD. 这些实现了 RDD 类的所有类包含了特定于该类型的 RDD 的功能. 通过转换或从 `SparkContext` 创建 RDD 将返回实现 RDD 的某个类的对象. 某些 RDD 操作在 Java 中具有与 Scala 不同的签名.
- RDD 有两类运算 Transformations 和 Action, 这些函数被定义在 RDD 函数类里，`PairRDDFunctions`, `OrderedRDDFunctions` 和 `GroupedRDDFunctions`.
  - Transformation
    - 输入是 RDD(数据集)
    - 输出也是 RDD(数据集)
    - RDD $\Longrightarrow$ RDD
  - Action
    - Spark 程序中必须包括一个 action 操作.
    - 输入是 RDD(数据集)
    - 输出是某种计算结果（例如，一个数值或者一列数值）或者把数据写入到文件系统中.
      - 注意：RDD 可能非常大，但是计算结果总是比较小的
    - RDD $\Longrightarrow$ 计算结果

Trans.	$map(f : T \Rightarrow U)$: RDD[T] ⇒ RDD[U] $filter(f : T \Rightarrow Bool)$: RDD[T] ⇒ RDD[T] $flatMap(f : T \Rightarrow Seq[U])$: RDD[T] ⇒ RDD[U] $sample(fraction : Float)$: RDD[T] ⇒ RDD[T] (Deterministic sampling) $groupByKey()$: RDD[(K, V)] ⇒ RDD[(K, Seq[V])] $reduceByKey(f : (V, V) \Rightarrow V)$: 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)] $mapValues(f : V \Rightarrow W)$: 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)]
Action	$count()$: RDD[T] ⇒ Long $collect()$: RDD[T] ⇒ Seq[T] $reduce(f : (T, T) \Rightarrow T)$: 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

表 1: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Wide Versus Narrow Dependencies

为了更好的理解 RDD 内部如何计算的，最重要的是要知道转换操作有两类：

- 窄依赖的 transformation
- 宽依赖的 transformation

- 窄变换是子 RDD 中的每个分区对父 RDD 中的分区具有简单、有限依赖性的变换.
  - 如果可以在设计时确定依赖关系，则无论父分区中的记录值如何，以及每个父级最多只有一个子分区，依赖关系都会很窄. 具体而言，窄变换中的分区可以依赖于一个父级（例如在映射运算符中），也可以依赖于在设计时已知的父分区的唯一子集（合并）. 因此，可以对数据的任意子集执行窄变换，而无需关于其他分区的任何信息.
- 宽依赖的变换不能在任意行 (e.g.,wordcount) 上执行，而是要求以特定方式对数据进行分区，
  - 例如，在排序中，根据其键的值对记录进行分区，以使相同范围内的键位于同一分区上.
  - 具有广泛依赖性的转换包括 `sort`，`reduceByKey`，`groupByKey`，`join` 以及 `rePartition`.

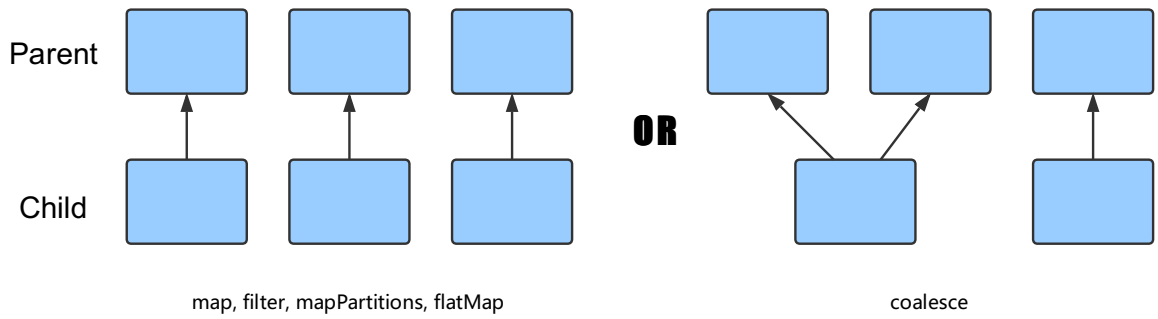

图 3: A simple diagram of dependencies between partitions for narrow transformations.

