1. 安装spark，下载：<http://spark.apache.org/downloads.html>
   1. 加环境变量：SPARK\_HOME，并把SPARK\_HOME\bin加入到PATH环境变量
   2. 下载：<https://github.com/SweetInk/hadoop-common-bin.git>，加环境变量：HADOOP\_HOME（如：D:\big\_data\hadoop-common-bin\2.7.3），并把HADOOP \_HOME\bin加入到PATH环境变量
   3. 现在就可以运行pyspark了，本目录下有一个用python写spark应用的代码模板：spark\_tpl.py
2. Spark中的parquet文件，为列式 存储格式，他包含表的schema信息，DataFrame可以直接保存为parquet文件，parquet文件也可以直接读取为DataFrame：
   1. peopleDF = spark.read.json("examples/src/main/resources/people.json")

peopleDF.write.parquet("people.parquet")

* 1. parquetFile = spark.read.parquet("people.parquet")

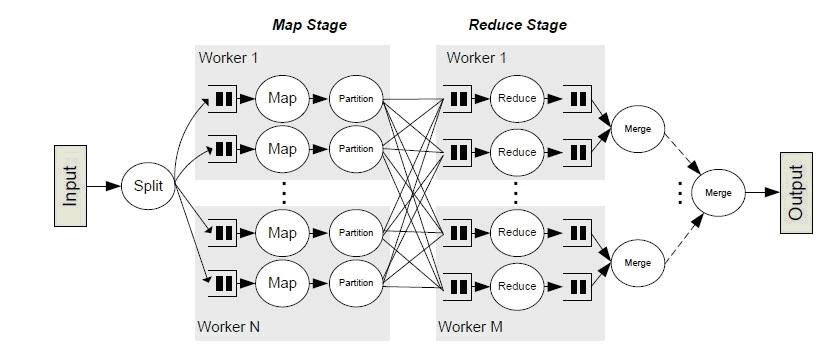
parquetFile.createOrReplaceTempView("parquetFile")

teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")

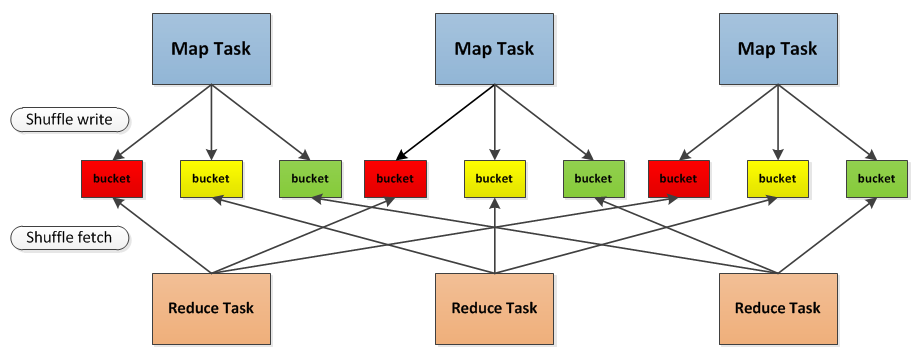
列式存储原理比较复杂，格式也比较复杂，目的是为了更快的定位并读取到所需要的数据，参考：<https://blog.csdn.net/yu616568/article/details/50993491>，理解它有助于写出更高性能的代码，如在使用Parquet的时候可以通过如下两种策略提升查询性能：1、类似于关系数据库的主键，对需要频繁过滤的列设置为有序的，这样在导入数据的时候会根据该列的顺序存储数据，这样可以最大化的利用最大值、最小值实现谓词下推。2、减小行组大小和页大小，这样增加跳过整个行组的可能性，但是此时需要权衡由于压缩和编码效率下降带来的I/O负载。

1. spark-shuffle（洗牌）解释：

下面这幅图清晰地描述了MapReduce算法的整个流程，其中shuffle phase是介于Map phase和Reduce phase之间：



Spark中shuffle的整一个流程：



首先每一个Mapper会根据Reducer的数量创建出相应的bucket，bucket的数量是M×R，其中MM是Map的个数，RR是Reduce的个数。

其次Mapper产生的结果会根据设置的partition算法填充到每个bucket中去。这里的partition算法是可以自定义的，当然默认的算法是根据key哈希到不同的bucket中去。

当Reducer启动时，它会根据自己task的id和所依赖的Mapper的id从远端或是本地的block manager中取得相应的bucket作为Reducer的输入进行处理。

这里的bucket是一个抽象概念，在实现中每个bucket可以对应一个文件，可以对应文件的一部分或是其他等。Apache Spark 的 Shuffle 过程与 Apache Hadoop 的 Shuffle 过程有着诸多类似，一些概念可直接套用，例如，Shuffle 过程中，提供数据的一端，被称作 Map 端，Map 端每个生成数据的任务称为 Mapper，对应的，接收数据的一端，被称作 Reduce 端，Reduce 端每个拉取数据的任务称为 Reducer，Shuffle 过程本质上都是将 Map 端获得的数据使用分区器进行划分，并将数据发送给对应的 Reducer 的过程。

