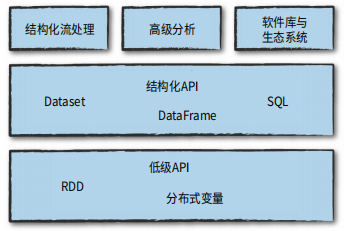
Apache Spark是一个在集群上运行的统一计算引擎以及一组并行数据处理软件库。

Spark是目前最流行的开源大数据处理引擎。

Spark支持多种常用的编程语言（Python，Java，Scala和R），

提供支持SQL、流处理、机器学习等多种任务的软件库，它既可以在笔记本计算机上

运行，也可以在数千台的服务器组成的集群上运行。



Spark工具集

Apache Spark是一个开源集群运算框架，相对于

Hadoop的MapReduce会在运行完工作后将中介数据存

放到磁盘中，Spark使用了存储器内运算技术，能在数

据尚未写入硬盘时即在存储器内分析运算。Spark 在存

储器内运行程序的运算速度能做到比 Hadoop MapReduce

的运算速度快上 100 倍，即便是运行程序于硬盘时，

Spark 也能快上 10 倍速度。

Spark 需要搭配集群管理员和分布式存储系统。

Spark 支持独立模式（本地 Spark 集群）

你可以使用Python，Java，Scala，R或SQL等语言来运行Spark。Spark本身是用Scala

编写的

启动Python控制台

./bin/pyspark

一个集群

一组计算机将许多机器的资

源集中在一起，使我们能够像使用单台计算机一样使用这些资源。但是如果一群机器

没有协调机制，那么这些机器并不能产生强大的计算能力，你需要一个软件框架来协

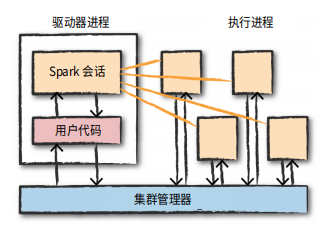
调它们之间的工作。Spark就是这样一种软件框架，它管理和协调跨多台计算机的计

算任务。

那么每一个机器的资源如何分配 比如硬盘空间、内存、线程 需要集群资源管理器，我们公司用的是yarn

Spark应用程序由一个驱动器进程和一组执行器进程组成。

图



驱动进程运行main()函数，

位于集群中的一个节点上，它负责三件事：维护Spark应用程序的相关信息；回应用

户的程序或输入；分析任务并分发给若干执行器进行处理。驱动器是必须的，它是

Spark应用程序的核心，它在应用程序执行的整个生命周期中维护着所有相关信息。

驱动器是必须的，它是

Spark应用程序的核心，它在应用程序执行的整个生命周期中维护着所有相关信息。

执行器负责执行驱动器分配给它的实际计算工作，这意味着每个执行器只负责两件

事：执行由驱动器分配给它的代码，并将该执行器的计算状态报告给运行驱动器的节

点。

用户可以配置指定每个节点上运行多少个执行器。

在本地模式下，

驱动器和执行器在个人计算机上运行（作为线程）而不是在集群上。

你可以通过名为SparkSession的驱动器来控制Spark应用程

序，你需要创建一个SparkSession实例来在群集中执行用户定义的操作，每一个Spark

应用程序都需要一个SparkSession与之对应。

DataFrame是最常见的结构化API，简单来说它是包含行和列的数据表。

说明这些列和

列类型的一些规则被称为模式（schema）。

Spark有几个核心抽象：Dataset，DataFrame，SQL表和弹性分布式数据集

（Resilient Distributed Datasets，RDD）。

为了让多个执行器并行地工作，Spark将数据分解成多个数据块，每个数据块叫做一

个分区。分区是位于集群中的一台物理机上的多行数据的集合，

DataFrame，你需要告诉Spark如何修改它以执行你想要的操作，这个过程被称为

转换。

注意这些转换并没有实际输出，这是因为我们仅指定了一个抽象转换。在我们调用一

个动作操作（我们将在本章后面详细介绍）之前，Spark不会真的执行转换操作。转

换操作是使用Spark表达业务逻辑的核心，有两类转换操作：第一类是指定窄依赖关

系的转换操作，第二类是指定宽依赖关系的转换操作。

具有窄依赖关系（narrow dependency）的转换操作（我们称之为窄转换）是每个输入

分区仅决定一个输出分区的转换。在前面的代码片段中，where语句指定了一个窄依

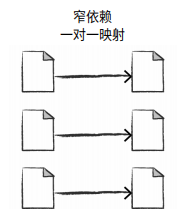
赖关系，其中一个分区最多只会对一个输出分区有影响，如图2-4所示。

具有宽依赖关系（wide dependency）的转换（或宽转换）是每个输入分区决定了多

个输出分区。这种宽依赖关系的转换经常被称为洗牌（shuffle）操作，它会在整个集

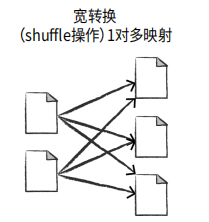
群中执行互相交换分区数据的功能。 如果是窄转换，Spark将自动执行流水线处理，

这意味着如果我们在DataFrame上指定了多个过滤操作，它们将全部在内存中执行。



而属于宽转换的shuffle操作不是这样，当我们执行shuffle操作时，Spark将结果写入磁

盘。图2-5中展示了宽转换操作。



一个指定动作操作的例子，我们启动一个Spark作业，首先执行过滤转换

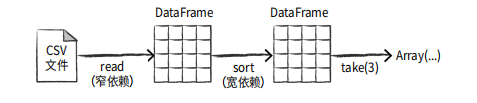
（一个窄转换），然后执行一个聚合操作（一个宽转换），再在每个分区上执行计数

count操作，然后通过collect操作将所有分区的结果汇集到一起，生成某个语言的一个

原生对象。你可以通过Spark UI看到所有这些操作，Spark UI是一个包含在Spark软件

包中的工具，你可以使用它监视Spark集群上运行的Spark作业。

最后执行count操作是动作操作才真正改变dataframe



sort操作不会修改DataFrame，因为sort是一个转换，它通过转换之

前的DataFrame来返回新的DataFrame。我们来看看当在结果DataFrame上执行

take操作时发生了什么（见图2-8）。

我们需要指定一个动作来触发这个计划的执行。在此之前，我们首先完成一个

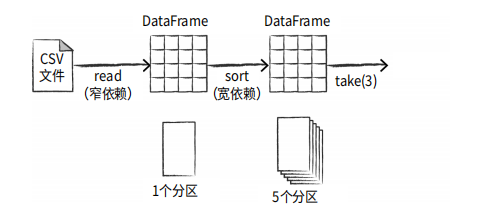
配置。默认情况下，shuffle操作会输出200个shuffle分区，我们将此值设置为5以减少

shuffle输出分区的数量：

spark.conf.set("spark.sql.shufflfle.partitions", "5")

flflightData2015.sort("count").take(2)

... Array([United States,Singapore,1], [Moldova,United States,1])



Spark-submit --master yarn/local xx.py

多次转换操作后建立

起指令的有向无环图。指令图的执行过程作为一个作业由一个动作操作触发，

在执行过程中一个作业被分解为多个阶段和任务在集群上执行。转换操作和动

作操作操纵的逻辑结构是DataFrame和Dataset，执行一次转换操作会都会创建

53一个新的DataFrame或Dataset，而动作操作则会触发计算，或者将DataFrame和

Dataset转换成本地语言类型

Spark包含六大"核心" 数据源

• CSV。

• JSON。

• Parquet。

• ORC。

• JDBC/ODBC连接。

• 纯文本文件。

Read API 结构

读取数据的核心结构如下：

DataFrameReader.format(...).option("key", "value").schema(...).load()

Write API 结构

DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).sa

ve()

