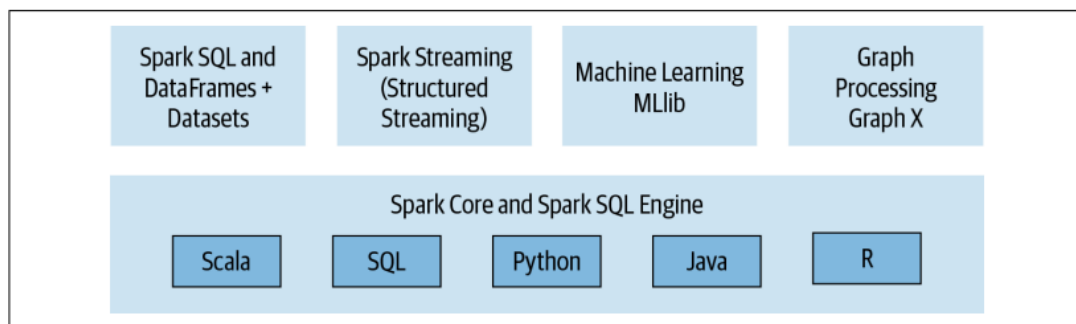
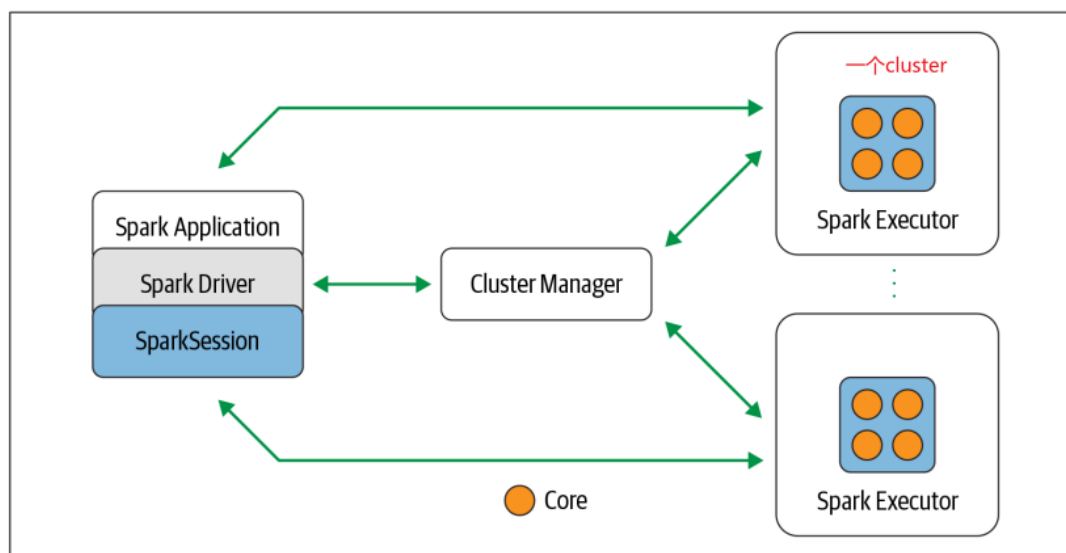


1、Spark概述

- Spark是使用Scala语言开发的，用于大型分布式数据处理和分析的计算引擎。它本身不是数据库，但是可以通过其Spark和Hadoop生态系统里面的其他组建连接其他关系数据库（比如MySQL），非关系数据库NoSql(redis, Hadoop HDFS), 流处理和信息队列(Kafka).
- Spark的计算基于内存而不是硬盘，使用**速度非常快**。Spark引擎内部使用有向无环图(DAG)来优化查询和计算语句，自动建立可以分布式计算的‘计算图’。其核心是弹性数据集(Resilient distributed datasets RDD)可在内存中迭代计算，更加适合运行需要交换数据的应用比如训练大型机器学习算法。
- Spark的结构和功能组件：



- **Spark core:** 提供核心的功能，用户使用不同语言的API然后交由Spark core优化运行，根据其支持的语言比如Scala Python Java等都会被优化转译为字节码，运行在集群机器中的JVM中。**Spark SQL:** 使用类似于SQL 语言的方式查询数据。**Spark Streaming:** 用于实时数据的流计算。**Spark MLlib** 集成了机器学习算法。**Spark GraphX** 实现图计算相关算法。
- Spark 语句是怎么样被“分布式”执行的：如下图每一个Spark应用有这些组成部分， driver,负责控制cluster的运行； driver 通过spark Session作为开发的接口来控制cluster manager.和spark exscutrer.



