Spark Sql

2018年2月10日 15:59

概述

Spark为结构化数据处理引入了一个称为Spark SQL的编程模块。它提供了一个称为**DataFrame（数据框）**的编程抽象，DF的底层仍然是RDD，并且可以充当分布式SQL查询引擎。

SparkSQL的由来

SparkSQL的前身是Shark。在Hadoop发展过程中，为了给熟悉RDBMS但又不理解MapReduce的技术人员提供快速上手的工具，Hive应运而生，是当时唯一运行在hadoop上的SQL-on-Hadoop工具。但是，MapReduce计算过程中大量的中间磁盘落地过程消耗了大量的I/O，运行效率较低。

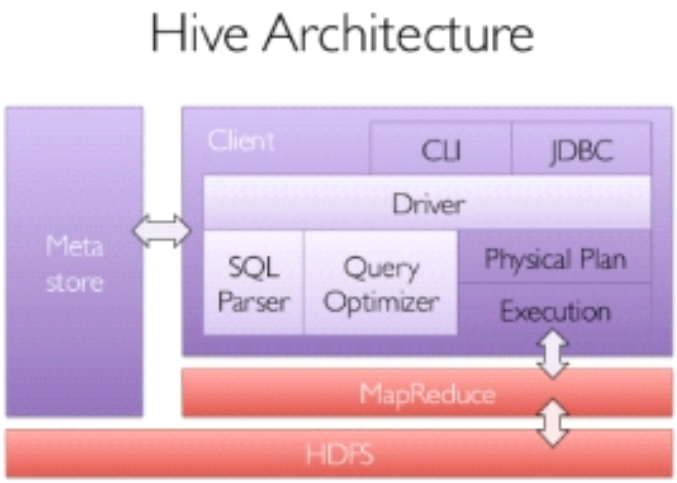
后来，为了提高SQL-on-Hadoop的效率，大量的SQL-on-Hadoop工具开始产生，其中表现较为突出的是：

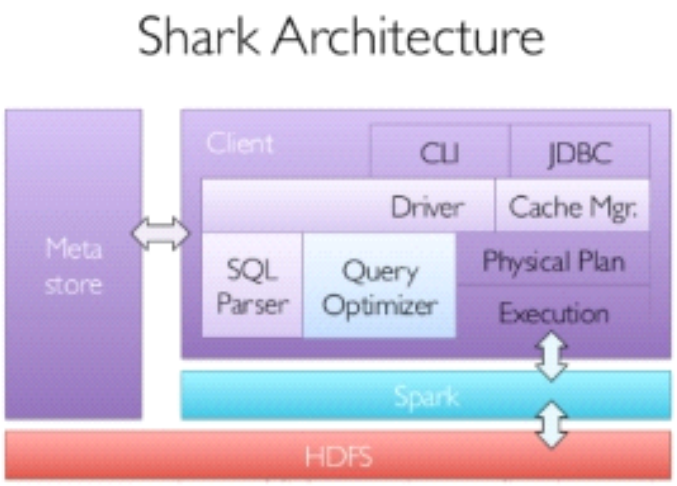
1）MapR的Drill

2）Cloudera的Impala

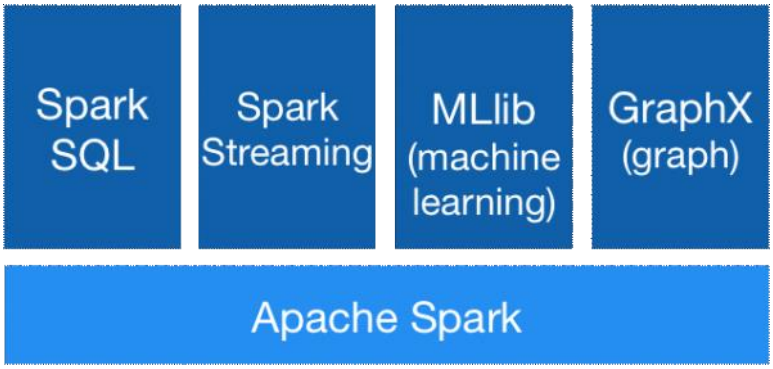
3）Shark

其中Shark是伯克利实验室Spark生态环境的组件之一，它基于Hive实施了一些改进，比如引入缓存管理，改进和优化执行器等，并使之能运行在Spark引擎上，从而使得SQL查询的速度得到10-100倍的提升。