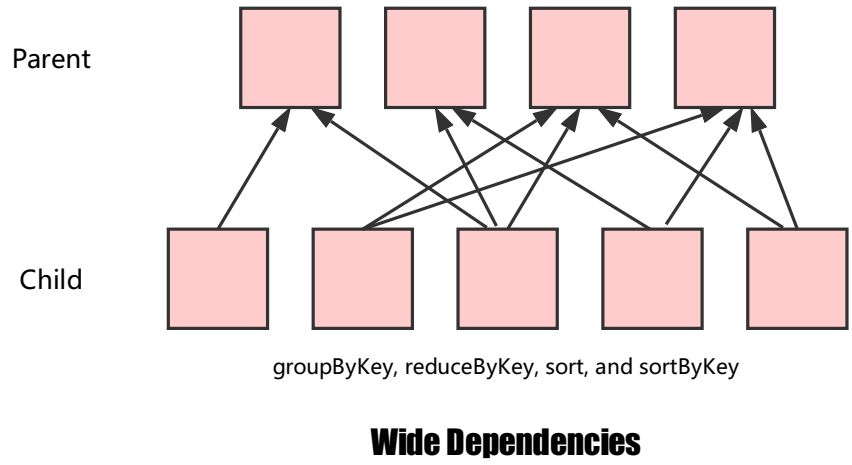

图 4: A simple diagram of dependencies between partitions for wide transformations.

Spark Job Scheduling

- Spark 应用程序由一个驱动程序进程和一系列执行程序进程组成，
  - 驱动程序进程是编写 Spark 高层逻辑的地方;
  - 而一系列执行程序进程可以分散在集群的各个节点上.
- Spark 程序本身在驱动程序节点中运行，并向执行程序发送一些指令.
- 一个 Spark 集群可以同时运行多个 Spark 应用程序. 应用程序由集群管理器调度，并对应于一个 SparkContext.
- Spark 应用程序可以运行多个并发作业. 作业对应于给定应用程序中 RDD 上调用的每个操作.

Resource Allocation Across Applications

- Spark 提供了两种跨应用程序分配资源的方法：
  - 静态分配
    * 每个应用程序在集群上分配有限的最大资源，并在应用程序的持续时间内保留它们（只要

SparkContext 仍在运行）.

- 动态分配
  - Spark1.2 开始出现
  - 根据需要, 在 Spark 应用程序中自动添加和删除执行程序.