- **Spark driver:** 主要任务 与cluster manager通信并申请计算资源(CPU 内存)给spark executor(本质上是JVM)使用; 将所有操作转化成DAG形式，并分配到集群中的各个executor上执行计算。
- **Spark Session:** spark2.0后，同一的编程接口，所有操作都通过这个session进行。通过这个接口可以定义数据集并使用导入数据的函数、调整JVM参数，执行SQL语句等等。

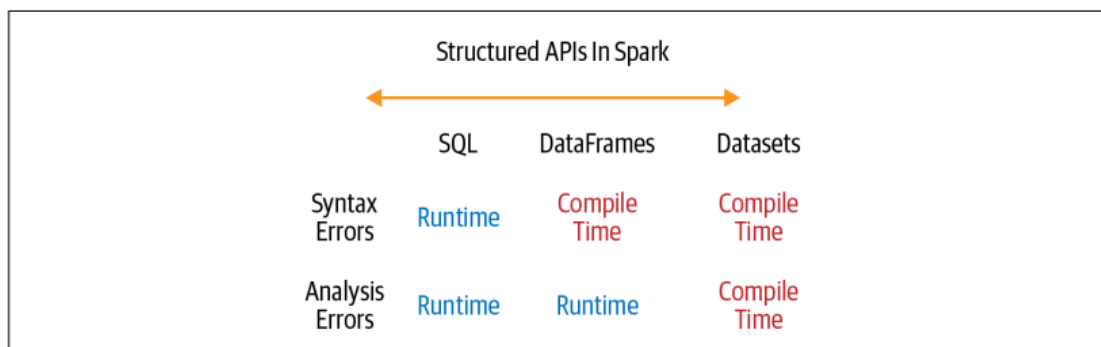
```
...  
// In Scala  
import org.apache.spark.sql.SparkSession
```

```
// Build SparkSession
val spark = SparkSession
  .builder
  .appName("LearnSpark")
  .config("spark.sql.shuffle.partitions", 6)
  .getOrCreate()
...
// Use the session to read JSON
val people = spark.read.json("...")
...
// Use the session to issue a SQL query
val resultsDF = spark.sql("SELECT city, pop, state, zip FROM table_name")
...
```

- **Cluster manager:** 负责管理集群中spark 运行中的node的资源。当前支持的有Spark 内置、Hadoop YARN, Apache Mesos, Kubernetes.
- **Spark executor:** 集群中的算节点，负责执行driver分配的任务，一般一个节点就只有一个 executor. 实现来说是一个JVM(java虚拟机)。可以是同一个主机，但是是虚拟机里面的不同实体创建的jvm, 也可以是一个很大的机房里物理意义上不同的机器，通过drive程序通信和执行计算过程。
- **Spark Docker 安装教程**

2、Spark 的核心，RDD，DataFrames 和DataSets

- 一点背景：这三种作用都是类似的，是Spark提供的数据处理的接口，所有操作都基于这之上在提供的API里进行。Spark 1.0时代是RDD, 后来1.3有了DataFrame, 1.6之后有了DataSet. 之后在2016年为了简化Spark团队又将后两者融合在一起了。它们有自己的特点和优势，不是说有了最新的DataSet了RDD就可以完全不用了。
- **RDD优势：** 1、高效和速度。2、一致性，RDD不可更改保证了数据可靠性。3、容错性，数据分别存在节点上，可重新计算会找回丢失的数据。
- **RDD 什么时候用：** 1、数据为非结构化数据，文本图像等等。2、数据没有严格的像数据库的模式定义，不想表格数据一样每一行有严格定义。
- 在JVM实现中，DataFrame 只是DataSet 的行 (row). 由于Python是动态语言，只能使用DataFrame.
- **DataSet/DataFrame 优势：** 使用简单，代码较少，用编译和运行速度换开发速度。有专门的优化器优化底层代码，高效。**适用情况：** 非结构化数据，需要对数据进行较高层次处理 比如SQL语句。



2.1 RDD和DataSet的创建

- Spark API入口: SparkSession(新, 2.0版本后) 和Spark Context(旧的方式, 1.0版本)

```
import org.apache.spark.sql.SparkSession
val spark:SparkSession = SparkSession.builder()
    .master("local[1]")
    .appName("SparkByExamples.com")
    .getOrCreate()
```

- 来自内存的一个对象集合, 来自外部数据集输入, 对已有RDD的转换。

```
// entrance of Spark is spark context
// spark context available as sc

// 定义1到10, 并行化计算, 并行度由运行代码的内核数确定
val params = sc.parallelize(1 to 10)
val result = params.map(performanceExpensiveComputation)

// outside data source
// define String type RDD
val text: RDD[String] = sc.textFile(path)
```

- DataFrame 可以想象成就是一个数据库的表, 或者python pandas里面的dataFrame一样, 由严格定义的行和列。DF有多种方式建立, 读取外部的结构化数据表, Hive, 现有的RDD。
- 建立DF:

```
// create DataFrame
val data = Seq(('James', '', 'Smith', '1991-04-01', 'M', 3000),
    ('Michael', 'Rose', '', '2000-05-19', 'M', 4000),
    ('Robert', '', 'Williams', '1978-09-05', 'M', 4000),
    ('Maria', 'Anne', 'Jones', '1967-12-01', 'F', 4000),
    ('Jen', 'Mary', 'Brown', '1980-02-17', 'F', -1)
)

val columns =
Seq("firstname", "middlename", "lastname", "dob", "gender", "salary")
df = spark.createDataFrame(data, schema = columns).toDF(columns:_*)
```

```
df.show()
```

>output:

```
+-----+-----+-----+-----+-----+-----+
|firstname|middlename|lastname|dob          |gender|salary|
+-----+-----+-----+-----+-----+-----+
|James    |          |Smith   |1991-04-01  |M      |3000   |
|Michael  |Rose     |        |2000-05-19  |M      |4000   |
|Robert   |          |Williams|1978-09-05  |M      |4000   |
|Maria    |Anne     |Jones   |1967-12-01  |F      |4000   |
|Jen      |Mary     |Brown   |1980-02-17  |F      |-1     |
+-----+-----+-----+-----+-----+-----+
```