1. parquet文件支持逐渐增加列，这样会得到多个parquet文件，这些文件的schema不相同但是是兼容的，使用代码可以控制进行合并：

sc = spark.sparkContext

squaresDF = spark.createDataFrame(sc.parallelize(range(1, 6))

.map(lambda i: Row(single=i, double=i \*\* 2)))

squaresDF.write.parquet("data/test\_table/key=1")

cubesDF = spark.createDataFrame(sc.parallelize(range(6, 11))

.map(lambda i: Row(single=i, triple=i \*\* 3)))

cubesDF.write.parquet("data/test\_table/key=2")

mergedDF=spark.read.option("mergeSchema","true").parquet("data/test\_table")

mergedDF.printSchema()

1. spark性能调优，如缓存表、使用SparkSession设置选项来提升性能。
2. 离散数据流（DStream）是 Spark Streaming 最基本的抽象，一个DStream内部是由一系列连续的RDD组成的，任何作 于 DStream 的 算子，其实都会被转化为对其内部 RDD 的 操作。
3. Spark stream性能调优：
   1. 减少每批次处理时间：
      1. 提升数据接收并发度：使用多个输入流Dstream；显示地将输入数据流划分为多个分区（使用 inputStream.repartition(<number of partitions>)）
      2. 提升数据处理并发度：在计算的各个阶段，较高的任务并发数可以提高系统资源的利用率
      3. 数据序列化：使用Kryo序列化来减少CPU和内存开销，另，对于Kryo，你可以考虑这些优化：注册自定义类型，禁用对象引用跟踪
      4. 任务启动开销
   2. 设置合理的批次间隔：处理数据的所消耗的时间应该要小于接收数据的间隔
   3. 内存调优
      1. DStream持久化级别（Persistence Level of DStreams）: 前面数据序列化（Data Serialization）这小节已经提到过，默认streaming的输入RDD会被持久化成序列化的字节流。相对于非序列化数据，这样可以减少内存占用和GC开销。如果启用Kryo序列化，还能进一步减少序列化数据大小和内存占用量。如果你还需要进一步减少内存占用的话，可以开启数据压缩（通过spark.rdd.compress这个配置设定），只不过数据压缩会增加CPU消耗。
      2. 清除老数据（Clearing old data）: 默认情况下，所有的输入数据以及DStream的transformation算子产生的持久化RDD都是自动清理的。Spark Streaming会根据所使用的transformation算子来清理老数据。例如，你用了一个窗口操作处理最近10分钟的数据，那么Spark Streaming会保留至少10分钟的数据，并且会主动把更早的数据都删掉。当然，你可以设置 streamingContext.remember 以保留更长时间段的数据（比如：你可能会需要交互式地查询更老的数据）。
      3. CMS垃圾回收器（CMS Garbage Collector）: 为了尽量减少GC暂停的时间，我们强烈建议使用CMS垃圾回收器（concurrent mark-and-sweep GC）。虽然CMS GC会稍微降低系统的总体吞吐量，但我们仍建议使用它，因为CMS GC能使批次处理的时间保持在一个比较恒定的水平上。最后，你需要确保在驱动器（通过spark-submit中的–driver-java-options设置）和执行器（使用spark.executor.extraJavaOptions配置参数）上都设置了CMS GC。
      4. 进一步减少GC开销：配合Tachyon使用堆外内存来持久化RDD；使用更多但是更小的执行器进程。这样GC压力就会分散到更多的JVM堆中。
4. Spark集群模式：Spark 应用程序在集群上作为独立的进程集合运行, 这些进程之间通过主程序（也被称为 Driver 程序）中的 SparkContext 对象来进行协调。具体来说，为了能够在集群上运行应用程序, SparkContext 要能连接到多种类型的集群管理器（Spark 自带的 Standalone 集群管理器, Mesos 或 YARN）, 这些集群管理器用于在应用程序之间分配资源。一旦连接上集群管理器，Spark 会在集群中的各个节点上为应用程序申请 Executor 用于执行计算任务和存储数据。接着，**Spark 将应用程序代码（传递给 SparkContext 的 JAR 包或者 Python 文件）发送给 Executor**。最后 SparkContext 将 Task 分发给各个 Executor 执行。
5. 使用spark-submit提交Spark任务时设置--deploy-mode参数：
   1. client：Driver进程在提交任务的机器节点上启动；如果该节点和worker集群位于同一网段就没问题；
   2. cluster：Driver进程在集群的worker节点里启动，如果提交任务的机器节点与worker集群所在的位置比较远（不在同一网段），采用这种模式就可以减少网络io延迟；这种模式还支持对driver进程的监控重启