默认都是parquet模式

保存模式

append

将输出文件追加到目标路径已存在的文件上或目录的文件列表

overwrite

将完全覆盖目标路径中已存在的任何数据

errorIfExists 如果目标路径已存在数据或文件，则抛出错误并返回写入操作失败

ignore

如果目标路径已存在数据或文件，则不执行任何操作

Parquet文件

Parquet是一种开源的面向列的数据存储格式，它提供了各种存储优化，尤其适合数据

分析。Parquet提供列压缩从而可以节省空间，而且它支持按列读取而非整个文件地读

取。作为一种文件格式，Parquet与Apache Spark配合得很好，而且实际上也是Spark

的默认文件格式。我们建议将数据写到Parquet以便长期存储，因为从Parquet文件读

取始终比从JSON文件或CSV文件效率更高。Parquet的另一个优点是它支持复杂类

型，也就是说如果列是一个数组（CSV文件无法存储数组列）、map映射或struct结构

体，仍可以正常读取和写入，不会出现任何问题。以下代码指定Parquet为文件读取格

式：

spark.read.format("parquet")

读Parquet文件

Parquet的可选项很少，因为它在存储数据时执行本身的schema，因此，你只需要指定

一下Parquet格式。如果我们对DataFrame的模式（schema）有严格的要求，则可以设

置schema。一般来说，在读取的时候使用默认的schema，所以就不需要再设置了，这

类似于CSV文件的infer Schema。然而由于schema内置于文件中，不需要推断，所以

Parquet文件格式更强大。

以下是一些从parquet中读取数据的简单例子：

spark.read.format(“parquet")

// in Scala

spark.read.format(“parquet")

.load(“/data/flflight-data/parquet/2010-summary.parquet").show(5)

# in Python

spark.read.format(“parquet")\

.load(“/data/flflight-data/parquet/2010-summary.parquet").show(5)

Parquet可选项

由于Parquet含有明确定义且与Spark概念密切一致的规范，所以它只有很少的可选

项，实际上只有两个，表9-5列出了可选项。

虽然只有两个选项，如果你使用的是不兼容的Parquet文件，仍然会遇到问题。

当使用不同版本的Spark（尤其是旧版本时）写入Parquet文件时要小心，因为

这可能会导致让人头疼的问题。

表9-5：Parquet数据源选项

read/

write Key

取值范围

默认值

说明

write compression None, uncom None

声明Spark应该使用什

或codec pressed, bzip2, deflate,

么压缩编解码器来读取

gzip, lz4,或 snappy

或写入文件

read merge Schema true，false

配置值spark. 增量地添加列到同一表

sql.par quet. /文件夹中的Parquet文

mergeSchema 件里，此选项用于启用

或禁用此功能

写Parquet文件

写Parquet文件和读取它一样简单，只需指定文件的位置即可。这里应用相同的分片规

则：

// in Scala

csvFile.write.format(“parquet").mode(“overwrite")

.save(“/tmp/my-parquet-fifile.parquet")

# in Python

csvFile.write.format(“parquet").mode(“overwrite")\

.save(“/tmp/my-parquet-fifile.parquet")

高级I/O概念

我们可以通过在写入之前控制数据分片来控制写入文件的并行度，还可以通过控制数

据分桶（bucketing）和数据划分（partitioning）来控制特定的数据布局方式。

可分割的文件类型和压缩

某些文件格式是“可分割的”，因此Spark可以只获取该文件中的满足查询条件的某

一个部分，无需读取整个文件，从而提高读取效率。此外，假设你使用的是 Hadoop

分布式文件系统（HDFS），则如果该文件包含多个文件块，分割文件则可进一步优

化提高性能。与此同时需要进行压缩管理，并非所有的压缩格式都是可分割的。存储

数据的方式对Spark作业稳定运行至关重要，我们推荐采用gzip压缩格式的Parquet文

件格式。

并行读数据

多个执行器不能同时读取同一文件，但可以同时读取不同的文件。通常，这意味着当

你从包含多个文件的文件夹中读取时，每个文件都将被视为DataFrame的一个分片，

并由执行器并行读取，多余的文件会进入读取队列等候。

并行写数据

写数据涉及的文件数量取决于DataFrame的分区数。默认情况是每个数据分片都会有

一定的数据写入，这意味着虽然我们指定的是一个“文件”，但实际上它是由一个文

件夹中的多个文件组成，每个文件对应着一个数据分片。

以下是代码示例：

csvFile.repartition(5).write.format("csv").save("/tmp/multiple.csv")

它会生成包含五个文件的文件夹，调用ls命令就可以查看到：

ls /tmp/multiple.csv

/tmp/multiple.csv/part-00000-767df509-ec97-4740-8e15-4e173d365a8b.csv

/tmp/multiple.csv/part-00001-767df509-ec97-4740-8e15-4e173d365a8b.csv

/tmp/multiple.csv/part-00002-767df509-ec97-4740-8e15-4e173d365a8b.csv

/tmp/multiple.csv/part-00003-767df509-ec97-4740-8e15-4e173d365a8b.csv

/tmp/multiple.csv/part-00004-767df509-ec97-4740-8e15-4e173d365a8b.csv

数据划分

数据划分工具支持你在写入数据时控制存储什么数据以及存储数据的位置。将文件写

出时，你可以将列编码为文件夹，这使得你在之后读取时可跳过大量数据，只读入与

问题相关的列数据而不必扫描整个数据集。所有基于文件的数据源都支持这些：

// in Scala

csvFile.limit(10).write.mode(“overwrite").partitionBy(“DEST\_COUNTRY\_NAME")

.save(“/tmp/partitioned-fifiles.parquet")

# in Python

csvFile.limit(10).write.mode(“overwrite").partitionBy(“DEST\_COUNTRY\_NAME")\

.save(“/tmp/partitioned-fifiles.parquet")

写操作完成后，Parquet“文件”中就会有一个文件夹列表：

$ ls /tmp/partitioned-fifiles.parquet

...

DEST\_COUNTRY\_NAME=Costa Rica/

DEST\_COUNTRY\_NAME=Egypt/

DEST\_COUNTRY\_NAME=Equatorial Guinea/

DEST\_COUNTRY\_NAME=Senegal/

DEST\_COUNTRY\_NAME=United States/

其中每一个都将包含Parquet文件，这些文件包含文件夹名称中谓词为 true的数据：

$ ls /tmp/partitioned-fifiles.parquet/DEST\_COUNTRY\_NAME=Senegal/

part-00000-tid.....parquet

读取程序对某表执行操作之前经常执行过滤操作，这时数据划分就是最简单的优化。

例如，基于日期来划分数据最常见，因为通常我们只想查看前一周的数据（而不是扫

描所有日期数据），这个优化可以极大提升读取程序的速度。

数据分桶

数据分桶是另一种文件组织方法，你可以使用该方法控制写入每个文件的数据。具有

相同桶 ID （哈希分桶的ID）的数据将放置到一个物理分区中，这样就可以避免在稍

后读取数据时进行shuffle（洗牌）。根据你之后希望如何使用该数据来对数据进行预

分区，就可以避免连接或聚合操作时执行代价很大的shuffle操作。

与其根据某列进行数据划分，不如考虑对数据进行分桶，因为某列如果存在很多不同

的值，就可能写出一大堆目录。这将创建一定数量的文件，数据也可以按照要求组织

起来放置到这些“桶”中：

val numberBuckets = 10

val columnToBucketBy = “count"

csvFile.write.format(“parquet").mode(“overwrite")

.bucketBy(numberBuckets, columnToBucketBy).saveAsTable(“bucketedFiles")

$ ls /user/hive/warehouse/bucketedfifiles/

part-00000-tid-1020575097626332666-8....parquet

part-00000-tid-1020575097626332666-8....parquet

part-00000-tid-1020575097626332666-8....parquet

...