2.2 RDD的转换和动作

- RDD提供了两种操作，**转换transformation**，即从现有RDD生成新的RDD。 **动作action**，则是对其进行某种计算 执行某种操作，结果要么返回给用户或者储存到外部。转换使用lazy evaluation, 对RDD执行一个动作前不会对转换操作执行实际动作。判断是T还是A, 如果一个操作的返回类型是RDD, 那它是一个转换操作， 否则是动作。
- 常见的转换： flatMap(), map(), reduceByKey(), filter(), sortByKey()
- 常见的动作： count(), collect(), first(), max(), reduce()

2.3 持久化

- 将计算的结果数据集缓存的到内存中，便于下一步计算时调用。对于大型应用，交互频繁的迭代算法这样可以大大节约时间。Hadoop 的MapReduce在执行另一个计算时必须要从硬盘中重新加载，即使这个数据是作为中间数据输入的，而Spark持久化可以在集群的内存中储存，要知道内存和硬盘的提取速度快了不止一个数量级了。
- 持久化的级别： cache()会将executor的内存持久化保存在每个分区，如果大小不够计算不会失败只是会重新计算分区大小，但是如果计算量大这样的代价还是很大，因此提供了一些折中的方法：序列化数据，再将数据储存。默认为MEMORY_ONLY, 序列化后的参数MEMORY_ONLY_SER. 虽然序列化需要一些计算时间，但是它生成更小更易储存的字节而不是对象。

```
// 假设现有 年份 当年某地温度的tuple数据
// (year, tempureture)
tuples.cache()
// cache()不会立即缓存RDD，直到下一个job运行时被缓存
// 取最大值
tuples.reduceByKey((a,b) => Math.max(a,b)).foreach(println(_))
> ...INFO: Added rdd ...

//运行另一个job，该RDD被加载
tuples.reduceByKey((a,b) => Math.min(a,b)).foreach(println(_))
```

2.4 序列化

- Spark 通过网络在executor之间传递数据和持久化数据前都要经过序列化，即将要对象或类转成字节。可以用java内置的java.io.Serializable, 但是效率较低。一般使用更加高效的Kryo 序列化库。
- 使用Kyro，先在设置中定义序列化的属性，然后需要先注册你定义的类。
- 序列化函数（也称为**闭包函数**）：函数也是需要被序列化的，如果引入了不可被序列化的类方法，需要在开发的时候发现它。

```
// 设置属性
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

// 注册,重写registerClasses() 的方法
class CustomKryoRegistrar extends KryoRegistrar{
  override def registerClasses(kryo: Kryo){
    kryo.registere(classOf[YOUR_CLASS_NAME])
  }
}

// 在driver程序中设置
conf.set("spark.kryo.registrator", "CustomKryoRegistrar")
```

2.5 共享变量

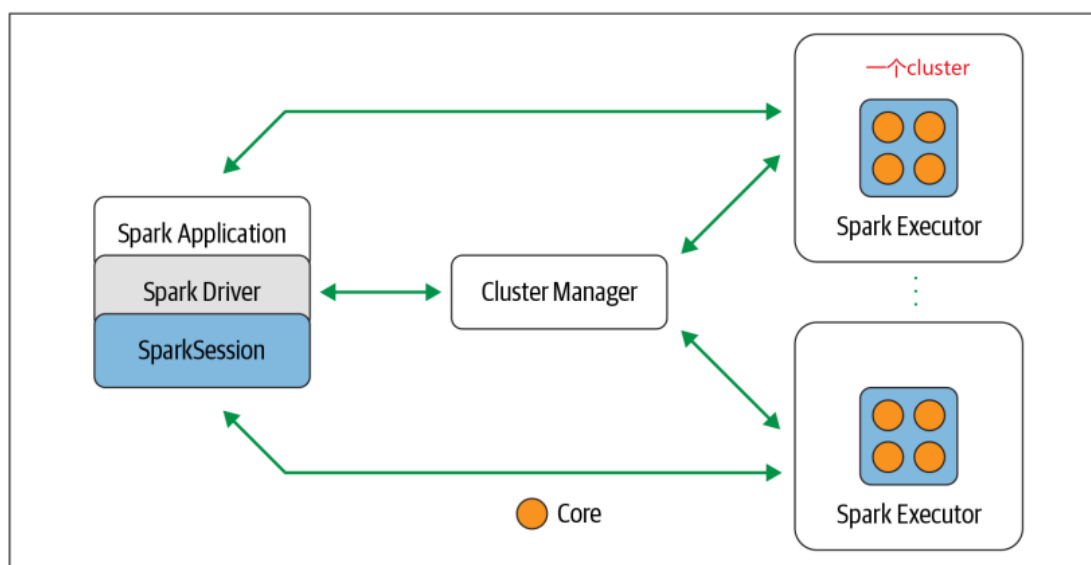
- 查询经常会使用到外部变量，这个变量一般在另一个节点或者集权上，所以如上文外部函数需要通过序列化后传输，这样有时候会降低效率。广播变量(broadcast variable)提供了一种解决方法。

```
//一个程序用到另一个查找表
val lookup = Map(1 -> "a", 2 -> "e", 3 -> "i", 4 -> "o", 5 -> "u")
val result = sc.parallelize(Array(2, 1, 3)).map(lookup(_))
assert(result.collect().toSet == Set("a", "e", "i"))
```