但是，随着Spark的发展，对于野心勃勃的Spark团队来说，Shark对于hive的太多依赖（如采用hive的语法解析器、查询优化器等等），制约了Spark的One Stack rule them all的既定方针，制约了spark各个组件的相互集成，所以提出了sparkSQL项目。



SparkSQL抛弃原有Shark的代码，汲取了Shark的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive兼容性等，重新开发了SparkSQL代码。

由于摆脱了对hive的依赖性，SparkSQL无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便。

2014年6月1日，Shark项目和SparkSQL项目的主持人Reynold Xin宣布：停止对Shark的开发，团队将所有资源放SparkSQL项目上，至此，Shark的发展画上了句话。

SparkSql特点

1）引入了新的RDD类型SchemaRDD，可以像传统数据库定义表一样来定义SchemaRDD

2）在应用程序中可以混合使用不同来源的数据，如可以将来自HiveQL的数据和来自SQL的数据进行Join操作。

3）内嵌了查询优化框架，在把SQL解析成逻辑执行计划之后，**最后变成RDD的计算**

为什么sparkSQL的性能会得到怎么大的提升呢？

主要sparkSQL在下面几点做了优化：

1）内存列存储（In-Memory Columnar Storage）

SparkSQL的表数据在内存中存储不是采用原生态的JVM对象存储方式，而是采用内存列存储，如下图所示。

该存储方式无论在**空间占用量**和**读取吞吐率**上都占有很大优势。

对于原生态的JVM对象存储方式，每个对象通常要增加12-16字节的额外开销（toString、hashcode等方法），如对于一个270MB的电商的商品表数据，使用这种方式读入内存，要使用970MB左右的内存空间（通常是2～5倍于原生数据空间）。

另外，使用这种方式，每个数据记录产生一个JVM对象，如果是大小为200GB的数据记录，堆栈将产生1.6亿个对象，这么多的对象，对于GC来说，可能要消耗几分钟的时间来处理（JVM的垃圾收集时间与堆栈中的对象数量呈线性相关。显然这种内存存储方式对于基于内

存计算的spark来说，很昂贵也负担不起）

SparkSql的存储方式：对于**内存列存储**来说，将所有原生数据类型的**列采用原生数组来存储**，将Hive支持的复杂数据类型（如array、map等）先序化后并接成一个字节数组来存储。

此外，基于列存储，每列数据都是同质的，所以可以数据类型转换的CPU消耗。此外，可以采用高效的压缩算法来压缩，是的数据更少。比如针对二元数据列，可以用字节编码压缩来实现（010101）

这样，每个列创建一个JVM对象，**从而可以快速的GC和紧凑的数据存储**；额外的，还可以使用低廉CPU开销的高效压缩方法（如字典编码、行长度编码等压缩方法）降低内存开销；更有趣的是，对于分析查询中频繁使用的聚合特定列，性能会得到很大的提高，原因就是这些列的数据放在一起，更容易读入内存进行计算。