数据分桶仅支持Spark管理的表。

管理文件大小

管理文件大小对数据写入不那么重要，但对之后的读取很重要。当你写入大量的小

文件时，由于管理所有的这些小文件而产生很大的元数据开销。许多文件系统（如

HDFS）都不能很好地处理大量的小文件，而Spark特别不适合处理小文件。你可能听

说过“小文件问题”，反之亦然，你也不希望文件太大，因为当你只需要其中几行

时，必须读取整个数据块就会使效率低下。

Spark 2.2中引入了一种更自动控制文件大小的新方法。之前介绍了输出文件数量，

与写入时数据分片数量以及选取的划分列有关。现在，则可以利用另一个工具来

限制输出文件大小，从而可以选出最优的文件大小。可以使用maxRecordsPerFile

选项来指定每个文件的最大记录数，这使得你可以通过控制写入每个文件的记

录数来控制文件大小。例如，如果你将写程序（writer）的选项设置为df.write.

option(“maxRecordsPerFile”，5000），Spark将确保每个文件最多包含5000条记

录。

spark.sql.shuffle.partitions 200

配置在为连接或聚合shuffle数据时要使用的

分区数

repartition

Repartition操作将对数据进行重新分区，跨节点的分区会执行shuffle操作。对于map

和filter操作，增加分区可以提高并行度：

words.repartition(10) // 10个分区

repartitionAndSortWithinPartitions

此操作将对数据重新分区，并指定每个输出分区的顺序。

Spark驱动器

Spark驱动器是控制你应用程 序的进程。它负责控制整个Spark应用程序的执行并

且维护着Spark集群的状态，即执行器的任务和状态，它必须与集群管理器交互

才能获得物理资源并启动执行器。简而言之，它只是一个物理机器上的一个进

程，负责维护群集上运行的应用程序的状态。

Spark执行器

Spark执行器也是一个进程，它负责执行由Spark驱动器分配的任务。执行器的核

心功能是：完成驱动器分配的任务，运行它们，并报告其状态（成功或失败）和

执行结果。每个Spark应用程序都有自己的执行器进程。

集群管理器

Spark驱动器和执行器并不是孤立存在的，集群管理器会将他们联系起来，集群

管理器负责维护一组运行Spark应用程序的机器。集群管理器也拥有自己的“驱

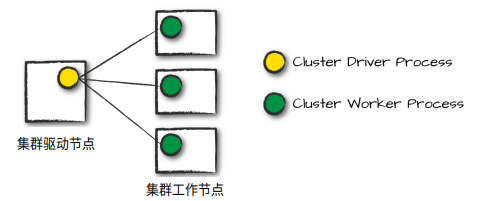
动器”（即master）和worker的抽象，核心区别在于集群管理器管理的是物理机

器，而不是进程。 图15-1展示了一个基本的集群配置，图左侧的机器是集群管

理器的驱动节点（即master），圆圈表示在每个物理节点上运行的进程，它们负

责管理每个物理节点。因为目前还没有运行Spark应用程序，所以这些进程只是

集群管理器的进程，并不是运行Spark应用程序的驱动器和执行器。

当实际运行Spark应用程序时，我们会从集群管理器那里请求资源来运行它。根据应

用程序的配置，我们可能获得一个运行Spark驱动器的机器资源，或者可能获得的是

我们Spark执行器的计算资源。在Spark应用程序执行过程中，集群管理器将负责管理

和运行执行应用程序的底层机器。

Spark目前支持三个集群管理器：一个简单的内置独立集群管理器，Apache Mesos和

Hadoop YARN。

执行模式

当在运行应用程序之前，通过选择执行模式你将能够确定计算资源的物理位置。你有

三种模式可供选择：

• 集群模式。

• 客户端模式。

• 本地模式。

我们将使用图15-1作为示例来详细介绍其中的每一个细节。在下面的章节中，实线框

矩形表示Spark驱动器，而虚线框句型则表示执行器。

集群模式

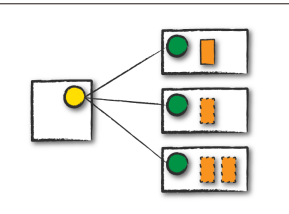
集群模式可能是运行Spark应用程序的最常见方式。在集群模式下，用户将预编译的

JAR包，Python脚本或R语言脚本提交给集群管理器。除执行器进程外，集群管理器

还会在集群内的某个工作节点上启动驱动器进程，这意味着集群管理器负责维护所有

与Spark应用程序相关的进程。在图15-2展示的情况中，集群管理器将Spark驱动器放

置在一个工作节点上，并将Spark执行器放在其他工作节点上。



客户端模式

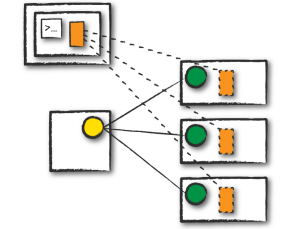
客户端模式与集群模式几乎相同，只是Spark驱动器保留在提交应用程序的客户端机

器上。这意味着客户端机器负责维护Spark驱动器进程，并且集群管理器维护执行器

进程。在图15-3中，我们使用一台集群外的机器上运行Spark应用程序，这些机器通

常被称为网关机器（gateway machines）或边缘节点（edge nodes）。在图15-3中，你

可以看到驱动器正在集群外部的计算机上运行，但工作节点位于集群中的 计算机上。

本地模式

本地模式与前两种模式有很大不同：它在一台机器上运行整个Spark应用程序。它通

过单机上的线程实现并行性。在本地模式上运行Spark是学习Spark的常用方法，也是

测试应用程序或进行迭代本地开发的常用方法，但是我们不建议使用本地模式运行生

产级别的应用程序。

Spark应用程序的生命周期（Spark外部）P250

--SparkSession P254

Spark作业 P256

我们将在第19章中详细介绍分区的数量，因为它是一个非常重要的参数。它应

该根据集群中的核的数量设置，以确保高效执行。以下是设置它的方法：

spark.conf.set("spark.sql.shufflfle.partitions"，50)

--P259

./bin/spark-submit \

--class <main-class> \

--master <master-url> \

--deploy-mode <deploy-mode> \

--conf <key>=<value> \

... # other options

<application-jar-or-script> \

[application-arguments]

--master MASTER\_URL 指定master节点URL，例如spark：//host：port，mesos：//

host：port，yarn，or local

--deploy-mode DEPLOY 配置是在本地以客户端模式 (“client”) 还是在一台集群中节

\_MODE

点上以集群模式(“cluster”)运行应用程序 (默认使用客户端模

式)

--py-files PY\_FILES

配置Python应用程序需要的.zip、.egg或者.py文件（即放在

PYTHONPATH路径上的文件），用逗号隔开

--driver-memory MEM 配置驱动器的内存大小（例如，1000MB，2GB）（默认：

1024MB）

--executor-memory MEM 配置执行器的内存大小（例如，1000MB，2GB）（默认：

1024MB）

Standalone Cluster --driver-cores NUM

驱动器的核心数量（默认：1）

Standalone/Mesos Either --total-executor-cores NUM 所有执行器的总核心数

Standalone/YARN Either --executor-cores NUM1

每个执行器的核心数(默认：

YARN 模式下为 1 ，或在

standalone模式下worker节点的

所有可用核心数)

YARN

Either --driver-cores NUM

集群模式下的驱动器的核心数

（默认：1）

YARN

Either --num-executors NUM

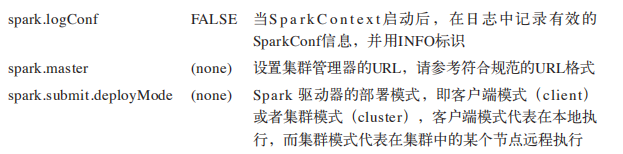
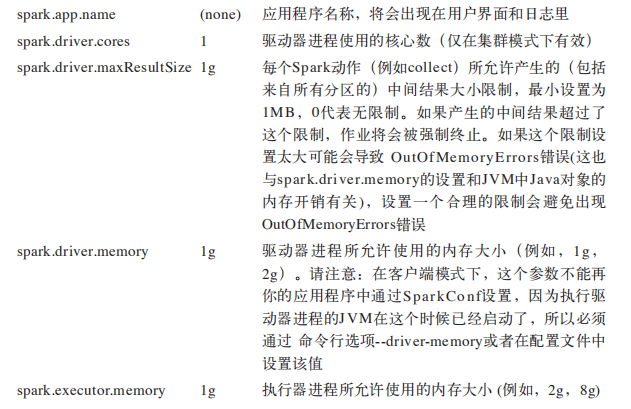
启动的执行器（默认：2）。如

