

它肯定需要有一个分布式的cluster

这个cluster它有很多个node组成

就是一个一个的节点组成

我们把一个node就叫做一个worker

一个Spark的cluster里面

它就会有多个worker node

这时一个node就可以认为是一台

我们可以把它当做一台机器

在一个worker node上面

Spark会可能启动一个或者说多个executor

在同一个worker node上面

在这边我们假设

每一台worker node

都只启一个executor

而一个executor是

在这个node上面的一个进程

单单启动一个进程

是不能够达到很高的一个并行度的

所以在Spark里面

它会在executor内部

再去启很多个task

而这个task是真正去做计算的单元

这个task是以进程的形式存在的

那么也就是说

我在Spark的并行度上面可以看到

首先第一级别

它是有一个worker node的并行级别

然后第二个再往细的分

它是一个

executor并行级别是一个

进程的一个并行级别

最底层真正去做一个

并行化的计算任务的时候

它是以task这种线程的方式

去做的一个并行

也就是说Spark最低的一个并行级别

是一个线程级别的并行

它的整个框架是以线程的并行级别

来实现这种分布式的计算

这张图上我们还可以看到

Driver

它这个Driver Program

Driver Program其实就是我们

Spark的一些应用

比如说word count这个应用

它就是通过Spark Context

去把它传递给Cluster Manager

Cluster Manager它是一个

用来管理这么多

worker node的一个Manager

就相当于它是一个

像master一样的东西

由于Spark它有不同的运行模式

现在的话它至少可以支持三种

第一种是standalone模式

standalone模式的话

它就是一个master worker这种架构

还有一种是yarn的这种模式

在yarn模式下面

Cluster Manager就由

yarn会自己去管理这些worker node

还有一种mesas

mesas mode的话跟yarn模

这些都是差不多

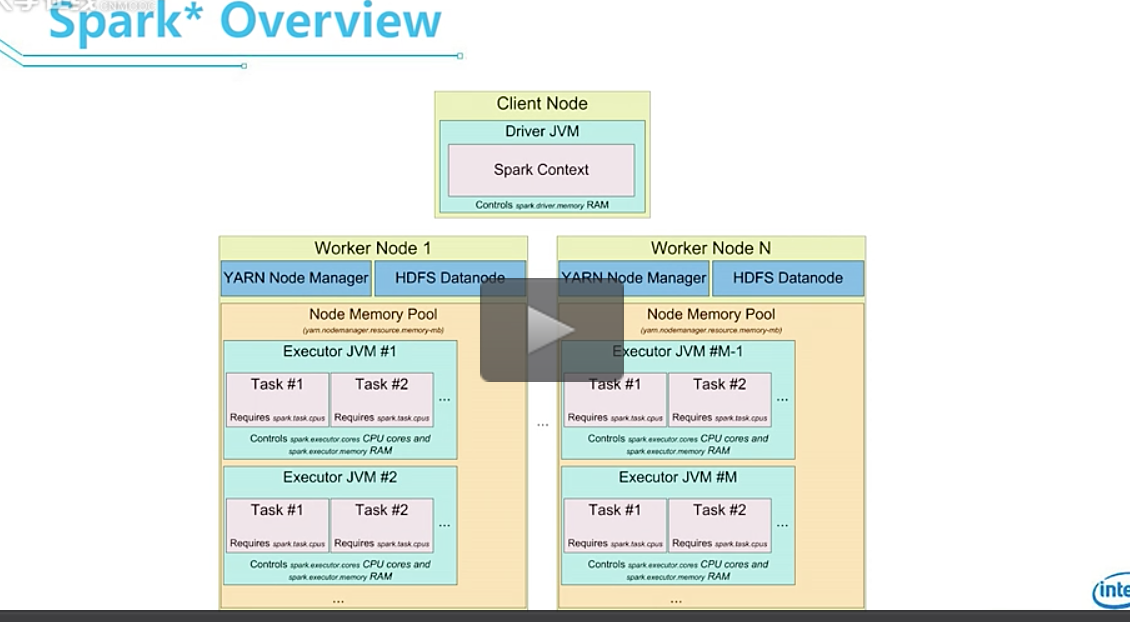
基本上都是由一个Cluster Manager

去管理下辖的

不同的worker node里面

怎么去进行一些任务的分配

或者说一些资源的调度等等



我们这里用yarn mode

来作为一个例子

我们可以看到

在yarn mode上面

我们其实也是可以有

多个的worker node

worker node 1到

直到worker node N

在每一个worker node上面

首先在yarn mode上面

它会有一些yarn相关的一些进程

包括node manager进程

另外一个的话也包括

HDFS Datanode进程

这两个是一样相关的一些进程

然后每一个worker node上面

我们可以启动这些executor

在一个worker node上面

我们可以启多个

从executor 1甚至到executor N都可以

在每个worker node上面

都可以启动多个的executor

在每一个executor内部

我们也可以启动多个task

task1、task2甚至到task N

比如说这个executor

它需要多少个CPU

它需要多少个memory

都是通过

比如说Spark ？的时候

你可以去指定

然后由一样的调度

去给你分配出一些

每一个executor

给它多少个CPU

给它多少memory的这种资源

来进行一个资源上面的分配

然后分配完了之后

我们这个Spark的application

就可以在这些资源的基础上

去启东这些一个一个的线程级别的task

task的启动是根据

我在每一个默认情况下

在每一个executor上面

我给了它多少个CPU

那么我默认就可以并行去启动多少个task

我们现在来看一下Spark Shuffle

所谓Shuffle就是一个

将数据重新进行归类的一个过程

在Spark里面

Spark Shuffle会将Spark里面

RDD的一个DAD

进行一个stage和stage之间的一个划分

比如说我这边的一个Shuffle的过程

会将RDD里面生成一个stage1

和一个stage2

在RDD这个Shuffle之前

我这边

比如说这是一个M Stage

那这边就是

下面的话这边是一个R Stage

这里面这个Shuffle干了什么呢

首先我们来看一下

在stage1里面

我们这边有这么一个RDD

在这个RDD里面

我们有很多个

比如说这边有四个partation

比如说我假设

这一个partation里面

它呈放的原始数据是a1.b1

我这一个partation里面

呈放的数据是a2.b2

我在经过第一次的转换

比如说我进行转换

我只是做一个？的话

我这边就会变成一个

a1.b2

从这一个变到这一个

那对于这一块的话

这边就会变成a2.b2

那我这个Shuffle它做了什么呢

它就是说

我为了得到这一块原始数据里面

我想要拿到

a到底出现了多少次

b出现了多少次

或者甚至说这边有cde

cde分别出现了多少次的话

那我就要把这些数据进行归类

归类的过程就是说

我a要把它统一到一个地方

然后我再来进行计算

这个a出现多少次

这个过程就是一个Shuffle的过程

比如说我经过Shuffle之后

我需要让a都放在同一块partation里面

去进行计算

去进行count到底这个a出现了多少次

那么我就需要把这一块里面的a

把它也传到这个里面来

我要把第二个partation里面的a

也要传到这边来

第三个里面的a也要传到这边来

然后这边就可以拿到

a这边到底出现了多少次

这个过程就是一个Shuffle的过程

所以在这一块的话

我这边就会拿到一个

从第一个partation过来的a1

以及从第二个partation过来的a2

以及从第三个partation过来的a

比如说随便一个a3好了

那这边比如说第一块的话

第一个partation我这边a出现了一次

第二个partation里面

我a出现了两次

那我在这边汇集到了

所有关于a的一个集合

在最后我就进行一个累加

那这一块的话

我就能够拿到a所有的数据

所以说a等于

对应的就是

a这边后面就等于1加2加3等等

这边最后算出来的一个数字

那我就拿到了一个

所以跟a相关

a出现了多少次的结果

这个Shuffle的过程

我能够把之前

每一块里面的集合

把它抽出来每一块

让每一块相关的都落到

新的partation里面

让它能够放到一起

然后进行一个跟方便的计算的过程

所以说它是一个将数据重新组合的过程

所以这个就是Spark的一个Shuffle

更具体的话就是说

比如说我这边有很多个Map Task

下面也有很多个Reduce Task

Map Task Map Task

每一个Map Task

它就会为我自己Map Task所处理的数据

重新去把它进行一个归类

比如说我这个Map Task的话

我就要对应三个bucket

因为我这边有三个Reduce

因为每个Reduce它对应的是一个

不同的集合

我就要对这个Map Task里面所有的数据

分成三个集合

每一个集合我们交给它一个bucket

每一个bucket就对应一个Reduce

就像每对应一个类

对于Map Task

每一个Map Task

Map Task

Map Task1 Map Task2 Map Task3

每一个Map Task

它都要去生成这三类的数据

就是这边的蓝色、黄色、绿色

这三类的数据

对于每一个Reduce Task来说

它就要从这些Map Task

归完的这些数据

去拿这些数据过来

每一个红色就代表了

Reduce Task1

每一个黄色代表着Reduce Task2

需要处理的数据

每一个绿色的

就会代表Reduce Task3

需要处理的数据

所以这个Reduce Task会分别从

Map Task1 Map Task2

Map Task3上面