行存储 VS 列存储

2017年11月29日 19:04

概述

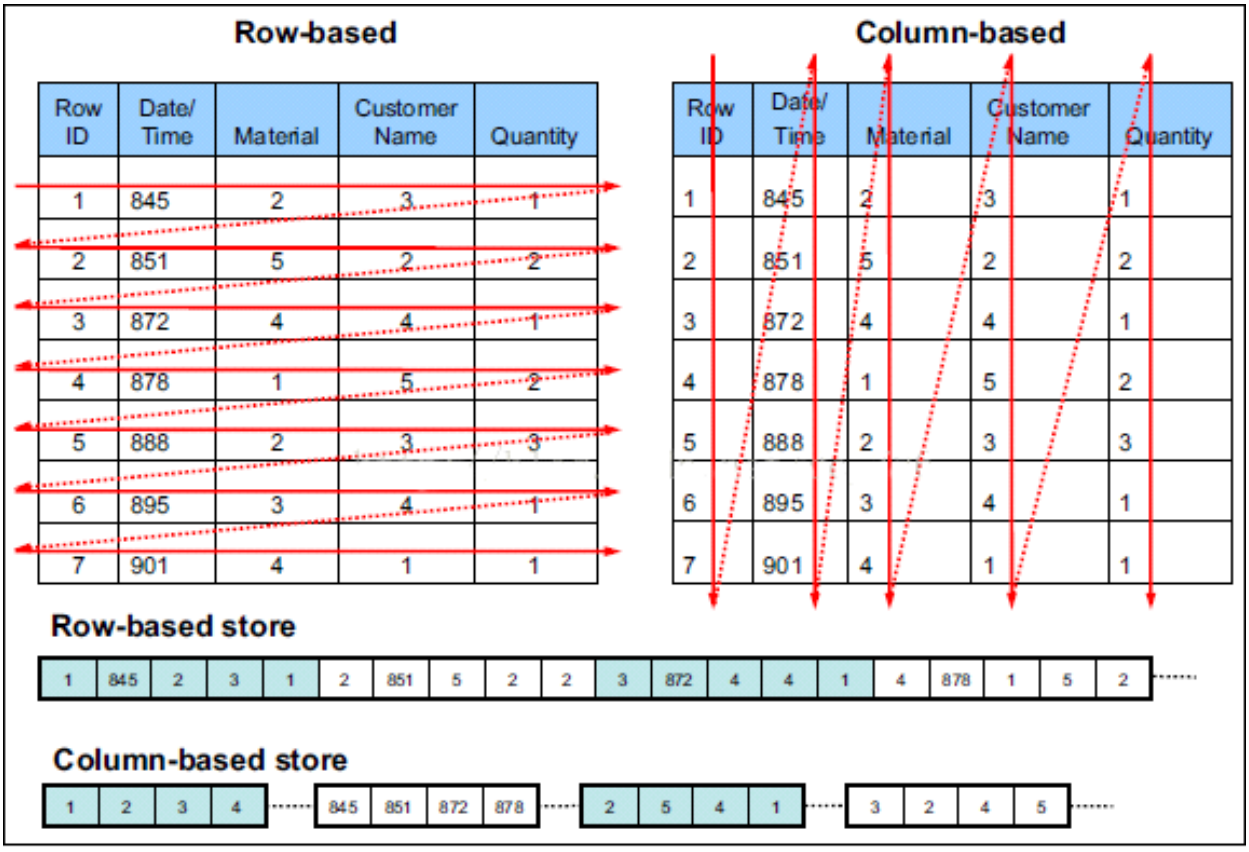
目前大数据存储有两种方案可供选择：行存储（Row-Based）和列存储（Column-Based）。业界对两种存储方案有很多争持，集中焦点是:谁能够更有效地处理海量数据，且兼顾安全、可靠、完整性。从目前发展情况看，关系数据库已经不适应这种巨大的存储量和计算要求，基本是淘汰出局。在已知的几种大数据处理软件中，Hadoop的HBase采用列存储，MongoDB是文档型的行存储，Lexst是二进制型的行存储。

什么是列存储？

列式存储(column-based)是相对于传统关系型数据库的行式存储(Row-basedstorage)来说的。简单来说两者的区别就是如何组织表：

Ø Row-based storage stores atable in a sequence of rows.

Ø Column-based storage storesa table in a sequence of columns.



从上图可以很清楚地看到，行式存储下一张表的数据都是放在一起的，但列式存储下都被分开

保存了。所以它们就有了如下这些优缺点对比：

在数据写入上的对比

1）行存储的写入是一次完成。如果这种写入建立在操作系统的文件系统上，可以保证写入过程的成功或者失败，数据的完整性因此可以确定。

2）列存储由于需要把一行记录拆分成单列保存，**写入次数明显比行存储多（意味着磁头调度次数多，而磁头调度是需要时间的，一般在1ms~10ms)**，再加上磁头需要在盘片上移动和定位花费的时间，实际时间消耗会更大。所以，行存储在写入上占有很大的优势。

3）还有数据修改,这实际也是一次写入过程。不同的是，数据修改是对磁盘上的记录做删除标记。行存储是在指定位置写入一次，列存储是将磁盘定位到多个列上分别写入，这个过程仍是行存储的列数倍。所以，数据修改也是以行存储占优。