果配置了动态分配，那么初始

的执行器数量最少为NUM

以本地模式运行，即将master设置为local或local[\*]（在计算机

的所有核心上运行），

spark.executor.cores（用于控制使用

的核心数）和spark.fifiles.maxPartitionBytes（读取输入文件时的最大分块大小）。

有两种部署模式可用于在YARN上启动Spark。正如前面的章节所讨论的，集群

模式将spark驱动器作为由YARN集群管理的进程，客户端在创建应用程序后退

出。在客户端模式下，驱动器将运行在客户端进程中，因此YARN只负责将执

行器的资源授予应用程序，而不是维护master节点。另外值得注意的是，在集

群模式下，Spark不一定在你正在执行spark-submit命令的同一台机器上运行。

因此，库和外部jar必须手动配置或通过--jars命令行参数配置。

动态分配

如果在同一个群集上同时运行多个Spark应用程序，Spark提供了一种可以根据工作负

载动态调整应用程序占用的资源的机制。也就是说，在应用程序不再使用资源时将资

源返回给集群，并在有资源需要时再次请求使用。如果多个应用程序共享Spark集群

中的资源，资源的动态分配就尤为重要。

此功能在默认情况下处于禁用状态，但是在粗粒度集群管理器上都支持，即

Standalone模式，YARN模式和Mesos粗粒度模式。使用此功能有两个要求：首先，你

的应用程序必须将spark.dynamicAllocation.enabled属性设置为true；其次，你要

在每个工作节点上设置外部Shuffle服务，并在应用程序中将spark.shuffle.service.

enabled属性设置为true。外部shuffle服务的目的是为了允许终止执行进程而不删除由

它们输出的shuffle文件。每个集群管理器对设置动态分配功能的方法都不同，这在配

置作业调度部分进行了描述。由于篇幅有限，本书不详细介绍动态分配的配置参数，

请参阅动态分配配置的相关表格。

监控与调试

监控级别

1Spark应用程序和作业

Spark UI和

Spark日志

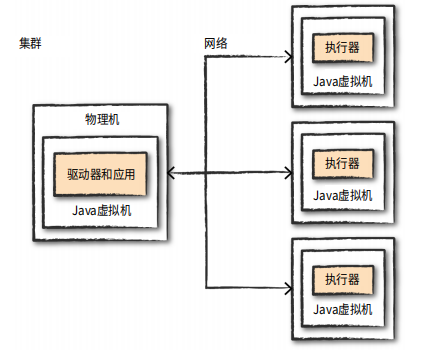
2.JVM

3.操作系统/主机

JVM运行在主机操作系统（OS）上，监视这些机器的运行状态也很重要，这包

括监控诸如CPU、网络、I/O等。

4.集群



要监控的主要有两个方面：运行应用程序的进程信息（CPU使用率，内存使用率

等）以及查询执行过程（作业和任务）。

应用程序的所有状态都

会在驱动器进程上有所反映，你需要确保它正确而稳定的运行。如果你只能监控一台

机器或一台JVM，那首选就是驱动器节点。



百度

spark.dynamicAllocation.enabled

设置为true

并行度

如果需要加速某一个特定阶段，你应该做的第一件事就是增加并行度。通常，如果一

个阶段需要处理大量的数据，我们建议你分配集群中每个CPU核心至少有两到三个任

务。你可以通过spark.default.parallelism属性来设置它，并同时根据集群中的核心

数量来调整spark.sql.shufflfle.partitions属性。

Parquet文件

Parquet是一种开源的面向列的数据存储格式，它提供了各种存储优化，尤其适合数据

分析。Parquet提供列压缩从而可以节省空间，而且它支持按列读取而非整个文件地读

取。作为一种文件格式，Parquet与Apache Spark配合得很好，而且实际上也是Spark

的默认文件格式。我们建议将数据写到Parquet以便长期存储，因为从Parquet文件读

取始终比从JSON文件或CSV文件效率更高。Parquet的另一个优点是它支持复杂类

型，也就是说如果列是一个数组（CSV文件无法存储数组列）、map映射或struct结构

体，仍可以正常读取和写入，不会出现任何问题。以下代码指定Parquet为文件读取格

式：

spark.read.format("parquet")

读Parquet文件

Parquet的可选项很少，因为它在存储数据时执行本身的schema，因此，你只需要指定

一下Parquet格式。如果我们对DataFrame的模式（schema）有严格的要求，则可以设

置schema。一般来说，在读取的时候使用默认的schema，所以就不需要再设置了，这

类似于CSV文件的infer Schema。然而由于schema内置于文件中，不需要推断，所以

Parquet文件格式更强大。

以下是一些从parquet中读取数据的简单例子：

spark.read.format(“parquet")

// in Scala

spark.read.format(“parquet")

.load(“/data/flflight-data/parquet/2010-summary.parquet").show(5)

# in Python

spark.read.format(“parquet")\

.load(“/data/flflight-data/parquet/2010-summary.parquet").show(5)

Parquet可选项

由于Parquet含有明确定义且与Spark概念密切一致的规范，所以它只有很少的可选

项，实际上只有两个，表9-5列出了可选项。

虽然只有两个选项，如果你使用的是不兼容的Parquet文件，仍然会遇到问题。

当使用不同版本的Spark（尤其是旧版本时）写入Parquet文件时要小心，因为

这可能会导致让人头疼的问题。

表9-5：Parquet数据源选项

read/

write Key

取值范围

默认值

说明

write compression None, uncom None

声明Spark应该使用什

或codec pressed, bzip2, deflate,

么压缩编解码器来读取

gzip, lz4,或 snappy

或写入文件

read merge Schema true，false

配置值spark. 增量地添加列到同一表

sql.par quet. /文件夹中的Parquet文

mergeSchema 件里，此选项用于启用

或禁用此功能

写Parquet文件

写Parquet文件和读取它一样简单，只需指定文件的位置即可。这里应用相同的分片规

则：

// in Scala

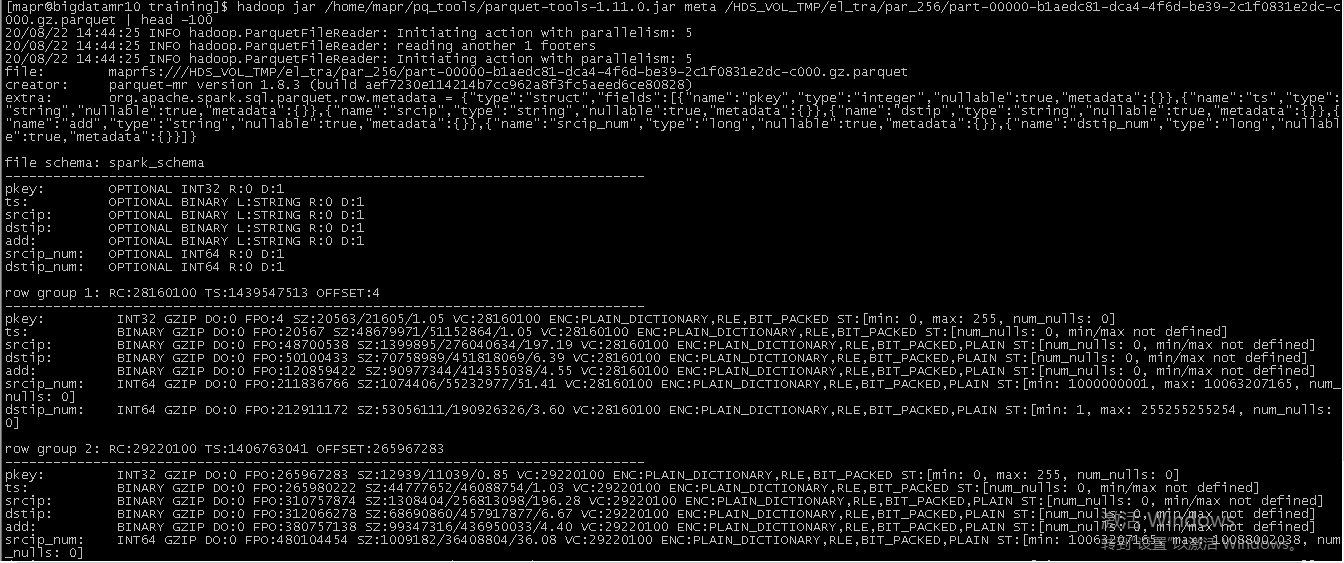
csvFile.write.format(“parquet").mode(“overwrite")