去把红色的这些bucket去取过来

进行计算

这就是整个Shuffle的过程

然后这个bucket只是一个抽象

它以什么样子的形式存在

是要看Spark里面怎么去实现

后面我们会讲到

在这张图上我们可以看到

从Map Task去生成这些bucket的过程

是一个Shuffle write的过程

而从Reduce Task

去把这些bucket

这些数据拿过来的这个过程

叫做Shuffle fetch

或者叫Shuffle lead的过程

另外一个的话呢

我们在Shuffle write

或者说Shuffle fetch

整个的这个过程之中

我们可能会加入一些

Aggregator的这些操作

从而使得数据量进一步的减小

比如说在这个Shuffle之前

我们会进行一个

先进行一个Aggregator

Aggregator之后

我的数据量就会进行

如果重复的数据比较多的话

我的数据量就会急剧的下降

我们这样子的话

生产的这些bucket数据就会更小一点

这样子的话

就会在Reduce去拿这些数据的时候

产生的网络开销就会小一点

这个就是插入Aggregator的一些作用

这个里面的

比如说Map Task

这边的Aggregator的话

我们其实跟Hadoop里面

比如像mapside combine这种的功能

是类似的

接下来我们看一下

在Spark里面Shuffle是怎么实现的

在Spark里面

Shuffle可能会有几种不同的内容

实现方式我们首先来看

Hash Based的这个Shuffle

在Hash Based Shuffle里面

它就完全相当于是以Hash的这种方式

去对一个partation里面的数据进行归类

不同的一类的数据

它就会放到不同的File里面去

那也就是说

你有多少个Reduce Task

那每一个Map Task

就会生产多少个File

比如说在这张图上面

我们可以看到

每一个Map Task都生成了四个File

就相当于四个Reduce Task

它被分成了四类

每一个Reduce Task都会去

从每一个Map Task这边

去拿一个File

拿到这边来去进行处理

这就是Hash Based的一个Shuffle

根据刚才之前的那张图

我们可以很容易的发现

我如果有M个Map Task

以及有R个Reduce Task的话

我生成的总的Shuffle文件数量的话

就是M乘以R的这么多

Shuffle文件数量

如果M和R都非常非常多的话

那这个生成的总的文件数量

是非常非常多的

另外

因为Map Task如果很多

之前我们介绍过

我们一个executor里面

可以同时运行很多个Map Task

而这个同时运行的Map Task的个数

我们可以是由这个executor

被分配到的CPU的个数来决定的

默认情况下是这样

那这个CPU的个数

我们假设是C的话

那么每一个executor

同时都会有C个Map Task在同时运行

而每一个Map Task

它都会生成R个文件

就相当于我在每一个executor里面

都会生成C乘以R个Shuffle文件

那么如果当R非常大的时候

每一个executor都会生成

非常非常多的Shuffle文件

这同时也就会生成很多个file handlers

我们也知道

你生成非常非常多的file handler

会影响机器的并行速度

另外的话

operation system在操作系统里面

也会有一个上限

如果特别特别大就有可能超过上限

因为一个executor就会有C乘以R

你有可能在一个node上面

就会有很多个executor

N个executor的话

那就是N去乘以一个C乘以R个文件

在一个node上面要这么多

而对于memory来讲

它也是一个巨大的开销

如果说一个executor要C乘以R个文件的话

那我每写文件的时候

在Spark里面

每一个默认情况下

它是需要64K的buffer

为每一个文件会准备64K的buffer

那也就是说如果

我这个C乘以R非常大的话

再乘以一个64K的话

它对内存也是一个非常大的压力

所以Hash Based的Shuffle

会暴露出很多问题

首先总的文件数量非常非常的多

会导致在系统里面

Inode会非常非常的多

小文件的处理非常非常耗磁盘的IO

特别是在Shuffle read的时候

因为我要频繁的去访问

不同的小的碎片文件

另外它会对内存会产生巨大的开销

如果R非常非常多

每一个文件我都需要64K

去做这个文件的buffer

我就需要非常非常多的memory

memory也压力非常大

所以说Hash Based的Shuffle在这些方面

它是有一些性能的缺陷

我们刚才看到Hash Based的Shuffle

它会有一些性能上面的缺陷

特别是当M乘以R

这个数量非常非常大的时候

那我再去读取这些文件的时候

因为这些都是一些小的一些碎片文件

一个bucket对应一个文件

那么这样子的话

我读取这些小文件的时候

会对整个的desk的IO可能会

压力会比较大

所以我们就想是否能够

让这个文件数目更加减少呢

就有人提出是否可以让这些bucket

去共用一些file

比如说我们现在

M我有100个Map Task

而当前我的CPU的个数

我却只有10个

在这个class里面

我有100个的Map Task

而我只有10个CPU

在这种情况下

我有多少个CPU

在默认情况下

我就只能去同时运行多少个Map Task

而在这个情况下面

同时运行的Map Task数目就是10个

在每一个Map Task这

比如说我们这边有4个Reduce Task

每一个Map Task

它都需要去生成4个bucket

在这个图上就是红、黄、绿、蓝

这4个bucket

在之前Hash Based的Shuffle

的这个模式下

我就相当于每一个Map Task

都会生成4个file

OK 我们现在同时运行了10个Map Task

那我至少可不可以生成4乘以10个的file

这是第一批的这10个Map Task

会生成4乘以10个的file

每一个bucket放在了一个file里面

第二批10个的Map Task

运行的时候

我就会看

第一个bucket

它是会应该放到哪一个file里面去

第二个Map Task

它的bucket

因为这些所有的bucket

是否可以跟之前的bucket

放在同一个file里面

这样的话我们不就可以

将整个的file数目较少了吗

所有的红色的bucket

最终都是要放到同一个Reduce Task里面

那我把它同样放到同一个file里面去的话

我这个Reduce Task

只要直接去访问这么一个file不就可以了吗

所以说这样子的话

我如果把多个bucket

去共享一个file的话

那么在这种情况下我就可以去

很大程度上去减小

总的file数目

同时我能让一个file的大小能够更大

而在Reduce Shuffle fetch

或者说Shuffle read file的时候

我一次可以读更多的数据

特别是在机械硬盘的时候

我这样子就可以极大的提高

硬盘读写的效率

所以我在第二次Map Task起来的时候

我就看

这个红色的bucket

它就应该放在第一个file里面

去追加到

之前这个bucket写到这个file里面去

这个黄色的

同样也去追加到它之前

前一批这10个Map Task所生成

之前生成的的file里面去 去追加

所以最终如果说我是有10个批次

就是说我耗时10个的情况下

我的Map Task有100个的情况下

我就相当于我要分10批Map Task去运行

那么最终的结果就是

我每一个file里面

就会有10个bucket共用了一个file

在Reduce Task去拿这些bucket的时候

它一次就可以直接拿一个file出去

就可以了

所以说一个file就相当于对应了一个

C乘以R

这什么意思呢

就是说我同时运行了C个Map Task

并且生成了

每个Map Task生成了R个文件

所以

总共的文件数量我就只需要生成

C乘以R个文件数量就够了

如果当C远小于M的时候

那么这个整个的文件数量

就会小非常非常的多

并且每一个file又比原来的file会大很多

所以效率上面