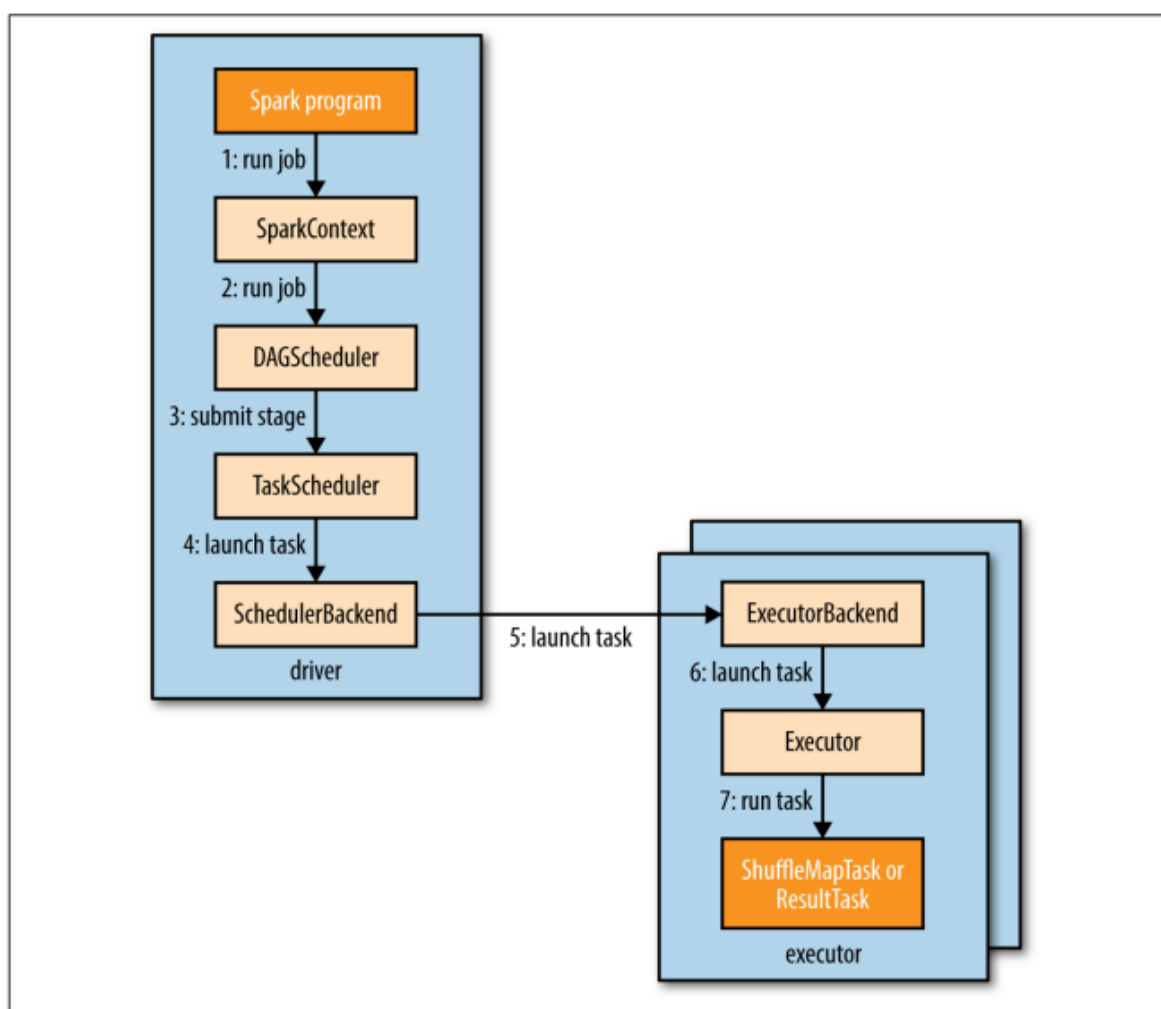
- 广播变量：**序列化后发送给各个executor并缓存以便适时调用，常规变量因为时在函数内部的，所以每次任务都需要序列化传输。现有创建一个sc.broadcast变量，返回对应类型的封装。广播变量是单向传播的，无法更新，也无法从executor传会driver。

```
val lookup: Broadcast[Map[Int, String]] =
  sc.broadcast(Map(1 -> "a", 2 -> "e", 3 -> "i", 4 -> "o", 5 -> "u"))
val result = sc.parallelize(Array(2, 1, 3)).map(lookup.value(_))
assert(result.collect().toSet == Set("a", "e", "i"))
```

3、Spark内部运行机制



- 回顾上文图2, Spark运行时有两个独立实体, driver 和executor. Driver负责管理调度 SparkContext应用, executor专属于应用, 应用运行它就运行并执行该应用的任务。