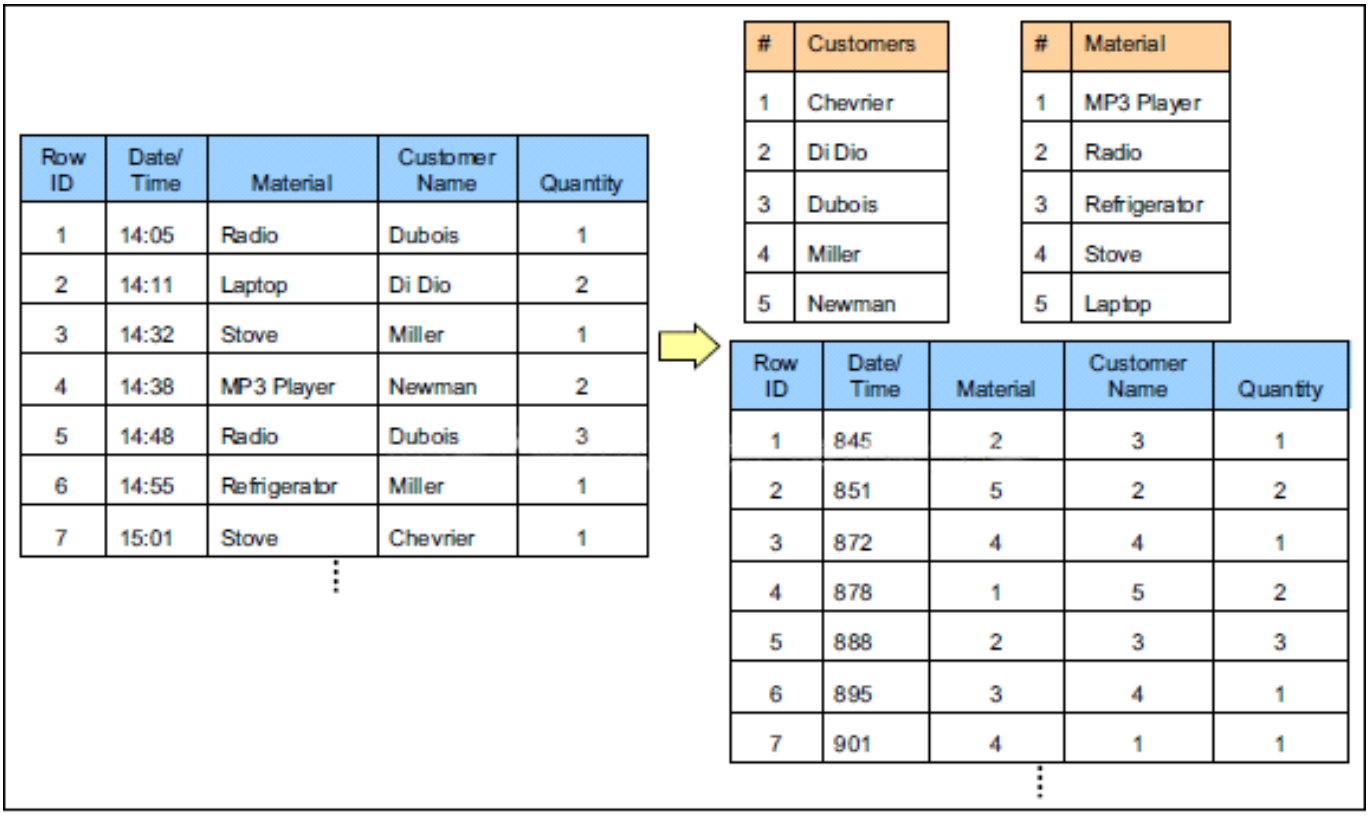
在数据读取上的对比

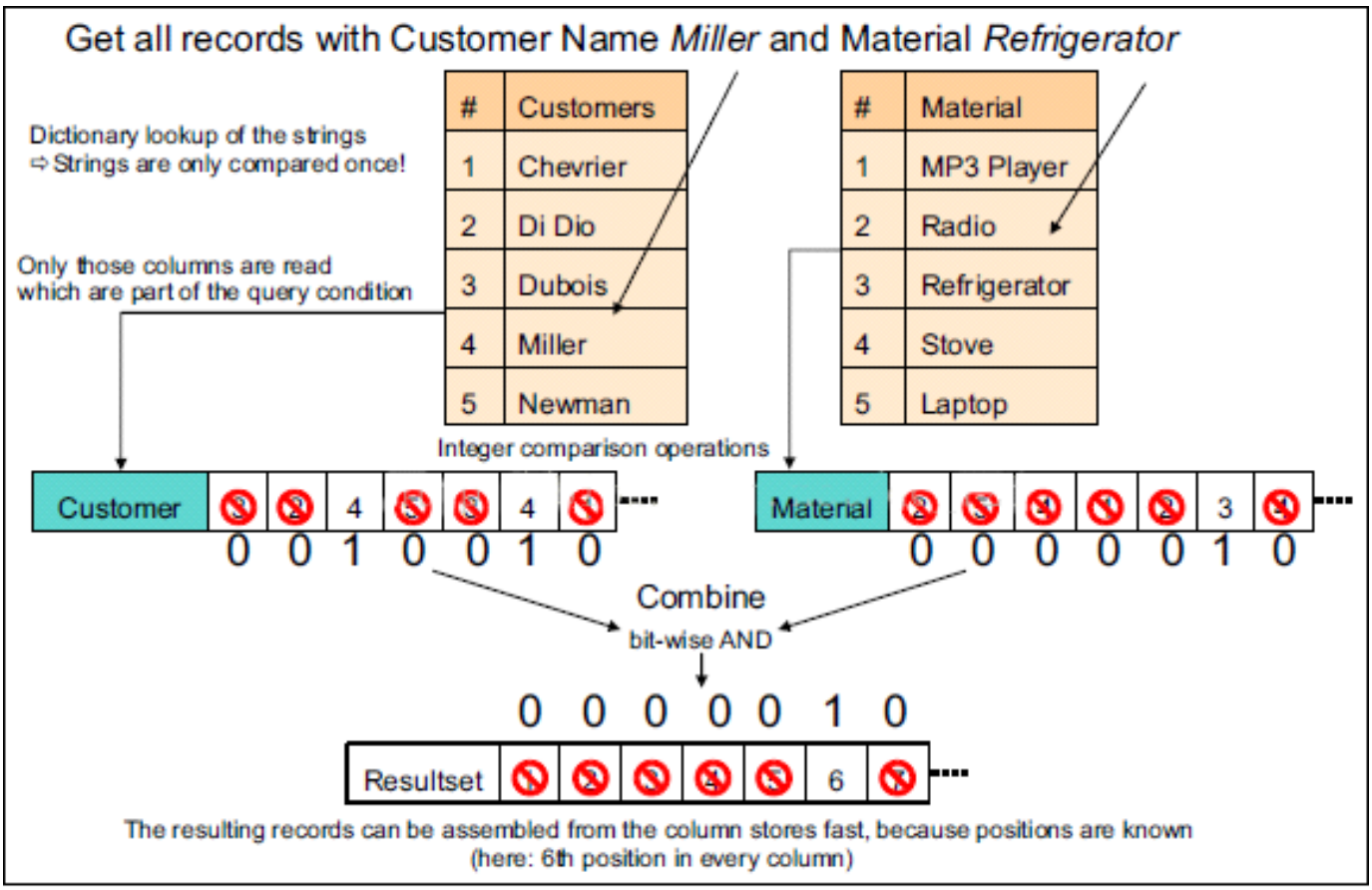
1）数据读取时，行存储通常将一行数据完全读出，如果只需要其中几列数据的情况，就会存在冗余列，出于缩短处理时间的考量，消除冗余列的过程通常是在内存中进行的。

2）列存储每次读取的数据是集合的一段或者全部，不存在冗余性问题。

3）两种存储的数据分布。由于列存储的每一列数据类型是同质的，不存在二义性问题。比如说某列数据类型为整型(int)，那么它的数据集合一定是整型数据。这种情况使数据解析变得十分容易。相比之下，行存储则要复杂得多，因为在一行记录中保存了多种类型的数据，数据解析需要在多种数据类型之间频繁转换，这个操作很消耗CPU，增加了解析的时间。所以，列存储的解析过程更有利于分析大数据。

4）从数据的压缩以及更性能的读取来对比





优缺点

显而易见，两种存储格式都有各自的优缺点：

1）行存储的写入是一次性完成，消耗的时间比列存储少，并且能够保证数据的完整性，缺点是数据读取过程中会产生冗余数据，如果只有少量数据，此影响可以忽略;数量大可能会影响到数据的处理效率。

2）列存储在写入效率、保证数据完整性上都不如行存储，它的优势是在读取过程，不会产生冗余数据，这对数据完整性要求不高的大数据处理领域，比如互联网，犹为重要。

两种存储格式各自的特性都决定了它们的使用场景。

列存储的适用场景

1）一般来说，一个OLAP类型的查询可能需要访问几百万甚至几十亿个数据行，且该查询往往只关心少数几个数据列。例如，查询今年销量最高的前20个商品，这个查询只关心三个数据列：时间（date）、商品（item）以及销售量（sales amount）。商品的其他数据列，例如商品URL、商品描述、商品所属店铺，等等，对这个查询都是没有意义的。