总的文件数量又少了

而每一个文件的读取的效率又高了

所以说通过这个Consolidate Shuffle

就可以极大地提高文件读写的效率

我们看到

Consolidate Shuffle file

它相对于之前

最开始的Hash Based的Shuffle

它从M乘以R个文件数量

下降到了C乘以R个文件数量

但是这并没有解决几个问题

第一 如果我的CPU的数量

就是这个C的数量

是等于M的数量的时候

它是跟原来的是一模一样的

就是说任何性能问题都没有解决

另外

我之前所有同时打开的文件句柄

file handler

它跟之前原来的这个

最开始的Hash Based的Shuffle

也是没有改变的

仍然是C乘以R

另外

我每一个未打开的这些文件是

之前会有一些memory的buffer

那些buffer

比如说每一个都是32加32K

那仍然是C乘以R去乘以一个memory的数据

32加32K

所以说对于内存上面

它也仍然没有任何的一个变化

所以在这个Consolidate Shuffle里面

Consolidate Shuffle

唯一相对于原始的

Hash Based的Shuffle的好处就在于

当M的数量是大于C的数量的时候

这个Consolidate Shuffle

才会体现出它的优势

所以说这个Consolidate Shuffle

仍然没有从根本上去解决

一些比如说像文件过多的问题

以及memory消耗太多的这些问题

所以呢

我们还需要一些其他的方式来

去提高Shuffle的性能

针对刚才之前讲过的

最原始的Hash Based的Shuffle以及

对原始Hash Based的改进版

Consolidate Shuffle

之前这两种模式的缺陷

后来我们引入了Sort Based Shuffle

Sort Based Shuffle

就是说我会将Map Task里面的所有数据

进行一个Sort

首先它至少会根据它的partation

去进行Sort

所谓的partation就是说

假设我现在有4个Reduce Task

这个Reduce Task

它将

在Map Task这边

它会对我这些数据

进行一个数据的partation

比如说partation1的话

我就会放给Reduce1

partation2的话我就会个Reduce2

这样子的话我根据partation的ID的大小

我就可以去进行一个排序

排在最前面的是file1的数据

排在第二的是file2的数据

接下来就是3、4

这样这个数据的存在方式就是

在这个文件里面的一个一个segment

当然

至少它会根据排序ID去排序

甚至如果说？？的话

我会在排序内部会根据key

去进行排序

首先相对于之前的

Hash Based的Shuffle的话

我对于每一个Map Task

我只需要生成这么一个文件就够了

这个文件里面

只不过会有很多的FileSegment

而每一个FileSegment

就相当于抽象出来

其实就是一个bucket

一个FileSegment就是一个bucket

只不过这些bucket是按顺序

存放在了这个文件里面

最终我以Sort Based Shuffle

它的总的文件数目

所有的Shuffle的文件数目

就只有M个文件数

而相对于之前Hash Based的Shuffle的话

就是M乘以R个数目

这样子的话它也会去减少了一个M倍

会至少减少M倍的总的文件数目

然后相对于Consolidate Shuffle的话

它也是需要C乘以R的总的文件数目

而一般情况下

C乘以R都是会大于这个M的

另外

我们看

这个是总的文件数目

我们再看同时打开的文件数目是多少呢

同时打开的文件数目

对于Sort Based Shuffle的话

它同时打开的文件数目是C个文件数目

就是说总共有多少个？

我就会有多少个Map Task在运行

一个Map Task只打开一个文件

所以说它只有C个文件

前无论是原始的这个Hash Based Shuffle

还是Consolidate Shuffle

它的同时打开的文件数目就是

C乘以一个R的总的文件数目

所以说仍然都是减轻了R倍的总的

file handlers

以及减轻了R倍的文件的缓存

所以说在本质上面能够得到非常大提升

就是说我总的文件数目是非常非常少的

而且我对内存的压力

又会减轻非常非常的多

所以这个就是Sort Based Shuffle

会相对于之前那些Shuffle

性能上面的一个极大的提升

之前讲完了所有的这些Hash Based的

Consolidate 的

然后还有Sort Based的Shuffle

我们现在来做一个总结

看一下具体整个的流程

它们是怎么样子的

对于Hash Based的Shuffle

我对于整个一个Executor来说

它里面有很多个Task

每一个Task去处理了一个

RDD里面的一个partation

所以说每一个partation

会对应一个Map Task

而一个Map Task

在Hash Based的Shuffle

这种模式下面

每一个Map Task都会去真成

Reduce个bucket

而一个bucket是以file的形式存在的

也就是说

每一个Map Task

都会生成Reduce个Outout File

这个Outout File就是Shuffle File

就是每一个Map Task

都会生成Reduce个Shuffle File

总的文件数目的话

它就是有多少个Map Task

那就会有Map Task个数目

乘以Reduce Task数目的乘积

对于每一个Executor就会生成这么多file

然后

在Reduce这边的话

每一个Reduce

就会去拿相应的这么一个file

去拿过来

去进行一个计算

这个就是Hash Based Shuffle

Consolidate Shuffle

在前面这一个Map阶段

其实是

在Map Task的阶段

就每一个Map Task

它仍然会去生成Reduce个Outout File

就是Reduce个Shuffle File

但是假设我这边是三个CPU

这个Executor只分配给了三个CPU

那么也就是说

同时它只能去处理三个partation

每个partation的话

它就会

假设这个Reduce Shuffle也是三的话

那么一个Map Task就会生成三个

Outout File

第一波的Map Task结束了之后

它就会第二波的这些partation

也会生成三个Map Task

每个Map Task仍然会把对应的这些

由Reduce的这些归类的这些数据

进行追加到前面的写了的bucket后面

就相当于我仍然是

这个Map Task

仍然写同样的三个文件

同样的三个file

只不过把不同的partation里面的数据

追加到了不同的对文件里面去而已

这个就是Consolidate Shuffle

在Reduce端的话

我仍然是

我有杜少个Reduce

每一个Reduce

它只会去拿对应的文件而已

这个是Consolidate的一个Shuffle

接下来我们再看Sort Based Shuffle

Sort Based Shuffle的话

每一个Map Task

它只会生成一个Outout File

比如说像这边的话

我这个Outout File

每一个Map Task

只会生成一个Outout File

只不过这个Outout File里面

它会有一个一个的FileSegment

一个FileSegment就相当于

之前抽象出来的一个bucket一样的

每一个FileSegment

会对应一个Reduce

Reduce Task

所以第一个FileSegment就到

Reduce Task1

那么我这个文件的第一个FileSegment

也会到这个Task1

它的第一个FileSegment也会到它

因为我这个是

比如说是按照

每一个Outout File是按照partation排序

但是具体我这个Reduce Task

应该要去拿多少数据

它是不知道的

所以我在Sort Based Shuffle里面

它会有一个index文件

所谓index就是说

我在Reduce过来拿你这个