The Spark Application

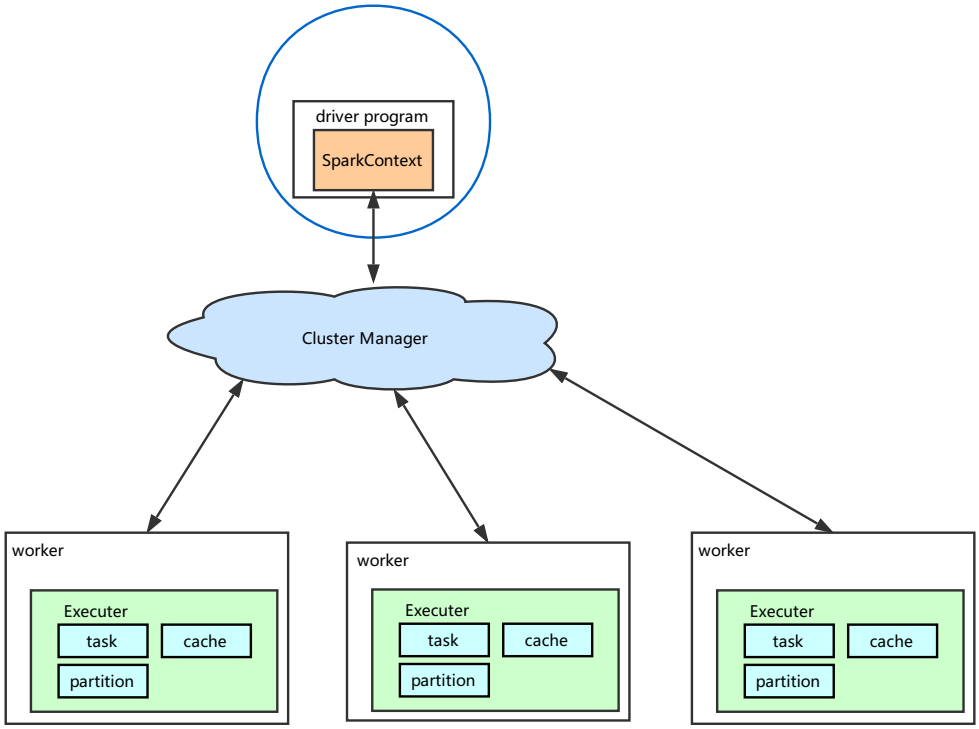

图 5: 单个 Spark 应用程序的运行架构.

- 每个应用程序
  - 一个自己的 SparkContext，多个 Executor
  - SparkContext 从外部的某种资源管理系统获取资源
    - 例如：standalone, hadoop YARN, apache Mesos
  - 每个 executor 运行在一个不同的 worker node 上
  - SparkContext 协调多个 worker 运行
- 应用程序
  - 有一个 driver 主程序，创建 SparkContext，发出各种 RDD 操作要求
- Executor：执行并行的运算，存储数据
- 多个应用程序
  - 各自有自己的 SparkContext
    - 互相隔离，但是也无法共享数据
  - 必须通过外部的文件系统进行数据共享

Default Spark Scheduler

默认情况下，Spark 以先进先出的方式安排作业. 但是，Spark 确实提供了一个公平的调度程序，它以循环方式将任务分配给并发作业，即为每个作业分配一些任务，直到作业全部完成为止. 公平调度程序可确保作业获得更均匀的群集资源份额. 然后，Spark 应用程序按照在 SparkContext 上调用相应操作的顺序启动作业.

The Anatomy of a Spark Job

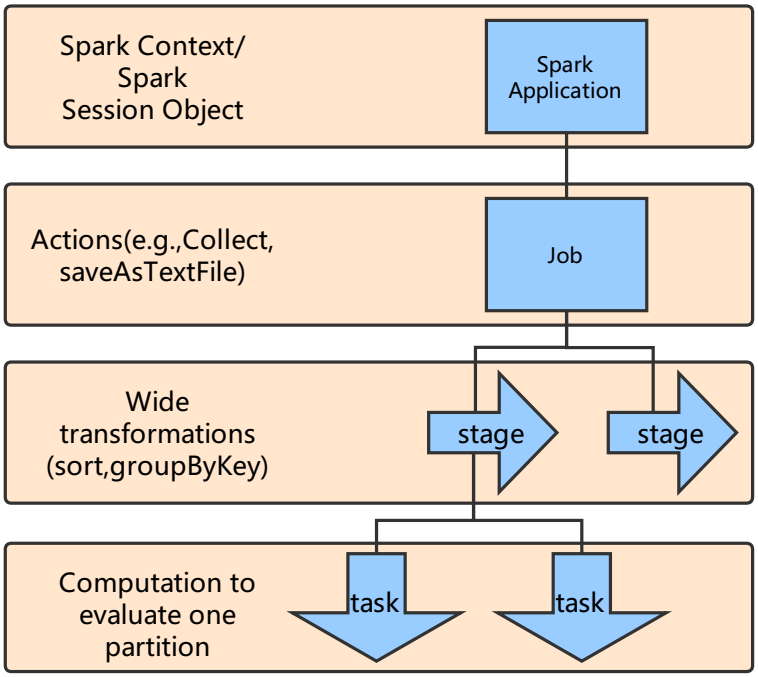

图 6: The Spark application tree.

DAG

Spark 的高级调度层使用 RDD 依赖关系为每个 Spark Job 构建 Stage 的有向无环图（DAG）. 在 Spark API 中，这称为 DAG 调度程序. 您可能已经注意到，与连接到群集，配置参数或启动 Spark 作业有关的错误显示为 DAG Scheduler 错误. 这是因为 DAG 处理 Spark 作业的执行. DAG 为每个作业构建一个阶段图，确定运行每个任务的位置，并将该信息传递给 `TaskScheduler`，后者负责在集群上运行任务. `TaskScheduler` 创建一个包含分区之间依赖关系的图形.