而列式数据库只需要读取存储着“时间、商品、销量”的数据列，而行式数据库需要读取所有的数据列。因此，列式数据库大大地提高了OLAP大数据量查询的效率

OLTP　OnLine Transaction Processor 在线联机事务处理系统（比如Mysql，Oracle等产品）

OLAP　OnLine Analaysier Processor 在线联机分析处理系统（比如Hive Hbase等）

表 4.1　OLTP 系统与 OLAP 系统的比较

特征	OLTP	OLAP
特性	操作处理	信息处理
面向	事务	分析
用户	办事员、DBA、数据库专业人员	知识工人（如经理、主管、分析人员）
功能	日常操作	长期信息需求、决策支持
DB 设计	基于 E-R，面向应用	星形/雪花、面向主题
数据	当前的、确保最新	历史的、跨时间维护
汇总	原始的、高度详细	汇总的、统一的
视图	详细、一般关系	汇总的、多维的
工作单元	短的、简单事务	复杂查询
访问	读/写	大多为读
关注	数据进入	信息输出
操作	主码上索引/散列	大量扫描
访问记录数量	数十	数百万
用户数	数千	数百
DB 规模	GB 到高达 GB	≥TB
优先	高性能、高可用性	高灵活性、终端用户自治
度量	事务吞吐量	查询吞吐量、响应时间

注：该表部分基于 Chaudhuri 和 Dayal[CD97]。

2）很多列式数据库还支持列族（column group，Bigtable系统中称为locality group），即将多个经常一起访问的数据列的各个值存放在一起。如果读取的数据列属于相同的列族，列式数据库可以从相同的地方一次性读取多个数据列的值，避免了多个数据列的合并。列族是一种行列混合存储模式，这种模式能够同时满足OLTP和OLAP的查询需求。

3）此外，由于同一个数据列的数据重复度很高，因此，列式数据库压缩时有很大的优势。例如，Google Bigtable列式数据库对网页库压缩可以达到15倍以上的压缩率。另外，可以针对列式存储做专门的索引优化。比如，性别列只有两个值，“男”和“女”，可以对这一列建立位图索引：

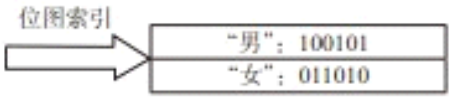
如下图所示

“男”对应的位图为100101，表示第1、4、6行值为“男”

“女”对应的位图为011010，表示第2、3、5行值为“女”

如果需要查找男性或者女性的个数，只需要统计相应的位图中1出现的次数即可。另外，建立位图索引后0和1的重复度高，可以采用专门的编码方式对其进行压缩。

行号	性别
1	男
2	女
3	女
4	男
5	女
6	男



当然，如果每次查询涉及的数据量较小或者大部分查询都需要整行的数据，列式数据库并不适用。

最后总结如下

传统行式数据库的特性如下：

①数据是按行存储的。

②没有索引的查询使用大量I/O。比如一般的数据库表都会建立索引，通过索引加快查询效率。

③建立索引和物化视图需要花费大量的时间和资源。

④面对查询需求，数据库必须被大量膨胀才能满足需求。

列式数据库的特性如下：

①数据按列存储，即每一列单独存放。

②数据即索引。

③只访问查询涉及的列，可以大量降低系统I/O。

④每一列由一个线程来处理，即查询的并发处理性能高。

⑤数据类型一致，数据特征相似，可以高效压缩。比如有增量压缩、前缀压缩算法都是基于列存储的类型定制的，所以可以大幅度提高压缩比，有利于存储和网络输出数据带宽的消耗。

SparkSQL入门

2018年2月10日 16:14

概述

SparkSql将RDD封装成一个DataFrame对象，这个对象类似于关系型数据库中的表。

创建DataFrame对象

DataFrame就相当于数据库的一张表。它是个只读的表，不能在运算过程再往里加元素。
RDD.toDF(“列名”)

```
scala> val rdd = sc.parallelize(List(1,2,3,4,5,6))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:21

scala> rdd.toDF("id")
res0: org.apache.spark.sql.DataFrame = [id: int]

scala> res0.show#默认只显示20条数据
+---+
| id|
+---+
|  1|
|  2|
|  3|
|  4|
|  5|
|  6|
+---+
scala> res0.printSchema #查看列的类型等属性
root
|-- id: integer (nullable = true)
```

创建多列DataFrame对象

DataFrame就相当于数据库的一张表。

```
scala> sc.parallelize(List( (1,"beijing"),(2,"shanghai") ) )
```

```
res3: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[5] at parallelize at <console>:22

scala> res3.toDF("id","name")
res4: org.apache.spark.sql.DataFrame = [id: int, name: string]

scala> res4.show
+---+--------+
| id|    name|
+---+--------+
|  1| beijing|
|  2|shanghai|
+---+--------+
```