Shuffle File里面的一个

FileSegment的时候

具体应该拿Segment的多少数据

因为Reduce我是不知道

我读你这个文件

拿Segment

这个Segment有多长

它的？是多少

它都不知道

所以说需要index去定位的

我在Map Task

在去写这个Outout file的时候

他就会生成一个index文件

告诉它

去表面我这个Shuffle File里面的

每一个Segment的？是多少

它的长度是多少

这个是通过index来实现的

所以每一个人Reduce Task

去读这个FileSegment的时候

它就去看这个index

就是说我这个FileSegment

是在这个Shuffle File的哪一部分

它的长度是多少

然后我去把它读出来

这样子的话就可以把

每一个它应该读的数据全部拿过来

到了Reduce端

我们一般都会用一个

假设我们是需要排序的话

我们去用一个

无论大？或者是小？的话

去做这么一个计算

比如说我从三个Map Task里面

拿到了三段FileSegment

这一段是Map Task1的

这一段是Map Task2的

这一段是Map Task3的

这三个FileSegment都是属于

Reduce Task1的

对于Reduce Task1

它怎么去处理这些FileSegment呢

因为这些FileSegment

这个里面它都是会进行排序

排序完了之后

我就会去拿一个小？

去 或者说大？

去看我这边第一个数据进行去比较然后

把最大的或者说最小的都去拿出来

然后最终我就生成了一个文件

这个文件的话

就是已经被排序好了

那么对于这个Reduce来说

它就做完了

这个就是Sort Based的Shuffle

大家好

我是英特尔大数据组的软件工程师

张天伦

今天我讲课的主题是

大数据流处理系统

主要介绍为什么需要流处理系统

包括流和流处理的概念

重点讲解流处理系统的容错性

最后介绍流处理系统的典型运算

我们已经生活在一个数据驱动的时代

数据无时无刻不在产生

来源多种多样

一个社交平台上

每秒可以产生6000条新消息

我们有数十亿台物联网设备

时刻在收集数据

一个搜索引擎每秒要处理

40000次请求

还有各种网站上如影随行的广告

用户每次浏览点击广告都会产生数据

而所有这些数据到达后台系统

系统在处理的时候

还会生成大量的系统日志

通过处理这些数据

我们能够发掘出其中蕴藏的价值

我们可以知道正在发生哪些热点事件

我们可以知道用户的喜好

给他们推荐音乐与书籍

我们可以找出疾病的根源

可以研发新药

我们可以创造出人工智能

那么我们要如何去处理这些数据呢

首先数据很多 非常多

多到不可能存在一台机器上

所以我们需要一个

分布式的数据处理系统

同时在我们的物理集群中

网络出错和机器宕机是正常现象

我们需要保证系统有良好的容错性

保证应用结果的正确性

再者用户需要及时的反馈

数据的价值是随着时间而降低

这就要求系统实时返回计算结果

不同应用对实时性的要求不同

图中从左到右

实时性的要求从高到低

第一类应用对实时性要求最高

通常要在毫秒级处理完数据

比如网站需要在用户

打开网页的几百毫秒内推荐广告

否则用户可能已经跳转到其它地方

再比如银行要在第一时间

发现用户的欺诈行为

中止交易 避免损失

第二类应用像通过监控摄像头

对嫌疑人车辆的排查布控

对突发公共事件的监测

需要在一定时间窗口内处理才有价值

不然嫌疑人逃逸了

事件过去了

分析结果就没有意义了

这类应用通常需要在

秒级到分钟级处理完数据

第三类应用对数据处理的

实时性要求最低

比如搜索引擎的索引更新和日志分析

越快处理价值越高

一般在分钟到小时级别

在典型的批处理系统中

数据需要预先按小时或天划分成批次

一次执行只处理一个批次

同一批的数据会存储在

一个分布式文件系统上

通过一个或多个Map-Shuffle-Reduce

过程处理数据

每个阶段都需要将中间结果写入磁盘

一旦某个阶段出错

可以从中间结果中快速恢复

而不用从头再来

批处理系统通常只能处理

预先划分好的数据

数据是前一个小时或者前一天的

因此不能保证实时性

而在流处理系统中

数据不会预先分成一个个批次

而是实时流入

连续不断 无穷无尽

系统则持续不断地

分布式处理流入的数据

实时返回结果

那它的容错性如何呢

下面我们会具体讲解

因为流是无穷无尽的

一个流处理系统必须保持在线

出错后能迅速恢复

流处理系统的容错性保证有三个层次

图中从上到下

保证由弱到强

因为物理集群的不稳定

所以消息很容易丢失

最弱的保证就是允许消息丢失

对于近似计算

比如统计每分钟的大致用户访问量来说

这并不会影响结果的正确性

再强一点 消息不能丢失

但是可以重复

此时系统要能检测到消息丢失

并且重放丢失的消息

像数据清洗这类应用

重复的无用数据还是会被过滤掉

但是不允许有用消息的丢失

最强的保证是消息不丢不重

比如搜索引擎按广告点击量

向广告主收费

如果消息重了

广告主会被多收费

如果消息丢了

搜索引擎就要蒙受损失

如何检测消息丢失呢

一般有两种做法

第一种是中心节点确认

图中起点A先发送一个

带有独特ID的消息

给中心确认节点 Acker

再把消息本身发送给 B

B收到之后会发送确认消息给Acker

同时告知它将消息发给了C

C收到B的消息后

也会通知Acker消息收到了

假如在一定时间内

Acker没有收到所有的确认消息

就会超时报错

第二种办法是上下游确认

A给B发消息

每条消息会带一个独特ID

B收到后会回复一个

带有该ID的确认消息

如果A没有收到对应的确认消息

则消息丢失

一般通过批量确认可以提高性能

比如每一百条消息确认一次

如果消息在丢失之后需要重放

一般有两种做法

一种是基于可重放队列

数据源的消息在进入流处理系统前

会先被写入一个可重放队列

一旦系统在处理过程中出错

可以从队列中重新读取之前的消息

另一种做法是从上游重放

各个节点将计算结果

写入到可持久化存储

如果下游节点出错

则上游节点从存储中读取出错消息

进行重放

那么如何去除重复消息呢

如果计算本身是幂等运算

即重复运算不影响结果

则无需去重

例如关系运算中的与运算和或运算

实际应用中如广告推荐

第二种做法是在接收端过滤

每条消息都有一个独特的ID

接收端收到消息会查询存储

是否已经收到过这个ID

如果没有就接收该消息

并将ID保存

如果存在

则为重复消息

直接丢弃

通常采用布隆过滤器

可以减少访问存储次数

提升性能

通过以上介绍

我们看到流处理系统

可以通过相应的算法

提供不同层次的容错性保证

从而满足了我们一开始提出的

大数据处理的三大要求

分布式数据处理

实时性和容错性

下面我们来介绍

流处理系统中的典型运算

这是一个网页搜索异常检测的应用

当用户进行网页搜索的时候

首先应用会对关键词

进行窗口聚合运算

统计其在一段时间内的搜索量

统计结果会发送给一个机器学习模块

构建出关键词的历史模型

知道它的正常走势

同时新的统计结果会和这个历史模型

进行比对

进行主键连接运算

过高或者过低都会触发异常通知

这中间主要涉及到两个运算

一个是窗口聚合运算

一个是主键连接运算

我们首先来看窗口聚合运算

一个消息从数据源产生

到进入流处理系统总会存在延迟

消息被处理的时钟

总是晚于消息产生的时钟

并且消息在传输过程中可能发生乱序

比如图中 7点钟的消息

到了7点15分才进入系统

7点08分的消息

会在7点12分的消息之后进入系统

这里我们定义两个时钟

一个是消息时钟

即消息产生的时间

另一个是系统时钟

即消息进入系统被处理的时间

我们通过一张坐标图

来看这两个时钟的差别

图中横坐标是消息时钟

纵坐标是系统时钟

由于存在延时

一条消息的系统时钟

总是大于消息时钟

因此图上所有的点都落在左上方

如果我们现在要统计

7点到7点10分的消息数量

我们不能在系统时间

为7点10分的时候输出结果

因为此时有些消息

还在网络传输的过程中

例如7点产生的消息

直到7点15才进入系统

那么我们什么时候才能输出

这个窗口的计算结果呢

通常我们会借助低水位线的概念

如图中的蓝色曲线

低水位线代表消息时钟的进度

它表示不会有更小时钟的消息

进入系统