- 作业提交:** 上图,当用户提供SparkContext调用一个动作(action)时, spark自动提交一个jb(步骤2), 然后运行两个调度程序, **DAG调度**和**任务调度**。DAG调度负责将任务分解为若干阶段并将这些阶段构成一个DAG,任务调度这负责把这些分解的每个阶段提交到不同的集群执行。
- DAG构建:** 一个job怎样被分为不同阶段。job阶段有两种类型的任务, shuffle map 和 result, shuffle map类似于你需要分开10kg混合在一起的红绿豆, 现在将它分成10份分给你的10个倒霉朋友每个朋友作为一个cluster执行1kg的分开两种颜色的任务。result任务则是运行在最终阶段, 比

如10个朋友分开完两份并统计好两种的数量，该计算任务在自己分区的cluster上计算，把结果返回给driver(你自己)，你再将最终结果10kg豆子汇总成最终结果。复杂的任务可以有多个shuffle 的阶段

- **任务调度**: 上图步骤3，任务集合被发送到调度程序后，构建任务到executor的映射，将任务分配到具体的内核，并在任务完成后继续分配直到任务集合完成。当任务完成或者失败时，executor都会向driver更新消息。具体向executor分配的顺序为：首先分配进程本地任务(process local), 然后节点本地(node local), 然后机架本地(rack local), 最后分配非本地任务或者推测任务(speculative task)
- **任务执行**: 上图步骤7，首先executor会确保任务相关的依赖文件和jar包都是最新的，然后由于任务代码经过序列化后发送到集群上的，先反序列化任务代码和函数，最后执行代码，由于任务运行在executor相同的JVM中，任务启动没有进程开销。

执行器和集群管理，executor具体是如何工作的

- Spark 任务要依赖executor来运行任务，而管理executor的为集群管理器cluster manager.Spark 提供了各种不同特征的集群管理器。
- **本地模式** local: executor 和driver在同一个JVM中，用于测试和小规模数据。
- **独立模式** standalone: 实现了简单的分布式系统，运行一个master多个worker, Spark 启动时 master 控制worker生成一个或多个executor。
- **YARN**: YARN是Hadoop使用的资源管理器，每个运行的Spark应用对应一个YARN的实例，每个executor在自己的YARN容器中运行。YARN相对于独立模式的优点是它考虑了集群上的其他应用，同一协调它们之间的资源和调度。YARN分为客户端模式(client)和集群模式(cluster)
- **YARN client**: driver程序在客户端运行，即独立与集群的另一台机器。1.driver构建新的spark context实例时就启动了与YARN之间的关联，2, 3.提交YARN应用后启动集群管理器上的容器，excutorLauncher的作用是向管理器请求资源启动backend进程，“真正的”executor的JVM. 启动executor的数量在spark-shell 中设置，还要设置每个用到的内核和内存数量。

- ```
%spark-shell --master yarn-client \
 --num-executors 4 \
 --executor-cores 1 \
 --executor-memory 2g
```