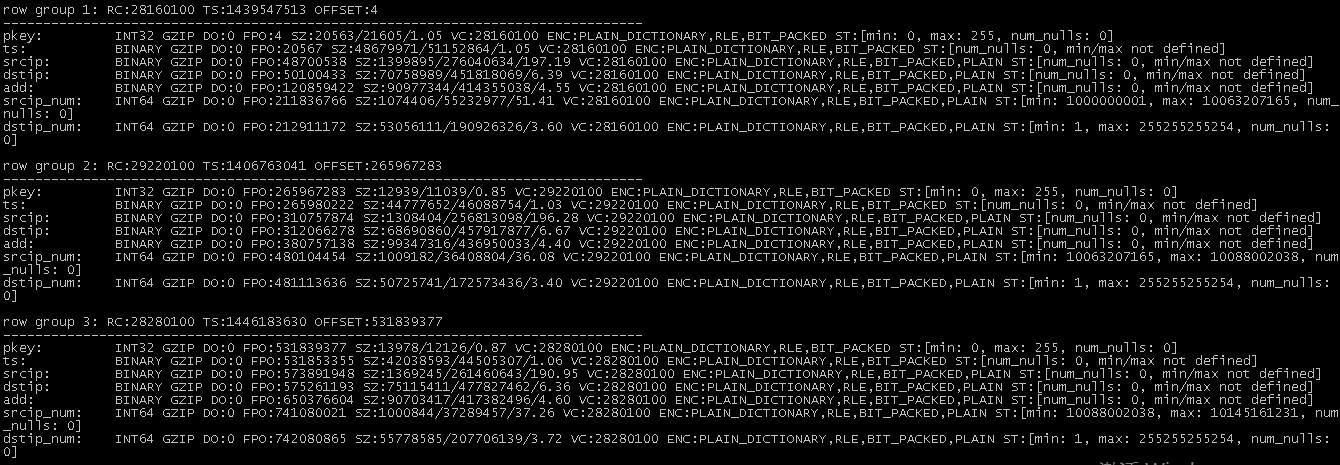
.save(“/tmp/my-parquet-fifile.parquet")

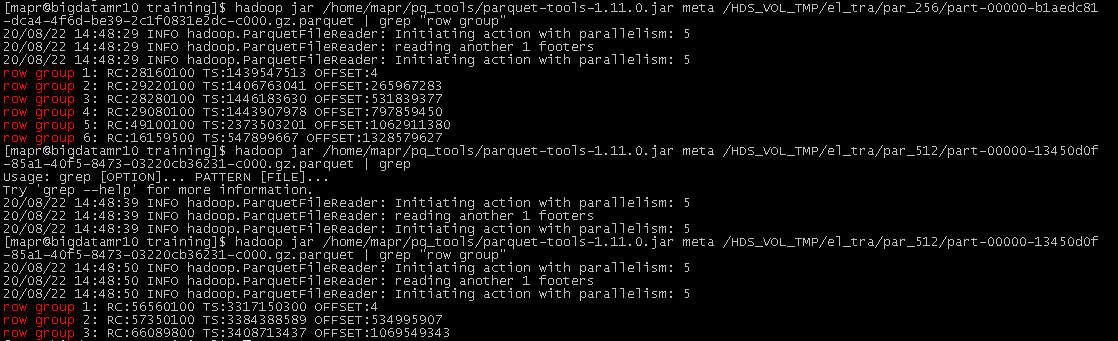
# in Python

csvFile.write.format(“parquet").mode(“overwrite")\

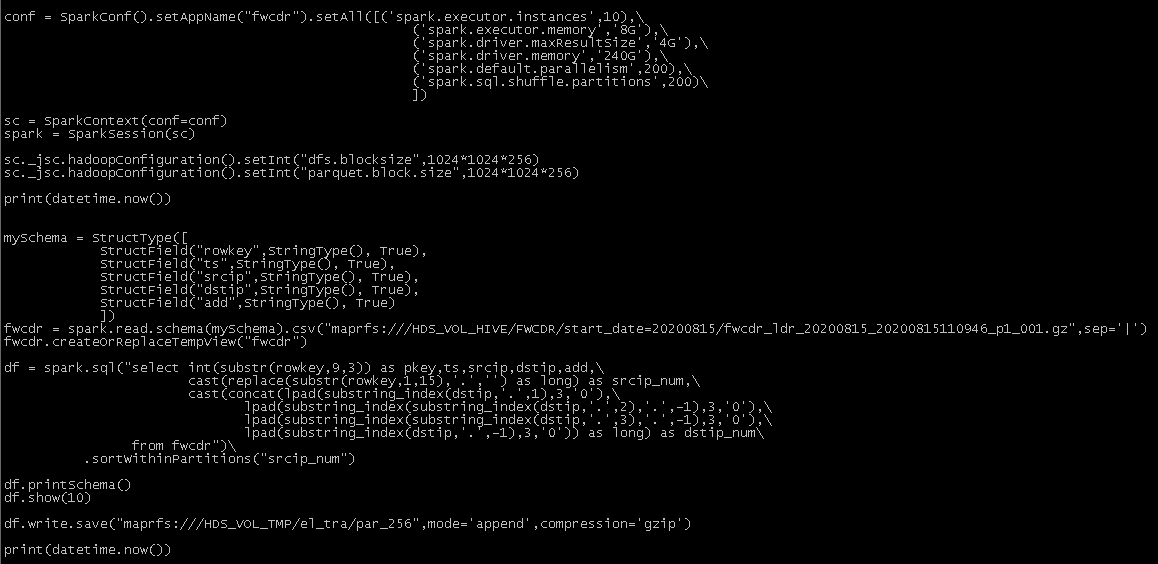
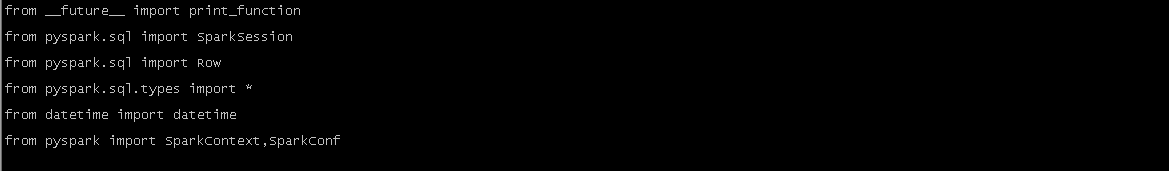
.save(“/tmp/my-parquet-fifile.parquet")