如果有系统通常会直接丢弃

例如图中到系统时钟为7点20时

我们可以说7点10分之前的消息

都已经进入系统

这个时候我们就可以输出

7点到7点10分的计算结果

低水位线是单调递增的

它的划定通常基于经验

以及应用的容错性

假如应用不能容忍消息丢失

那么低水位线的划定就会保守一些

假如应用可以容忍少量消息的丢失

那么低水位线

就会更接近图中的蓝色虚线

我们来看个具体例子

现在从7点开始

我们要基于消息时钟

统计窗口大小为10分钟的消息数量

系统会在内存中维护多个时间窗口

比如当系统时钟为7点18分时

7点到7点10分的窗口里

有两条7点的消息

7点10分到7点20的窗口

有一条7点12分的消息

此时低水位线小于7点10分

表示消息时钟小于7点10分的消息

还有可能进入系统

比如7点08分的消息

此时不能输出7点

到7点10这个窗口的结果

当系统时钟为7点20时

低水位线等于7点10分

我们认为不会再有小于7点10分的消息

进入系统

可以输出7点到7点10分的统计结果

此时系统内存中

7点10分到7点20的窗口有三条消息

7点20到7点30的窗口有一条消息

它们的统计结果

在低水位线划过窗口之前都不能输出

这就是窗口聚合运算的基本算法

流处理系统的另一种典型运算

是主键连接

主要有流和表的连接

流和流的连接两种

流和表连接中

内存里已经有一张哈希表

这样当消息到达时

我们就去查询哈希表的对应主键

例如主键是搜索关键词

输入是张三的窗口统计结果

通过在哈希表中查找张三的模型

我们就能知道张三的统计结果

是否有异常

两条流连接又分两种情况

如果一条流比另一条流先到达

我们可以在内存中将先到达的流数据

添加到哈希表中

第二条流到达的时候

就和流和表的连接同理

如果两条流到达的顺序不确定

那么我们需要动态去构建两个哈希表

对于同一个主键

如果流1先到了

我们就把对应数据

添加到它的哈希表里

流2到的时候去查询流1的哈希表

如果流2先到了

就刚好相反

因为内存的容量总是有限的

先到的流

不能一直等待另一个流的对应消息

所以我们通常会基于一定时间窗口

进行缓存

过了时间窗口

则对应的数据会从哈希表中删除

总结一下

这节课中我们介绍了

流是连续不断无穷无尽的数据

为了能实时处理这样的数据

我们需要一个流处理系统

我们介绍了流处理系统

有三个层次的容错性保证

以及相应的算法

最后我们介绍了

流处理系统的两种典型运算

当然我们只是概述了

流处理系统的基本知识

大家可以通过前面的连接

和这里的参考文献进行更深入的学习

最后感谢我的前同事

Zhong, Xiang对这篇文档的贡献

谢谢大家

接下来我们讲一下

Spark Memory Management

内存管理在Spark里扮演着一个

比较重要的角色

我们常说Spark比Hadoop

在性能上可以提升10倍甚至百倍

很大一个原因是因为Spark

能够更加高效的去利用内存的优势

相对于Disk的这种优势

比如说Disk Spark内存在管理上

就是说它能够在Stage和Stage之间

或者说job和job之间

它会有一些RDD的这种cache调整

从而避免

能够去一些磁盘的写

以及磁盘的读

从而直接利用内存的读取

来提升这种效率

在Spark里面

内存主要分为三大块

第一块就是一个Storage Memory

Storage Memory主要是涉及到

比如说是像RDD的cache

然后还有broadcast

然后还有task result

这种数据的这种存放

那还有第二块的话就是一些

execution memory

像shuffle的

像？？？

像enjoy这一块内存

另外的话就是一些像use memory

比如说

或者我们这边说的other memory

其实就是抛开execution memory

和storage memory以外的

其他memory这一块的memory

就是我们所说的use memory

和other memory

那在对于Spark memory这一块来说的话

就是说它

因为随着Spark版本的演进

它会有一些不同的管理方式

那我们这边主要是讲

会涉及到一些传统的Legacy模式

另外的话会有一种统一的管理模式

叫Unified Memory Management

那在这个Legacy模式下面

首先我们讲这个Legacy模式

在Legacy模式下面

它的这个storage memory

默认情况下它是占60%

那在像execution memory状态

就占20%

然后由于Spark是运行在JVM之上的

那么我们大家都知道

在Java里面这种object

我们想要拿到一个object

它占用了多少内存相对来说

是比较困难的

在Spark里面

他会自己实现一个？？？

这样一个类矩 做这么一件事情

就像C里面的这种shuffle这种

但是这里面的区别就在于？？？

它能够拿到一个Java object

它只是一个大概的大小

就是说它不能拿到一个绝对准确的数值

这个object到底占用多少内存

它这个是拿不到的

所以说在Spark内存管理上面

它都会有一个safety fraction

就是说在storage memory这一块

那它的safety fraction

就是90% 0.9

最后虽然storage占了60%

比如我100G的这个JVM

这个JVM它总的内存大小可以达到100G

那么我的60个G就可以用作这个storage memory

但是从Spark管理的角度来讲

它只能够认为

你最多只能用到60乘以0.9

等于54G

那你如果超过54G的话

那你的storage就相当于用满了

那同样的呢对于safe这一块

就是execution memory来说的话

那它的memory fraction就是0.8

它能够用到80%

另外的20%它也是用作缓存区的

就是说它不能够绝对的能够知道

我这边用了多少

所以我需要一个相对安全的一个阀值

来限定 你只能用这么多

如果超过了的话我就认为

你已经全部用满了

那另外这个像storage这一块

我们可以看到这里面还有20%的unroll

这个所谓的unroll是什么意思呢

就是说我在spark里面

因为spark它是一个

？？的这种计算方式为主

比如说读出的数据

都是在dask上面的时候

那我读一条数据进来

就是？？？的形式

我读一条数据进来

我把它计算出来

我再读一条数据进来

我把它计算出来

假设这是一个partion

其实我并不知道这个partion有多大

那么我只能一条一条的读

直到我把它读完为止

我才知道我这个partion到底读掉了多少

然后我读完了之后

我计算出来

出来的这个结果是多少

到这个时候我才知道

但是如果说我没有unroll

那么我就说我要把这个？？

计算出来的result cache

那我就不断地往这个storage里去丢

这个结果就是

如果这个partion非常非常大的话

那么我这个memory根本放不下的话

那么我直接往这个内存里丢的话

非常容易产生out of memory

这种？？

就像内存报表

解决这个方式就是我设一个

20%的这个unroll的memory

那么我一条一条？？过来计算

如果我发现

已经你这个地方超过20%

假设同时有一个前提就是说

我这边这个storage已经用满

至少我这边至少我可以保证

至少有20%的进行unroll的空间

我可以让你直接计算在内存里面

然后如果20%用满了

那么我剩余的这块部分

就是直接往disk上面delete它了

那当然这边剩余

当然如果这一块计算在内存里面的话

这些内存同样也会放到disk上面

因为在Spark里面

一个partion就相当于一个block

它就只能要么放在内存里面

要么放在disk上面

当然如果这边内存没有用满的话

当然仍然可以继续放

反正至少我能够保证你20%可以unroll

但是如果说我这边已经

这边storage已经已经满了

那你如果又超过这20%的话

那我就把你直接？？到disk上面去

当然这个的前提是你storage labor是

memory and disk

那如果是你memory only

那如果是出现这种情况的话

就直接把你给扔掉了