-

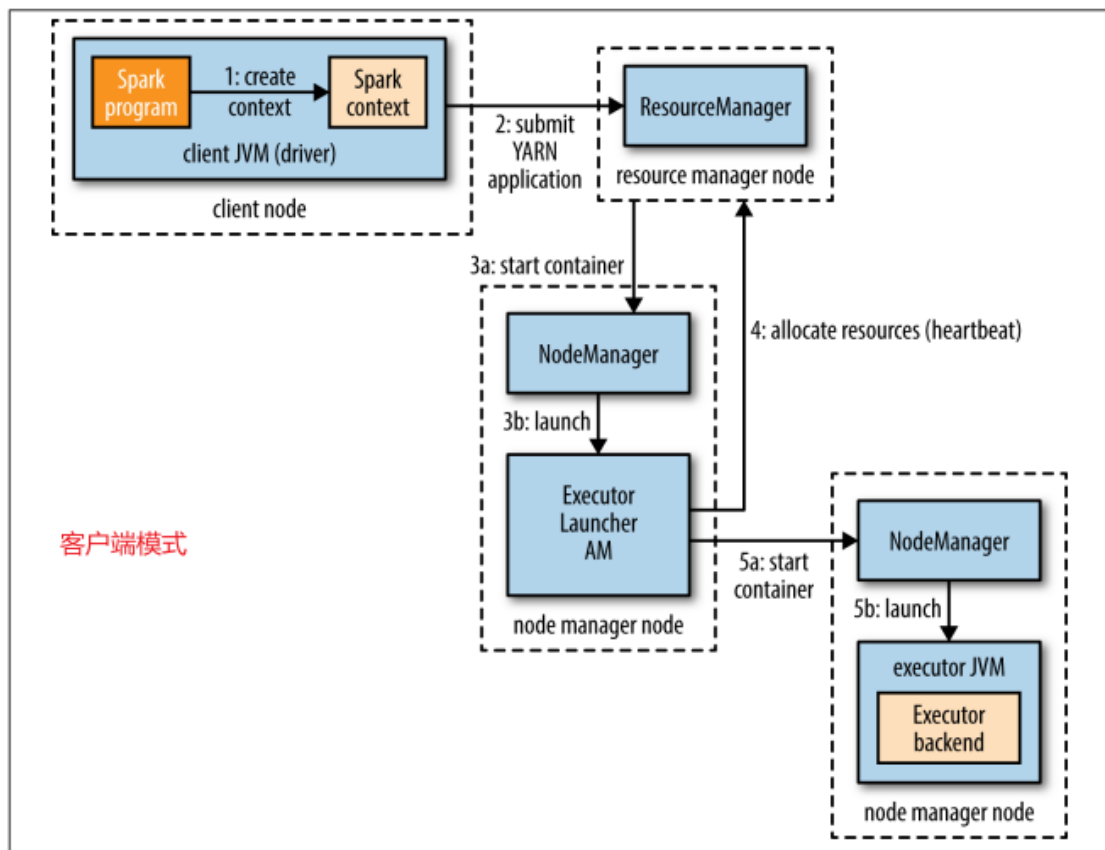


Figure 19-3. How Spark executors are started in YARN client mode

- **YARN Cluster:** driver程序在集群上运行. 客户端会启动YARN应用但是不会运行任何代码，注意图中的spark program是在集群中的容器中运行的。

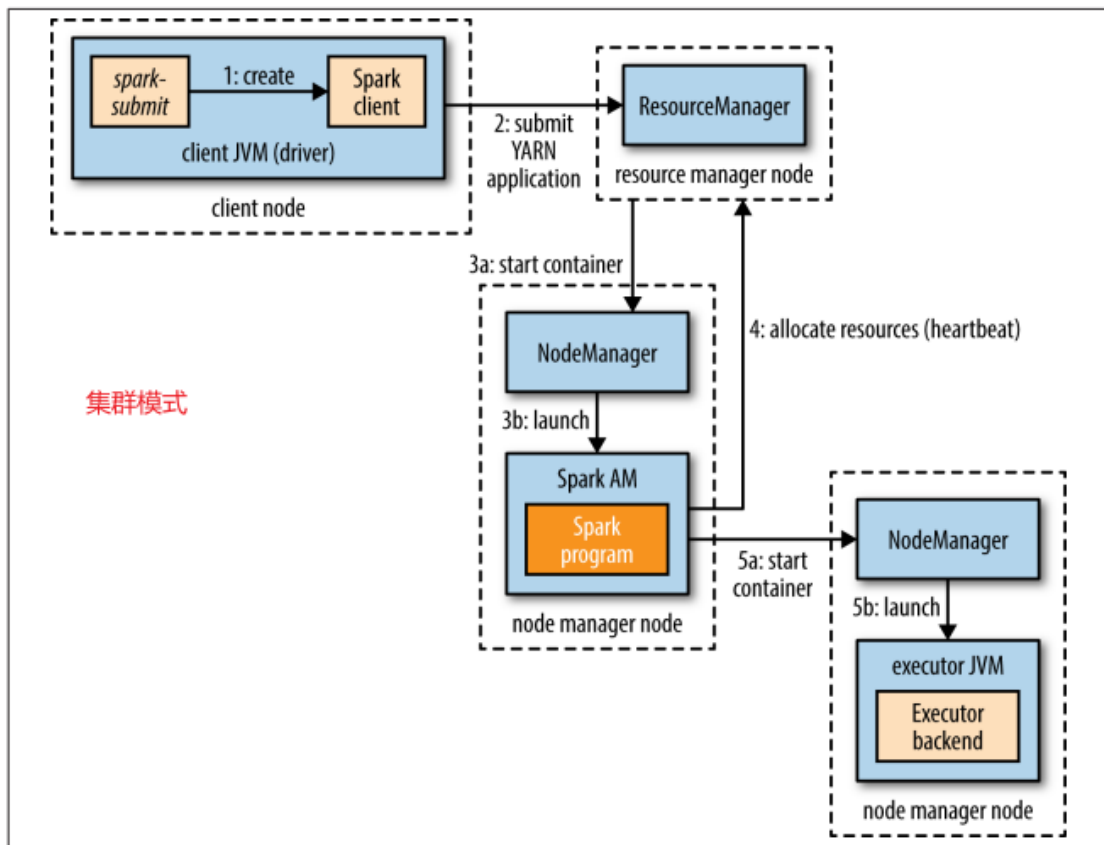


Figure 19-4. How Spark executors are started in YARN cluster mode

- 总结：不同部署模式对比：



Table 1-1. Cheat sheet for Spark deployment modes

| Mode           | Spark driver                                       | Spark executor                                            | Cluster manager                                                                                                       |
|----------------|----------------------------------------------------|-----------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| Local          | Runs on a single JVM, like a laptop or single node | Runs on the same JVM as the driver                        | Runs on the same host                                                                                                 |
| Standalone     | Can run on any node in the cluster                 | Each node in the cluster will launch its own executor JVM | Can be allocated arbitrarily to any host in the cluster                                                               |
| YARN (client)  | Runs on a client, not part of the cluster          | YARN's NodeManager's container                            | YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors |
| YARN (cluster) | Runs with the YARN Application Master              | Same as YARN client mode                                  | Same as YARN client mode                                                                                              |
| Kubernetes     | Runs in a Kubernetes pod                           | Each worker runs within its own pod                       | Kubernetes Master                                                                                                     |