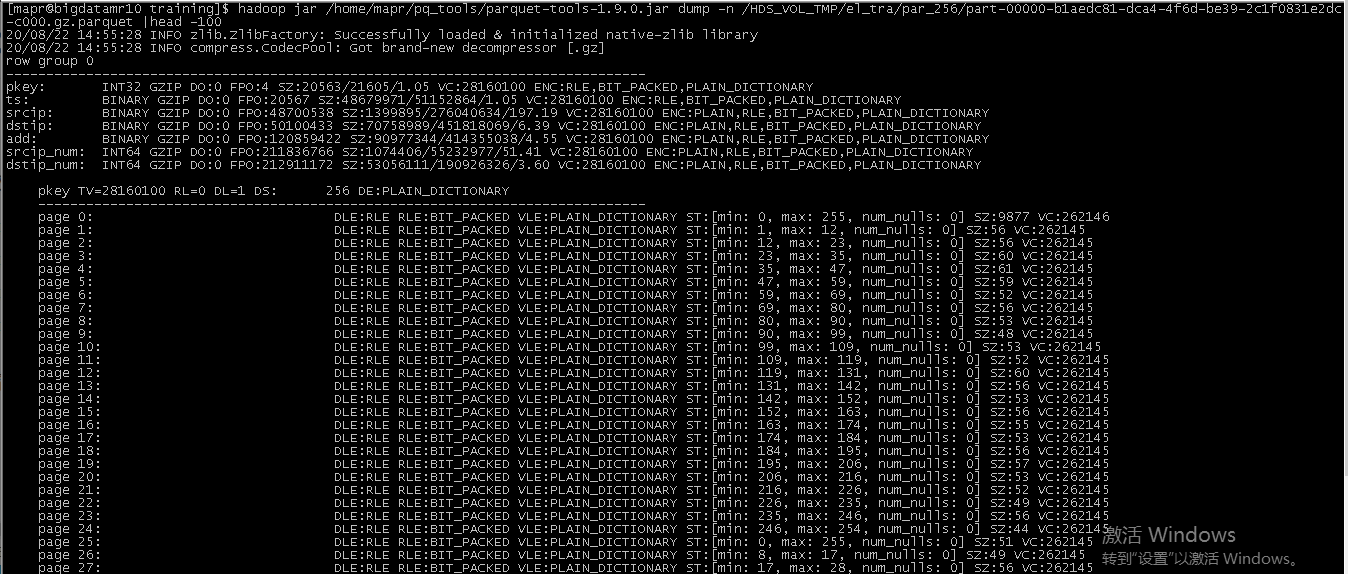




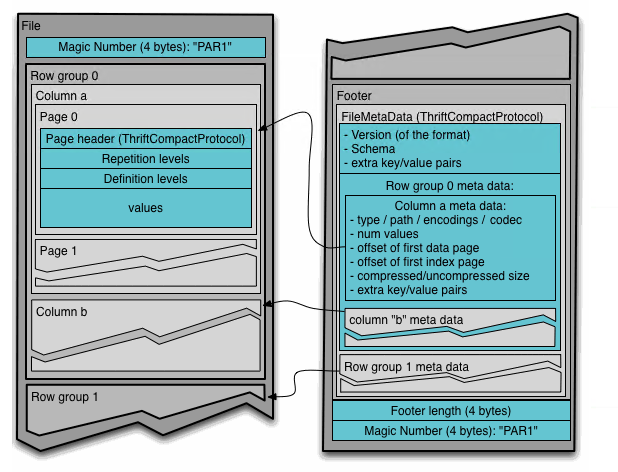


Parquet 内部的数据类型、存储格式





apacheparquet是一种列式数据存储格式，它提供了一种按列存储表格数据的方法。相同日期时间的列以Parquet格式作为行存储在一起，以便提供更好的存储、压缩和数据检索。



## Metadata

元数据有三种类型： file metadata, column (chunk) metadata and page header metadata。所有协议都使用TCompactProtocol序列化。

## Types

BOOLEAN: 1 bit boolean

INT32: 32 bit signed ints

INT64: 64 bit signed ints

INT96: 96 bit signed ints

FLOAT: IEEE 32-bit floating point values

DOUBLE: IEEE 64-bit floating point values

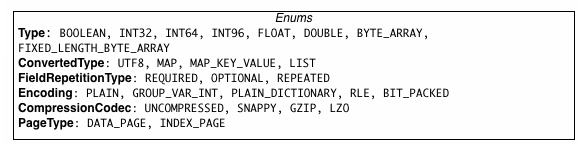
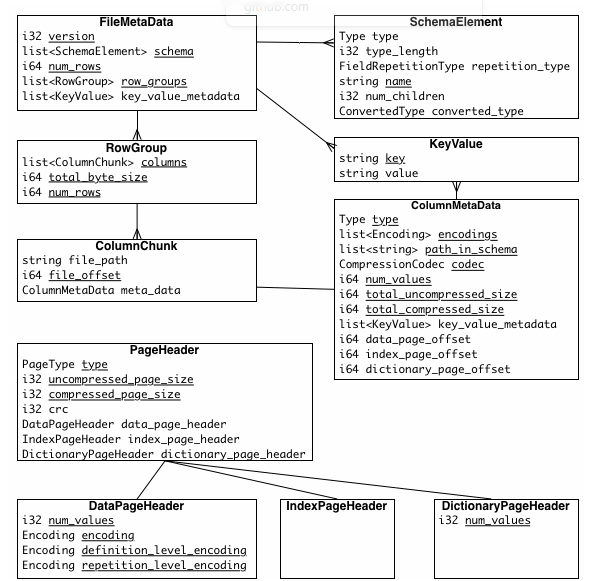
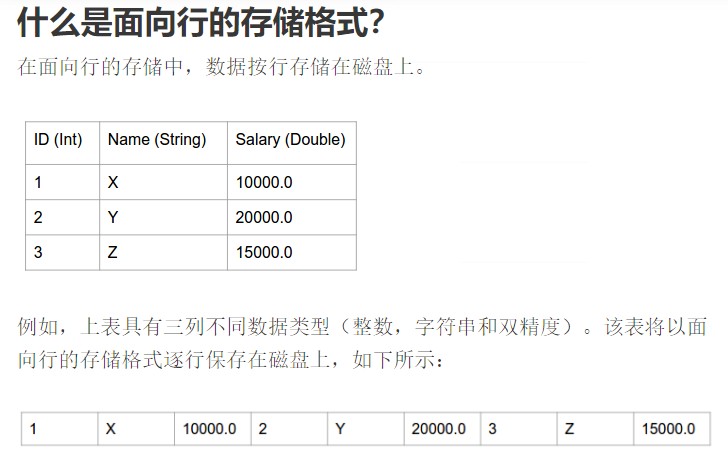
BYTE\_ARRAY: arbitrarily long byte arrays.

## Nested Encoding

Parquet uses the Dremel encoding with definition and repetition levels. Definition levels specify how many optional fields in the path for the column are defined. Repetition levels specify at what repeated field in the path has the value repeated. The max definition and repetition levels can be computed from the schema (i.e. how much nesting there is).

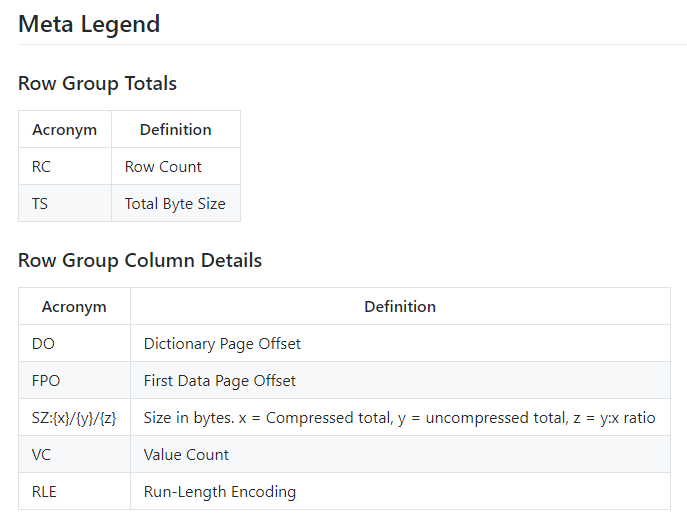
Parquet使用具有定义和重复级别的Dremel编码。定义级别指定在列的路径中定义了多少个可选字段。重复级别指定路径中哪个重复字段的值重复。最大定义和重复级别可以从模式中计算（即有多少嵌套）。

较大的行组允许更大的列块，从而可以执行更大的顺序IO。较大的组也需要在写入路径（或两次写入）中有更多的缓冲区。我们建议大行组（512MB-1GB）。因为可能需要读取整个行组，所以我们希望它完全适合一个HDFS块。因此，HDFS块大小也应设置为更大。优化的读取设置是：1GB行组、1GB HDFS块大小、每个HDFS文件1个HDFS块。