那这个是Spark Memory Management

这种是一个Legacy remote的一个情况

那另外Legacy remote的情况需要说明的一点

就是说我的storage memory

我这60%当然是用户是可以

自己去通过一些参数去设定的

当然这个execution memory

同样是也是可以设定的

但是用户设定了之后它就定死了

就是我storage memory

如果我设定了60%

那你只能用60%的空间

你不可能占用更多的空间了

同样你这个execution memory

如果说你最开始设了20%

那你在运行的过程中

你只能用到20%

你不能用再多

不能用更多了

所以说它就是一个

比较固定死了的一个

memory的这个分配方式

那这个的话就是这个Legacy remote

它相对来说不那么灵活

就是说你需要用户去看

你这个里面

它到底需要多少storage memory

需要多少execution memory

这个是需要用户在这个application运行之前

你是需要有一个预测的

就是说你需要用户对这个应用

相对来说比较了解

对Spark来说比较了解

它差不多用多少内存

这个对用户的要求

是相对来说比较高的

这个就是Spark memory的这种

Legacy模式

像之前我们看到了

在Legacy运行方式下面

我的这个storage memory

和这个execution memory

它是只能够限定在

自家的一亩三分地里面

就是说我storage memory

我就只能用原来的60%

我execution memory仍然用原来的20%

那现在的话

这个unified的这个memory management

这是一种比较更新的

或者说更强大的一种

memory管理模式

它在这个里面的话就是说

我并不限定死

storage memory是多少

execution memory是多少

我只限定了你这个

storage memory加上execution memory

它的总数是等于这个

JVM的75%的内存

当然这个是默认值

这个用户同样是也可以改的

我先默认这两个加起来是75%

那像之前就是storage memory

定死的60%

然后execution memory就是20%

那相当于我原来的话80%的

总的memory是可以用这两块的

但是每人有且只有0.6

storage有且只有0.6

execution的话

execution memory有且仅有0.2

那像在这个Unified的这种

memory management下面

我这两个加起来

我最多默认的是75%

就是0.75的这个总的JVM内存

那另外user memory就像之前的

像other就抛开

execution storage memory之外的

就是25% 0.25

那另外的话它会有一个

reserved memory之中

我的整个JVM先要去扣除

这个比如说像system reserved memory

因为它是一个进程

system reserved memory

它先扣除掉了300之后

就乘以0.75以及这个0.25

对于Unified memory

Unified memory management来说

最大的一个亮点是

我的这个storage memory

和execution memory

它并没有限定死它能用多少

只不过虽然我默认的情况下

最开始初始值

我的storage memory

可以用这个0.75里面的50%

然后它也可以用50%

但是它并没有限定死

你storage memory就只能用到这个50%

比如说在Unified memory management情况下

如果说我storage memory

想要更多的内存的话

它也是可以的

那么我就会去借用这个

execution memory的内存

那么这条中间线就可以下去了

就可以往下移

同样的的反之

execution memory如果想要更多的memory

那么它可以想向storage借

这条线就可以往上

execution memory用的

就可以多于50%

这是同样的道理

所以说

这个Unified memory management的话

就是这个统一的管理的话

它就更灵活

可以更高效的

让Spark去用

这种execution这边也好

storage那边也好

可以用更大的内存

或者如果说其中有些free的话

其中有些内存是空闲状态的话

我就可以更充分的去利用这些

空闲的内存

就是这两块里面

这75%里面

它更多空闲内存是可以取用掉的

举个最简单的例子就是说

我假设我的一些

storage memory这一块

比如说我假设在一个应用程序里面

我并没有涉及到RDD的cache

然后我只有非常少量的

比如说像task result非常少量的

那个broken task

那样子我的storage这一块

我就用的非常非常少

比如说可能

5%的内存都甚至不需要用

那那样子的情况下的话

我在execution memory

就可以非常多的去把它占用掉

就说如果我execution memory

我是有需求

用很多memory的话

我就可以把storage memory

这一块的内存全部用起来

那就相当于我的execution memory

就可以达到几乎

将近70%

甚至以上的这个整个的memory

那样子的话

我的内存的利用率就更大了

那像之前Legacy模式下的话

如果说指定我这边是不是只需要

甚至是5%的memory

或者更少的memory

是需要用户你指定

并且用户需要知道的这种情况下

而现在的话

就是说

它会自动的发现

你这边storage memeory

其实这些memory都没有用

还是都空着的话

那我们就可以去用过来

就是现在它是这种模式

当然这个里面具体

这边怎么能够向上

或者说这条线怎么能够向下

这个里面的这种判断逻辑

以及在什么样子的情况下

它能够向上

以及在什么样子的情况下

它不能够向上

这里面会涉及到像这样一些

相对复杂的逻辑

在这里不再耳诉

反正总之这个Unified memory

它就能够更加自动化的去

调准这两块的memory

让它们更加充分的利用当前Spark

可用的这种free的memoruy

然后防止

就是说更多的

更大程度上去减小

不管shuffle也好

也不管这边cache也好

它能够引入的这种disk的IO操作

更好的避免这种IO

更多的利用memory

从而更进一步的实现

内存的更高效的管理

以及更高效的利用率

这就是spark的memory management

主要是一个Legacy remote

和unified remote

总结下来就是unified remote

可以更灵活

而Legacy remote需要用户

可以更专业的去考量

他自己的应用程序行为是什么样子的

他需要提前去进行设置一些

重要的一些参数

像现在Unified的话

用户就不需要做这种工作

它可以自动的

更加智能的去实现这一个行为

今天主要就跟大家介绍

Spark相关的这些内容

今天就讲到这边

谢谢大家

大家好

我是来自Intel大数据团队的Li Zhichao

主要从事分布式计算

以及数据分析方面的工作

今天主要给大家带来

Spark SQL的一些入门介绍

希望能够起到一个抛砖引玉的作用

今天我们会覆盖这么几方面内容

希望达到的效果是

大家在听完之后

可以独立的去运行Spark SQL

然后对里面整一套流程的主要概念

有所了解

一开始我们会从一个最简单的

类似hello world的例子开始

接着是HIVE跟sparksq的结合

这也是在实践的过程当中

比较常见的内容

然后是sparksql对不同data format支持

最后 我们会走一遍sparksql

内部的实现流程

希望对大家以后深入了解

sparksql有所帮助

当然可能准备的比较仓促

如果有什么不到位的地方

希望大家指正

对于一个开源项目来说

它更新速度往往都是很快的

所以说要了解它

最方便的方式就是从它的官网开始

我们会打开官网链接

然后去走一遍

看它里面说什么东西

这个是Spark的主页

然后主页上面

它主要是对Spark的一些性能