例如3列的

```
scala> sc.parallelize(List( (1,"beijing",100780),(2,"shanghai",560090),(3,"xi'an",600329)))
res6: org.apache.spark.rdd.RDD[(Int, String, Int)] = ParallelCollectionRDD[10] at parallelize at <console>:22

scala> res6.toDF("id","name","postcode")
res7: org.apache.spark.sql.DataFrame = [id: int, name: string, postcode: int]

scala> res7.show
+---+--------+--------+
| id|    name|postcode|
+---+--------+--------+
|  1| beijing|  100780|
|  2|shanghai|  560090|
|  3|   xi'an|  600329|
+---+--------+--------+
```

可以看出，需要构建几列，tuple就有几个内容。

由外部文件构造DataFrame对象

1）txt文件

txt文件不能直接转换成，先利用RDD转换为tuple。然后toDF()转换为DataFrame。

```
scala> val rdd = sc.textFile("/root/words.txt")
```

```
.map( x => (x,1) )
.reduceByKey( (x,y) => x+y )
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[18] at reduceByKey at
<console>:21

scala> rdd.toDF("word","count")
res9: org.apache.spark.sql.DataFrame = [word: string, count: int]

scala> res9.show
+------+-----+
|  word|count|
+------+-----+
| spark|    3|
|  hive|    1|
|hadoop|    2|
|   big|    2|
|  scla|    1|
|  data|    1|
+------+-----+
```

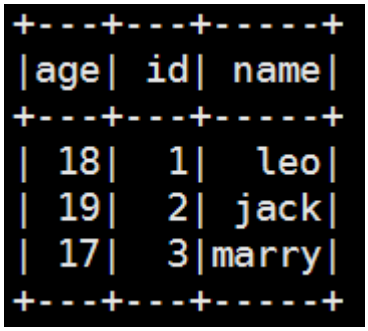
2) json文件

📖 **文件代码：**

```
{"id":1, "name":"leo", "age":18}
{"id":2, "name":"jack", "age":19}
{"id":3, "name":"marry", "age":17}
```

📖 **代码：**

```
import org.apache.spark.sql.SQLContext
scala>val sqc=new SQLContext(sc)
scala> val tb4=sqc.read.json("/home/software/people.json")
scala> tb4.show
```



```
+---+---+-----+
|age| id| name|
+---+---+-----+
| 18|  1|  leo|
| 19|  2| jack|
| 17|  3|marry|
+---+---+-----+
```

3）parquet文件

[pɑːrˈkeɪ]

1. 什么是Parquet数据格式？

Parquet是一种**列式存储格式**，可以被多种查询引擎支持（Hive、Impala、Drill等），并且它是语言和平台无关的。

2. Parquet文件下载后是否可以直接读取和修改呢？

Parquet文件是以二进制方式存储的，是不可以直接读取和修改的。Parquet文件是自解析的，文件中包括该文件的数据和元数据。

3. 列式存储和行式存储相比有哪些优势呢？

可以只读取需要的数据，降低IO数据量；

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进一步节约存储空间。

参考链接：

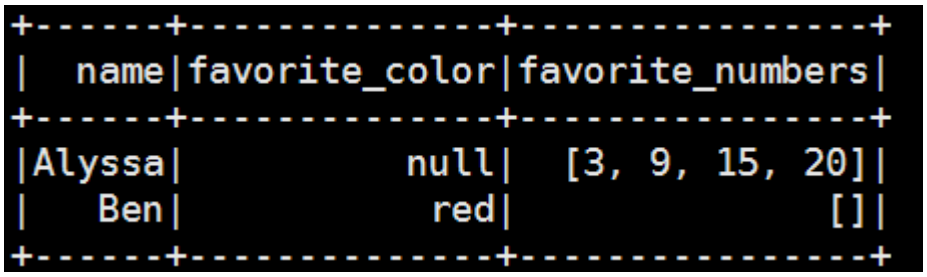
http://blog.csdn.net/yu616568/article/details/51868447 讲解了parquet文件格式

http://www.infoq.com/cn/articles/in-depth-analysis-of-parquet-column-storage-format 讲解了parquet列式存储