Head cat

Block（hdfs Block）：这意味着hdfs中的一个块，其含义对于描述这个文件格式没有改变。该文件格式设计用于在hdfs之上很好地工作。

file：必须包含文件元数据的hdfs文件。它不需要实际包含数据。

Row group：将数据分为行的逻辑水平分区。对于行组没有保证的物理结构。行组由数据集中每个列的列块组成。

Column chunk：特定列的数据块。它们位于特定的行组中，并保证在文件中是连续的。

page：列块被分成多个页面。页面在概念上是一个不可分割的单元（就压缩和编码而言）。在一个列块中可以有多个页面类型交错。

在层次结构上，文件由一个或多个行组组成。一个行组每列只包含一个列块。列块包含一个或多个页面。

列式存储布局（比如 Parquet）可以加速查询，因为它只检查所有需要的列并对它们的值执行计算，因此只读取一个数据文件或表的小部分数据。Parquet 还支持灵活的压缩选项，因此可以显著减少磁盘上的存储。

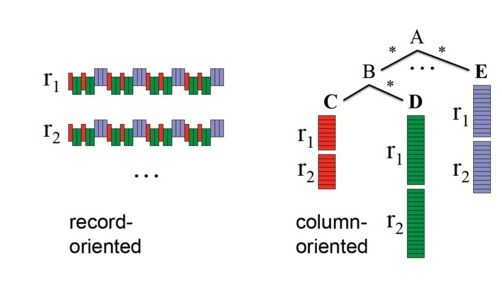
如果您在 HDFS 上拥有基于文本的数据文件或表，而且正在使用 Spark SQL 对它们执行查询，那么强烈推荐将文本数据文件转换为 Parquet 数据文件，以实现性能和存储收益。当然，转换需要时间，但查询性能的提升在某些情况下可能达到 30 倍或更高，存储的节省可高达 75%！

查询性能

Parquet 文件是自描述性的，所以保留了模式

1. 可以跳过不符合条件的数据，只读取需要的数据，降低 IO 数据量。
2. 压缩编码可以降低磁盘存储空间。由于同一列的数据类型是一样的，可以使用更高效的压缩编码（例如 Run Length Encoding 和 Delta Encoding）进一步节约存储空间。
3. 只读取需要的列，支持向量运算，能够获取更好的扫描性能。

关系型数据的列式存储比较好理解，而嵌套类型数据的列存储则会遇到一些麻烦。如图 1 所示，我们把嵌套数据类型的一行叫做一个记录（record)，嵌套数据类型的特点是一个 record 中的 column 除了可以是 Int, Long, String 这样的原语（primitive）类型以外，还可以是 List, Map, Set 这样的复杂类型。在行式存储中一行的多列是连续的写在一起的，在列式存储中数据按列分开存储，例如可以只读取 A.B.C 这一列的数据而不去读 A.E 和 A.B.D，那么如何根据读取出来的各个列的数据重构出一行记录呢？





record shredding and assembly algorithm

记录分解与组装算法

按列的高效压缩和编码技术，实现降低存储空间，提高 IO 效率

## Parquet 适配多种计算框架

Parquet 是语言无关的，而且不与任何一种数据处理框架绑定在一起，适配多种语言和组件，能够与 Parquet 配合的组件有：

查询引擎: Hive, Impala, Pig, Presto, Drill, Tajo, HAWQ, IBM Big SQL

计算框架: MapReduce, Spark, Cascading, Crunch, Scalding, Kite

数据模型: Avro, Thrift, Protocol Buffers, POJOs

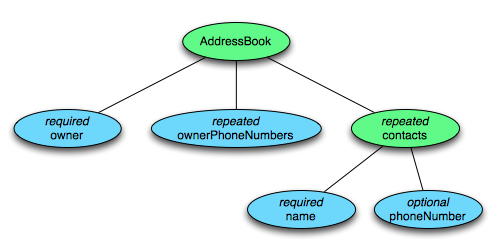
|  |
| --- |
| message AddressBook { |
|  | required string owner; |
|  | repeated string ownerPhoneNumbers; |
|  | repeated group contacts { |
|  | required string name; |
|  | optional string phoneNumber; |
|  | } |
|  | } |



这个 schema 中每条记录表示一个人的 AddressBook。有且只有一个 owner，owner 可以有 0 个或者多个 ownerPhoneNumbers，owner 可以有 0 个或者多个 contacts。每个 contact 有且只有一个 name，这个 contact 的 phoneNumber 可有可无。这个 schema 可以用图 3 的树结构来表示。

每个 schema 的结构是这样的：根叫做 message，message 包含多个 fields。每个 field 包含三个属性：repetition, type, name。repetition 可以是以下三种：required（出现 1 次），optional（出现 0 次或者 1 次），repeated（出现 0 次或者多次）。type 可以是一个 group 或者一个 primitive 类型。

Parquet 格式的数据类型没有复杂的 Map, List, Set 等，而是使用 repeated fields 和 groups 来表示。例如 List 和 Set 可以被表示成一个 repeated field，Map 可以表示成一个包含有 key-value 对的 repeated field，而且 key 是 required 的。

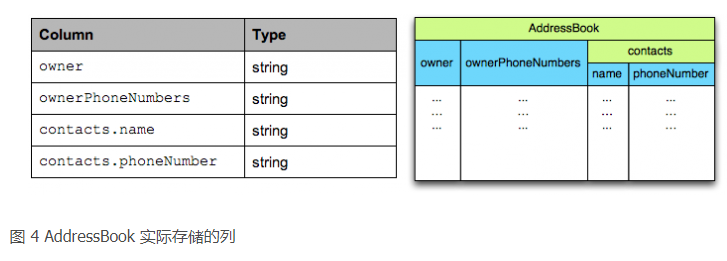




## Parquet 文件的存储格式

那么如何把内存中每个 AddressBook 对象按照列式存储格式存储下来呢？

在 Parquet 格式的存储中，一个 schema 的树结构有几个叶子节点，实际的存储中就会有多少 column。例如上面这个 schema 的数据存储实际上有四个 column



Parquet 文件在磁盘上的分布情况如图 5 所示。所有的数据被水平切分成 Row group，一个 Row group 包含这个 Row group 对应的区间内的所有列的 column chunk。一个 column chunk 负责存储某一列的数据，这些数据是这一列的 Repetition levels, Definition levels 和 values（详见后文）。一个 column chunk 是由 Page 组成的，Page 是压缩和编码的单元，对数据模型来说是透明的。一个 Parquet 文件最后是 Footer，存储了文件的元数据信息和统计信息。Row group 是数据读写时候的缓存单元，所以推荐设置较大的 Row group 从而带来较大的并行度，当然也需要较大的内存空间作为代价。一般情况下推荐配置一个 Row group 大小 1G，一个 HDFS 块大小 1G，一个 HDFS 文件只含有一个块。

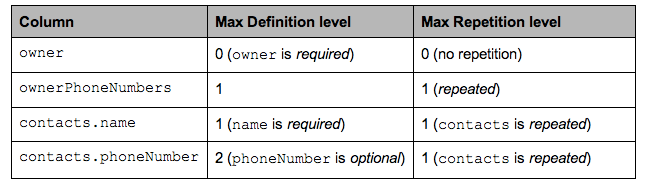
Repetition Level 和 Definition Level 可以说是为了支持嵌套类型而设计的，但是它同样适用于简单数据类型。在 Parquet 中我们只需定义和存储 schema 的叶子节点所在列的 Repetition Level 和 Definition Level。

### Definition Level

嵌套数据类型的特点是有些 field 可以是空的，也就是没有定义。如果一个 field 是定义的，那么它的所有的父节点都是被定义的。从根节点开始遍历，当某一个 field 的路径上的节点开始是空的时候我们记录下当前的深度作为这个 field 的 Definition Level。如果一个 field 的 Definition Level 等于这个 field 的最大 Definition Level 就说明这个 field 是有数据的。对于 required 类型的 field 必须是有定义的，所以这个 Definition Level 是不需要的。在关系型数据中，optional 类型的 field 被编码成 0 表示空和 1 表示非空（或者反之）。