会有比较详细的介绍

比如说Spark的速度是Hadoop的很多倍

这里有一个很basic图说

Spark的确是比Hadoop快很多

主要我们是看这张图

这张图 Spark项目本身

它是由很多个组件构成的

最下面那个是SparkCal

SparkCal就是定义了很多

通用的API

比如说RDD的概念

还有一些通讯机制等等

基于这个SparkCal的实现

在它的上面

实现了很多其他的组件

Spark SQL、Spark Streaming

继续学习算法、图计算等等

我们要看的是Spark SQL

Spark SQL也是整一个Spark

最活跃的一个项目之一

我们点击进去看一下

好 我们走一遍这边的介绍

一开始它会介绍这边

Spark SQL它提供了很多丰富的API

比如说java、Scala、Python等等

它可以很方便地

跟现有的Spark程序进行结合

通过data source API的一个抽象

它可以让我们可以以一个

统一的方式去访问不同的dataframes

当然了 最重要的一点

它对HIVE的兼容性

Spark SQL跟HIVE是兼容的

HIVE是什么呢

HIVE是开源的一个

也是分布式的SQL计算引擎

比较重要的一个区别是

HIVE是基于MapReduce的

Spark SQL肯定是基于Spark的

这张图上我们可以看到一些

橘黄色的区域

橘黄色区域里面它列举了

Meta Store、HiveQL、UDFs等等

这些都是HIVE里面一些已经存在的unit

但Spark SQL可以直接利用而访问它

最后是一些已有的BI工具

Spark SQL可以通过JDBC

或者ODBC的方式

可以很好的去连接上已有的BI工具

并且融合使用它

好 接下来我们看一下

如果要用Spark SQL的话

我们应该从哪里开始

从代码层面上看

他总共有两种context

一个是HiveContex

一个是SQLContext

刚开始接触的时候

都会觉得很疑惑

我应该选择哪一个

我们其实从这里可以很清楚的看到

HiveContex其实是sqlcontext的一个子类

相当于超集

它提供了额外的对hive的支持

所以在生产环境当中

用HiveContext居多

然后接下来它有另外三个入口

spark-shell、spark-sql

还有thrift server

通过spark-shell

主要是你可以直接输入SQL代码

然后去用编程的方式去操作Spark SQL

spark-sql这个接口

主要是你可以直接输入SQL语句

然后就可以很方便的去执行了

对于thrift server而言

它的好处是

比如说你现在有一些现有的BI工具

一些JDBC的客户端

比如说beeline

你已经用它去对现有的

一些传统数据库进行分析了

你也想用同样的工具去

操纵Spark SQL怎么办

你就可以把它连上thrift server

通过JDBC的方式连接就可以操作了

这是它带来的好处

这个是一个类似于hello world的例子

一个比较简单的SQL例子

比如说我们现在有一个people.txt

这样的一个文件

很简单

它里面只有三行

然后每一行两列

第一列是人名

第二列是年龄

怎么去把这个txt文件

把它转换成一个SQL能够识别的表呢

简单而言有两种方式

一种你可以通过编程的方式去把

每一行数据切分开

然后再用case class的方式

去定一些scheme

再把两块数据进行融合

就可以变成一张表

另外一种就是用HIVE提供的一些

CREATE TABLE的语句

直接去create一张table

然后再用load这样的

另外一条语句去把txt文件

load进表里面

两个方式

我们看一下

通过编程的方式是怎么去做的

首先就像前面介绍的一样

我们会有一个context选择

这里选择了sqlcontext

因为只是简单的测试

定义了context之后

我们会导入一些隐式转换函数

隐式转换函数的作用后面会介绍

然后再定义了一个person的case class

主要是描述一下我这张表

应该长什么样的

比如说我这张表应该会包含两列

第一列是人名

它的类型是字符串

第二列是年龄

它的类型是整型

然后这边就是一个编程的过程

把一个txt文件

把它转换成一个people类型的RDD

代码应该比较简单

就是先每一行根据逗号去split开

然后再把它映射成一个

person的case class

之后我们得到的是一个

people类型的RDD

这样的RDD

在结合我们之前的隐式转换函数

它会隐式的把

people类型的RDD转换成一个

DataFrameHolder

DataFrameHolder就有一个

to DataFrameHolder的一个函数

通过这个函数调用我们最后返回的是一个

DataFrame

我们用people来命名它

一个people的DataFrame

拿到这个DataFrame之后

我们就可以registerTempTable

TempTable是什么意思呢

就是说

我这只是一个临时的表

当我这个context关闭之后

你再重新打开

这张表是看不见的

一般来说

如果测试什么的

你可以直接调用registerTempTable

这样的方式

另外 如果你用

HIVEcontext

它启用另外的接口

你可以用saveAsTable这样的方式

可以把整张Table把它序列化在磁盘上面

下一次你再重新打开context的时候

这张表还是一样存在的

当表已经创建好之后

我们怎么去访问表里面的数据呢

那注意最简单的方式就是像传统的一样

我们直接执行一条sql语句

然后它就会返回结果

拿到了对应的结果之后呢

我们可以通过两种方式去访问它

一种是通过index

比如说index是从零开始的

我想访问取人名

那就index 0

可以拿出人名 反应出来

另外你也可以通过名字

就有field的名字

比如说这个field叫name

你也可以通过

getAsStringname这种方式

去把它拿出来

可能这种方式

有没有觉得会比较别扭一点

就是说其实它返回的

从简单的理解

它是一个ROW的集合

你拿到的是一条一条的ROW

就是说它里面给我们提供的额外信息

并不是特别多

但是在新的版本里面

会有更好的抽象

比如说它就返回一个person的集合

你就可以很好的向

利用case class的方式去

访问里面的字段了

有兴趣的读者可以去看一下

最新版本的DataSet这样的API

这里讲得是DataFrame

然后这里我提供了一个比较完整的代码

在听完这个课程之后

大家可以去试一下这个完整代码

中国高水平大学慕课联盟官方网站

简体中文

© 版权所有 2015 上海交通大学慕课研究院

战略合作：

沪ICP备06018291号

客服电话：4008296199

QQ 客服：[2782661530](tencent://message?v=3&uin=2782661530&site=www.cnmooc.org&menu=yes)

关注微信：[sjtucnmooc](javascript:void(0))

联盟

[服务条款](http://www.cnmooc.org/home/moocService.mooc)

[关于我们](http://www.cnmooc.org/home/about.mooc)

[联系我们](http://www.cnmooc.org/home/contact.mooc)

帮助

[常见问题](http://www.cnmooc.org/home/helpCenter/0/defaultmenu.mooc)

[学生手册](http://www.cnmooc.org/home/helpCenter/1/defaultmenu.mooc)

[证书查询](http://www.cnmooc.org/portal/cert/searchCert.mooc)

更多

[通知通告](http://www.cnmooc.org/home/sysBulletin/show-1.mooc)

[新闻动态](http://www.cnmooc.org/home/news/show-1.mooc)

[媒体报道](http://www.cnmooc.org/home/media-1.mooc)