## 项目实战--Spark 机器学习 用户流失分析

### 1. 问题定义，商业背景和要解决的问题

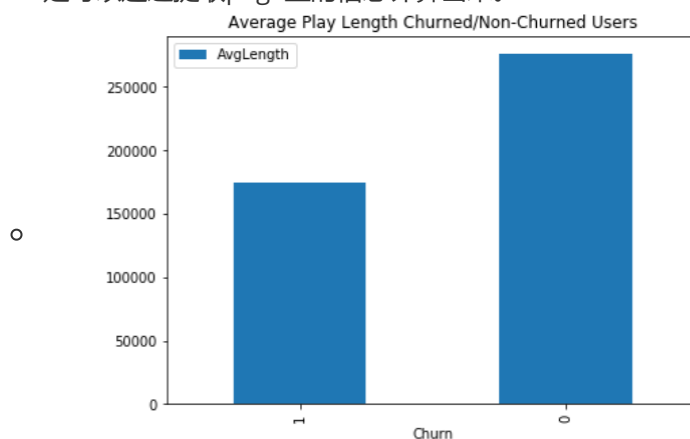
- 流失的定义，阶段内失去的客户数量/ 该阶段开始时客户数量. 可能放弃使用了 可能使用了潜在竞争对手的app,等等原因
- 怎么做：通过分析用户行为数据，发现流失用户流失前的使用有什么特征，以便未来预测将要流失的用户，对这部分用户进行提前干预以降低流失率。或者把这部分用户的行为提取出主要的模式，比如哪个页面跳出率异常的高，哪个功能使用率比较低，为修改新版本界面布局提供参考。

### 2. 数据介绍和EDA、特征工程

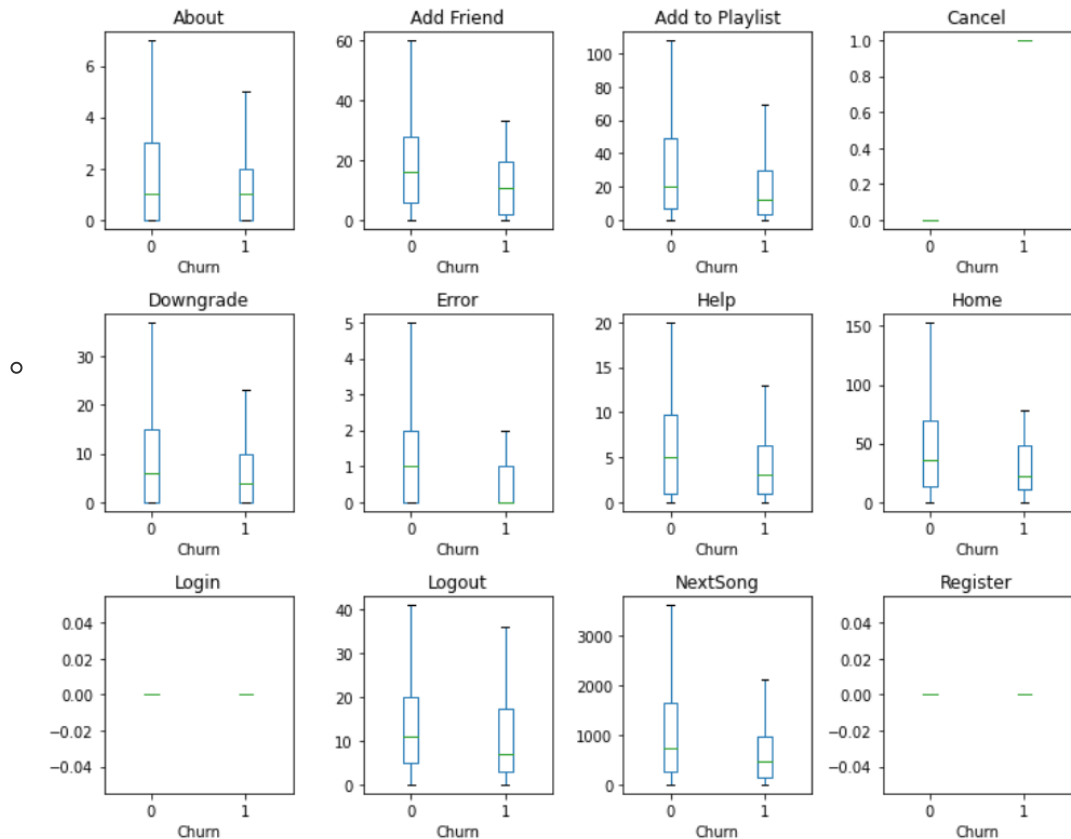
```
1 events.printSchema()
```

```
root
```

- ```
|-- artist: string (nullable = true)
|-- auth: string (nullable = true)
|-- firstName: string (nullable = true)
|-- gender: string (nullable = true)
|-- itemInSession: long (nullable = true)
|-- lastName: string (nullable = true)
|-- length: double (nullable = true)
o |-- level: string (nullable = true)
   |-- location: string (nullable = true)
   |-- method: string (nullable = true)
   |-- page: string (nullable = true)
   |-- registration: long (nullable = true)
   |-- sessionId: long (nullable = true)
   |-- song: string (nullable = true)
   |-- status: long (nullable = true)
   |-- ts: long (nullable = true)
   |-- userAgent: string (nullable = true)
   |-- userId: string (nullable = true)
```
- 该产品为在线音乐播放器，数据中记录了如用户听过的歌曲，歌手，地点，设备信息等等。以上是数据包含的所有特征列的名称，数据近28万行，包含226个用户的使用数据记录，其中52个在周期后流失了。用户的操作数据记录在page列中，主要操作有下一首，取消，添加播放列表取消订阅等等。
 - 由于流失和未流失的数量相差太多(52vs174), 可以进行如SMOTE上采样的方式构建较为平衡的类别标签。
 - Page中的用户操作，平均播放时间、看起来对流失率有相关，计算和可视化后发现它们的平均时长确实存在统计学意义上的区别，比如流失用户添加朋友，添加歌曲次数明显少于未流失的。
 - **特征工程**：创建新的特征，这些特征至少逻辑上要与流失率有关系，并且能够提高模型的准确率。比如平均播放时长，连续活跃天数，付费天数，日均使用时间，平均使用间隔等，用户性别，注册时长，听过的歌/歌手数量，设备数量等等。这些特征虽然数据没有直接给出来，但是可以通过提取page里的信息计算出来。