### Repetition Level

记录该 field 的值是在哪一个深度上重复的。只有 repeated 类型的 field 需要 Repetition Level，optional 和 required 类型的不需要。Repetition Level = 0 表示开始一个新的 record。在关系型数据中，repetion level 总是 0。



列裁剪与谓词下推：列裁剪，意思是只读取需要的列，实现高效的列扫描，减少 IO 操作；谓词下推，可以过滤掉不符合条件的数据，只读取需要的数据，进一步减少 IO 操作。

更高效的压缩与编码：因为同一列的数据类型相同，所以可以针对不同列使用更合适的压缩与编码方式，降低磁盘存储空间。

值得说明的是文件 header 部分中的 Magic Number，它的作用主要是为了做文件校验，验证文件是否是一个Parquet文件。

****1、通过 Hive 创建 Parquet 表****

****·**** 标准建表语句。举一个最简单的建表示例，如下：

create table t1 (id int) stored as parquet;

****·**** 通过 TextFile 表创建 Parquet 表。这是离线数仓中的常规操作，假设已经准备好了一张 TextFile 表 catalog\_sales，在 hive shell 命令行中演示如下操作：

-- 创建parquet表（不压缩）

> drop table if exists catalog\_sales\_par;

> create table catalog\_sales\_par stored as parquet as select \* from catalog\_sales;

-- 设置parquet为snappy压缩

> set parquet.compression=SNAPPY;

> create table catalog\_sales\_par\_snappy stored as parquet as select \* from catalog\_sales;

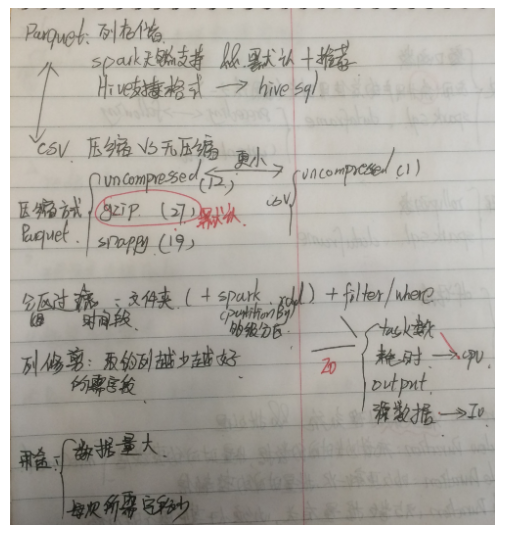
-- 设置parquet块大小

> set parquet.block.size=268435456;

> create table catalog\_sales\_par\_snappy2 stored as parquet as select \* from catalog\_sales;

****查看 Parquet 文件 Schema****

Parquet也是Spark SQL 的默认数据源，可通过参数spark.sql.sources.default 进行配置。



 这个 schema 中每条记录表示一个人的 AddressBook。

　　有且只有一个 owner，

　　owner 可以有 0 个或者多个 ownerPhoneNumbers，

　　owner 可以有 0 个或者多个 contacts。

　　　　每个 contact 有且只有一个 name，

　　　　这个 contact 的 phoneNumber 可有可无。

这个 schema 可以用图 3 的树结构来表示。

 每个 schema 的结构是这样的：

　　根叫做 message，message 包含多个 fields。

　　　　每个 field 包含三个属性：repetition, type, name。

　　　　　　repetition 可以是以下三种：required（出现 1 次），optional（出现 0 次或者 1 次），repeated（出现 0 次或者多次）。

　　　　　　type 可以是一个 group 或者一个 primitive 类型。

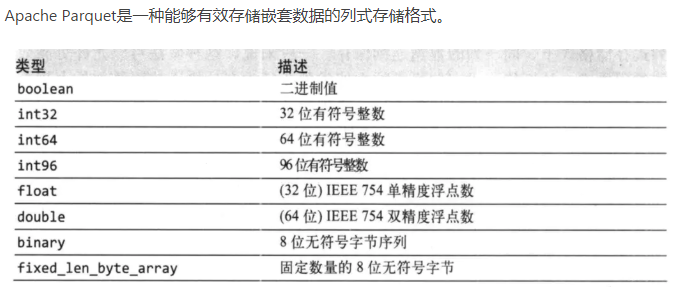
## **性能**

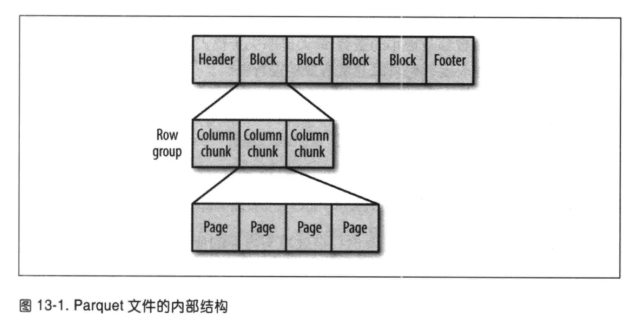
Parquet 列式存储带来的性能上的提高在业内已经得到了充分的认可，特别是当你们的表非常宽（column 非常多）的时候，Parquet 无论在资源利用率还是性能上都优势明显。具体的性能指标详见参考文档。

Spark 已经将 Parquet 设为默认的文件存储格式，Cloudera 投入了很多工程师到 Impala+Parquet 相关开发中，Hive/Pig 都原生支持 Parquet。

Parquet 现在为 Twitter 至少节省了 1/3 的存储空间，同时节省了大量的表扫描和反序列化的时间。这两方面直接反应就是节约成本和提高性能。

如果说 HDFS 是大数据时代文件系统的事实标准的话，Parquet 就是大数据时代存储格式的事实标准。



Parquet文件由一个文件头（header），一个或多个紧随其后的文件块（block），以及一个用于结尾的文件尾（footer）构成。文件头仅包含  
Parquet文件的每个文件块负责存储一个行组，行组由列块组成，且一个列块负责存储一列数据。每个列块中的的数据以页为单位。

作者：BigBigFlower  
链接：https://www.jianshu.com/p/76b7776ed567  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

A、公司当时已经上线spark 集群，而spark天然支持parquet，并为其推荐的存储格式(默认存储为parquet)。

B、hive 支持parquet格式存储，如果以后使用hiveql 进行查询，也完全兼容。

Drill可识别

****3.2.1分区过滤****

parquet结合spark，可以完美的实现支持分区过滤。如，需要某个产品某段时间的数据，则hdfs只取这个文件夹。

spark sql、rdd 等的filter、where关键字均能达到分区过滤的效果。

使用spark的partitionBy 可以实现分区，若传入多个参数，则创建多级分区。第一个字段作为一级分区，第二个字段作为2级分区。。。。。

****3.2.2 列修剪****

列修剪：其实说简单点就是我们要取回的那些列的数据。

当取得列越少，速度越快。当取所有列的数据时，比如我们的120列数据，这时效率将极低。同时，也就失去了使用parquet的意义。

* 当我们取出所有记录时，三种压缩方式耗时差别不大。耗时大概7分钟。
* 当我们仅取出某一天时，parquet的分区过滤优势便显示出来。仅为6分之一左右。貌似当时全量为七八天左右吧。
* 当我们仅取某一天的一个字段时，时间将再次缩短。这时，硬盘将只扫描该列所在rowgroup的柱面。大大节省IO。如有兴趣，可以参考[深入分析Parquet列式存储格式](http://www.infoq.com/cn/articles/in-depth-analysis-of-parquet-column-storage-format)
* ****分区过滤和列修剪可以帮助我们大幅节省磁盘IO。以减轻对服务器的压力。****
* ****如果你的数据字段非常多，但实际应用中，每个业务仅读取其中少量字段，parquet将是一个非常好的选择。****