刚才说到了spark sql和Hive的交互

这里画了一个大概的对比图

总体的意思是说

它们两个总体构成是差不多的

都有parser

有各种plan的转化

除了一个是基于mapreduce的

一个基于spark的

还有一些in memory的cache的区别

但这里有一点要说的是

spark sql

是可以直接访问HIVE的metastore的

这两个是兼容的

也就是说相当于它可以直接读取

HIVE的数据

Metastore

它主要是存储了一些 Table信息

还有一些 partition的信息

现在在业界大家都这么做

因为很多已有的数据存储

都是用HIVE去存储

那当它们想要迁移到spark sql的时候

就需要用spark sql直接去访问

HIVE Architecture的数据了

一般的做法是在spark里面

把metastore的访问信息配置好

就可以直接访问HIVE的表和

里面的内容了

我们这里会给个live demo

然后去看一下怎么去配置spark sql里面

hivemetastore的信息

然后怎么去体验一下

这两个之间是怎么去交互的

我们具体看一下怎么去配置

让spark sql和Hive之间能够相互访问

其实最主要的一点就是

怎么去让它们指向同样的metastore

左边这里有两个？

左边那个是spark安装目录

我们移到了confs这样的文件夹下面

里面主要是spark的一些配置信息

我们待会主要看的是

hive-site.xml这样的内容

右边这个是hive的安装目录

我们也看一下

里面也是有同样的一个hive-site.xml

这样的信息

我们打开它

看一下

hive这边我们已经把它

最主要的一点配置就是

把它的metastore访问指向了mysql

这样的一个地址

同样的在spark这边

也是通过hive-site xml这样的配置

把它指向了跟hive一样的

mysql的metastore存储地址

这两个都配置好之后

我们看一下继续操作

感受一下

先看一下现在有没有table这张表

好 没有table这张表

hive里面也是一样的

都没有table这张表

我们会首先在hive里面创建

people这张表

然后再在spark里面把数据插进去

这样就可以证明两者之间是可以

交互使用的

现在people这张表已经创建好了

那我们再在spark里面看一下

它也访问到了people这张表

那我们在spark里面把数据加载进去

注意到这里的加载语句

都是spark-sql的语句

也就是说在spark里面

可以无缝的去使用

hive里面的sql

当在spark里面把数据加载好之后

在hive也是可以把它select出来的

因为指向的都是同样的metastore

好 看看spark

好的 这是一个简单的例子

我们接下来看一下下面的内容

经过两个demo之后

我们接下来看一下Dataframe

这样的单元

Dataframe

其实在demo里面已经用过了

它其实简单的理解可以成为一个

含有丰富DSL操作的一个表的抽象

它是从R或者Pandas里面借鉴来的概念

这里我给了几个链接

里面包含了对Dataframe的一些详细介绍

还有一个link的链接

ling就是？？里面一个概念

其实它跟frame里面DSL很像

就是说我们为什么要用DSL这样的方式去

操作数据

为什么用它

那其实光从文字上来说

我们很难去感受DSL

跟直接写SQL语句有什么区别

我们接下来会给一个比较简单的例子

去看一下

这里有两部分

第一部分就是用传统的

SQL的语句的方式去选择

做一些projection

下面这部分就是用Dataframe里面

DSL的操作

我们也可以看到

比如说我们有一个people的Dataframe之后

我们可以先.where

就是设定一些条件

那条件之后

我们还可以继续跟着限定条件

相当于一个链式操作

一直选择下去

那它的好处呢 可能

在有的时候会更加灵活

去跟已有的代码组合起来

再去操作方面也会更加灵活

在现在很多在ML里面的很多API

都是基于Dataframe去进行操作了

因为它是比RDD

一个更加高层次的一个抽象

可以支持更多的方面的语法糖

这里是有一个完整的DSL的例子

大家课后可以去看一下

它可以是一种列式存储的一种format

列式存储是大数据领域里面

比较常用的格式

而parquet是里面的其中一个代表

Spark对parquet格式是有着很好的支持

你可以很方便的去把dataframe

存储成parquet格式

也可以把parquet格式反过来

把它转成一个dataframe去操作

那这里是给一个链接

去介绍parquet的

大家可以去看一下

下面有一张五颜六色的图

主要是给大家介绍一下列式存储

给大家一个直观的印象

列式存储跟行式存储的区别

比如说我们这里有一个简单的表

表里面有三列

A、B、C 3列

按ROW去存储 按行去存储的话

是怎么做呢

我们先存A1、B1、C1

然后再存A2、B2、C2

这样的按行去存储

连续的存起来

如果是按列去存储应该是怎么做呢

是把A1、A2、A3这一列存好了

再存B1、B2、B3这样的操作

列式存储

为什么要用列式存储呢

它有什么好处

举个例子

我们一般的表都是比较宽的表

比如说有200列

但是我们平时所需要的操作

可能我只需要其中的10列

如果按行数存储的话

我们需要把所有的列都取出来

因为从它存储的方式看来

我们就需要把它所有的列都取出来

但是列式存储

我们只需要取我们需要的列

这是好处之一

另外呢

列式存储

它把所有相同类型的数据都放一起了

比如说

A这列比如说它都是int类型

那么我们就可以针对同类型的数据

去选择编码和压缩算法

同时它会更好的支持向量计算等等

我们来看一下在spark sql里面

怎么去对parquet进行操作

这里有一个完整的代码

大家可以去看一下课后

然后对于dataframe

对parquet的读取其实很简单

首先我们有一个

就像前面的方式

拿到一个person类型的RDD

然后通过隐式转换

在.toDF的方式拿到一个dataframe

拿到dataframe之后

我们就可以write.format

然后传入parquet

就是说我们是parquet格式的

然后就可以把它存储起来了

还有另外一个额外的API

跟它类似的API

就是.parquet

其实这两个API是一样的

只是一个语法糖的区别

比如要读取怎么办呢

就是.read.parquet就可以了

很简单

这是通过所谓的DSL的操作

去访问parquet

如果是通过传统sql怎么办呢

它在create table的时候我们也会指定

比如说我们可以指定说

我们的table用的是parquet格式的

USING parquet

然后在path里面把parquet的路径

把它指定进去就可以了

之前我们提到spark对各种区域格式

是有很好的支持的

通过data sources api的抽象

我们可以以一个比较统一的方式

去访问不同的格式的数据

这里有个链接

主要是对data sources api的详细介绍

像这种databricks它有一个blog

大家可以去看一下

里面有很多关于spark的一些介绍

对于Built-in的类型

就是说内置的类型

不需要你加载额外的包就能支持的类型

它有Json、parquet、S3、HDFS等等

内置格式怎么访问呢

这里给了一个具体的例子

比如说访问Json的时候

在format里面指定是Json就可以了

基本上是统一的格式去访问不同的format

对于External就是说

我们需要加载额外的包去操作它

比如说CSV、Cassandra等等

我们都需要加载额外的包去支持它

下面我们会给一个具体的例子

去怎么加载额外的包

去访问这些不同的格式

刚才已经提到CSV格式不是内置的

但是它已经有外部包

去支持这样的data source

我们看一下这里给了一个链接

有一个spark-csv的这样的一个

External的data source api

我们可以在Maven里面

把这句话加载进来

然后再具体访问的时候只需要把

class的全称指定起来

然后就可以正常使用

去解析CSV了

然后这里有一个比较有用的链接

叫spark-packages.org

里面会包含很多spark相关的一些

有用的？包

大家有空可以去看一下

中国高水平大学慕课联盟官方网站

简体中文

© 版权所有 2015 上海交通大学慕课研究院

战略合作：

沪ICP备06018291号

客服电话：4008296199

QQ 客服：[2782661530](tencent://message?v=3&uin=2782661530&site=www.cnmooc.org&menu=yes)

关注微信：[sjtucnmooc](javascript:void(0))

联盟

[服务条款](http://www.cnmooc.org/home/moocService.mooc)

[关于我们](http://www.cnmooc.org/home/about.mooc)

[联系我们](http://www.cnmooc.org/home/contact.mooc)

帮助

[常见问题](http://www.cnmooc.org/home/helpCenter/0/defaultmenu.mooc)

[学生手册](http://www.cnmooc.org/home/helpCenter/1/defaultmenu.mooc)

[证书查询](http://www.cnmooc.org/portal/cert/searchCert.mooc)

更多

[通知通告](http://www.cnmooc.org/home/sysBulletin/show-1.mooc)

[新闻动态](http://www.cnmooc.org/home/news/show-1.mooc)

[媒体报道](http://www.cnmooc.org/home/media-1.mooc)