Jobs

Spark 作业是 Spark 执行层次结构的最高元素. 每个 Spark 作业对应一个动作，每个动作由 Spark 应用程序的驱动程序调用.

Stages

回顾一下延迟计算，在调用操作之前不会执行转换. 如前所述，通过调用 `action` 来定义作业. 一个动作可以包括一个或多个转换，并且宽依赖转换（wide transformations）将作业细分为阶段 (*stages*) .

每个阶段对应于 Spark 程序中的宽依赖转换所创建的 shuffle 依赖关系. 在高层，一个阶段可以被认为是一组计算任务，每个计算可以在一个执行器（Exector）上计算，而无需与其他执行器或驱动程序通信. 换句话说，只要在 worker 之间的存在网络通信，就会开始新的阶段; 例如，`shuffle`.

这些创建阶段边界的依赖项称为 `ShuffleDependencies`. `shuffle` 是由那些需要在分区中重新分配数据的 `sort` 或 `groupByKey` 等宽依赖转换引起的. 具有窄依赖性的若干变换可以组合为一个阶段. 正如我们在单词计数示例中看到的那样，我们过滤了停用词（*Listing2.,*），Spark 可以将 `flatMap`，`map` 和 `filter` 组合成一个阶段，因为这些转换都不需要 shuffle. 因此，每个执行器可以在数据的一次传递中连续地应用 `flatMap`，`map` 和 `filter`.

Tasks

一个阶段包括若干任务. 任务是执行层次结构中的最小单元，每个单元可以表示一个本地计算. 一个阶段中的所有任务在不同的数据上执行相同的代码. 一个任务不能在多个执行程序上执行. 但是，每个执行程序都有一个动态分配的用于运行任务的插槽数，并且可以在其生命周期内同时运行许多任务. 每个阶段的任务数对应于该阶段的输出 RDD 中的分区数.

```
1  def simpleSparkProgram(rdd : RDD[Double]): Long ={
2      //stage1
3      rdd.filter(_< 1000.0)
4        .map(x => (x, x) )
5      //stage2
6        .groupByKey()
7        .map{ case(value, groups) => (groups.sum, value)}
8      //stage 3
9        .sortByKey()
10       .count()
11 }
```

Listing 5: Different types of transformations showing stage boundaries

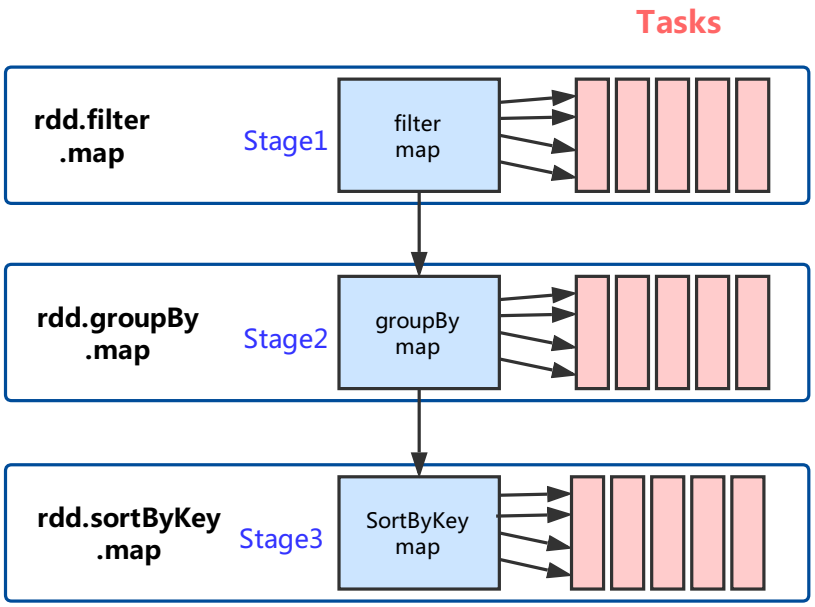

图 7: A stage diagram for the simple Spark program shown in Listing 7.