Boxplot grouped by Churn



3. 建立模型

- 创建了三个模型，逻辑回归，随机森林和提升树。Spark和sk learn 不同的是，输入的X值需要先向量化再输入模型中拟合。

```
vec_assembler = VectorAssembler(inputCols = feature_names, outputCol =
"Features")

# scale each column
scalar = MinMaxScaler(inputCol="Features", outputCol="ScaledFeatures")

# build classifiers
rf = RandomForestClassifier(featuresCol="ScaledFeatures",
labelCol="label",
numTrees = 50,
featureSubsetStrategy='sqrt')

lr = LogisticRegression(featuresCol="ScaledFeatures", labelCol="label",
maxIter=10, regParam=0.01)

gbt = GBTClassifier(featuresCol="ScaledFeatures", labelCol="label")
```

4. 模型评估

- 由于测试数据0和1样本不平衡，单纯的使用accuracy不能很好反应模型效果，比如100个样本了80个位1，20个位0，一个模型及时把全部测试数据预测为1，也有80%正确率。
- 定义confusion矩阵的四个量，TP true positive真实为正预测也为正标签，FP false positive, 真实为正预测为负，FN false negative, 和 Ture negative.
- 精准率表示在全部预测为真的例子里面，有多少是正确的。precision=TP/(TP+FP),

- 召回recall = TP/(TP+FN), 表示真实为真的标签中有多少被预测出来了。
- F1 score是两者的结合, $F1 = 2 * p / (p+r)$
- 其中一次训练表现最好的随机森林f1=0.82, 比没有特征工程前0.72提升不少。
- 下一步需要将完整数据集12GB 部署到集群上运行和训练。

5. 总结、对应的业务策略建议

- 数据可视化很重要, 拿到不熟悉的数据先检查数据的完整性, 相关性, 将一些特征图形化, 再检查我们要分析的指标是什么, 弄清定义和计算后考虑和这个指标相关性高的特征, 同样不明显时借助可视化工具。
- **特征工程也很重要**, 要理解业务流程, 指标的产生取创建新的特征。同样的, 计算出新特征先按照不同类别(这里是流失vs 不流失)做图, 如果一个特征两者差别明显, 比如平均听歌时长, 流失的明显比未流失的短, 这个就是一个新特征的备选项, 可以加入模型里训练。
- 就App这个使用场景来说, **以时间为一个维度, 另一个变量为一个维度可以衍生出很多有用的特征**, 使用时长是一个, 还要页面停留时间, 以天为单位哪个时间段出现高峰, 连续使用天数, 平均/最大间隔使用天数等等, app核心功能时长占使用总时间的比例等等(听歌 vs 看社区, 查找歌曲 看热榜, 商店等等)。
- 通过提升树模型的feature importance 图, 发现决定模型最重要的有这几个特征: 活跃天数, app内使用的session个数, 注册时间, 按歌曲'不喜欢'键等等。
- 根据上面的结论, app用户流失其实转化成了常见的问题了: 怎么提高日活等指标。如果将一个服务/产品分为三个重要维度, 我会从这三个方面分析: **内容, 平台和用户**。
- 内容就是这个app提供的服务本身, 音乐app当然也没有全面优质的音乐版权, 决定了用户的使用频率。其他向视频类app, 新闻类, 社交类甚至游戏类, 服务本身的质量和怎么样, 不同阶段这些内容提供者出现了什么变化可能会影响到相关指标。
- 平台本身, 平台怎么样能好用, 易用, 能吸引到用户轻易发现他们喜欢的内容也很重要。UI 布局, 节点的转换逻辑等等。比如上面的例子中session_num很大的用户, 可能的切换歌曲频繁造成的。

参考资料

主要笔记来自阅读以下两本书:

1. Hadoop: The definitive guide
2. Learning Spark 2nd

有用的网页:

1. <https://phoenixnap.com/kb/rdd-vs-dataframe-vs-dataset>
2. [Apache Spark Tutorial with Examples - Spark by {Examples} (sparkbyexamples.com)](<https://sparkbyexamples.com/>)
3. 项目来自于udacity的课程大项目