📖 **代码：**

scala>val tb5=sqc.read.parquet("/home/software/users.parquet")

scala> tb5.show



```
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

4) jdbc读取

实现步骤：

1) 将mysql 的驱动jar上传到spark的jars目录下

2) 重启spark服务

3) 进入spark客户端

4) 执行代码，比如在Mysql数据库下，有一个test库，在test库下有一张表为tabx

执行代码：

```
import org.apache.spark.sql.SQLContext
scala> val sqc = new SQLContext(sc);
scala> val prop = new java.util.Properties
scala> prop.put("user","root")
scala> prop.put("password","root")
scala>val tabx=sqc.read.jdbc("jdbc:mysql://hadoop01:3306/test","tabx",prop)
scala> tabx.show
+---+----+
| id|name|
+---+----+
|  1| aaa|
|  2| bbb|
|  3| ccc|
|  1| ddd|
|  2| eee|
|  3| fff|
+---+----+
```

注：如果报权限不足，则进入mysql，执行：

```
grant all privileges on *.* to 'root'@'hadoop01' identified by 'root' with grant option;
```

然后执行：

```
flush privileges;
```

SparkSql基础语法一上

2018年2月11日 17:59

通过方法来使用

(1)查询

df.select("id","name").show();

(2)带条件的查询

df.select($"id",$"name").where($"name" === "bbb").show()

(3)排序查询

orderBy/sort($"列名") 升序排列

orderBy/sort($"列名".desc) 降序排列

orderBy/sort($"列1" , $"列2".desc) 按两列排序

df.select($"id",$"name").orderBy($"name".desc).show

df.select($"id",$"name").sort($"name".desc).show

tabx.select($"id",$"name").sort($"id",$"name".desc).show

(4)分组查询

groupBy("列名", ...).max(列名) 求最大值

groupBy("列名", ...).min(列名) 求最小值

groupBy("列名", ...).avg(列名) 求平均值

groupBy("列名", ...).sum(列名) 求和

groupBy("列名", ...).count() 求个数

groupBy("列名", ...).agg 可以将多个方法进行聚合

```
scala>val rdd = sc.makeRDD(List((1,"a","bj",100),(2,"b","sh",80),(3,"c","gz",50),(4,"d","bj",45),(5,"e","gz",90)));

scala>val df = rdd.toDF("id","name","addr","score");

scala>df.groupBy("addr").count().show()

scala>df.groupBy("addr").agg(max($"score"), min($"score"), count($"*")).show
```

(5)连接查询

```
scala>val dept=sc.parallelize(List((100,"caiwubu"),(200,"yanfabu"))).toDF("deptid","deptname")

scala>val emp=sc.parallelize(List((1,100,"zhang"),(2,200,"li"),(3,300,"wang"))).toDF("id","did","name")

scala>dept.join(emp,$"deptid" === $"did").show

scala>dept.join(emp,$"deptid" === $"did","left").show
```

左向外联接的结果集包括 LEFT OUTER子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值。

```
scala>dept.join(emp,$"deptid" === $"did","right").show
```

(6)执行运算

```
val df = sc.makeRDD(List(1,2,3,4,5)).toDF("num");

df.select($"num" * 100).show
```

(7)使用列表

```
val df = sc.makeRDD(List(("zhang",Array("bj","sh")),("li",Array("sz","gz")))).toDF("name","addrs")

df.selectExpr("name","addrs[0]").show
```

(8)使用结构体

```
{"name":"陈晨","address":{"city":"西安","street":"南二环甲字1号"}}
```

{"name":"娜娜","address":{"city":"西安","street":"南二环甲字2号"}}

val df = sqlContext.read.json("file:///root/work/users.json")

dfs.select("name","address.street").show

(9)其他

df.count//获取记录总数

val row = df.first()//获取第一条记录

val value = row.getString(1)//获取该行指定列的值

df.collect //获取当前df对象中的所有数据为一个Array 其实就是调用了df对象对应的底层的rdd的collect方法

SparkSql基础语法—下

2018年2月11日 18:03

通过sql语句来调用

(0)创建表

df.registerTempTable("tabName")

(1)查询

```
val sqc = new org.apache.spark.sql.SQLContext(sc);

val df =

sc.makeRDD(List((1,"a","bj"),(2,"b","sh"),(3,"c","gz"),(4,"d","bj"),(5,"e","gz"))).toDF("id","name","addr");

df.registerTempTable("stu");

sqc.sql("select * from stu").show()
```

(2)带条件的查询

```
val df =

sc.makeRDD(List((1,"a","bj"),(2,"b","sh"),(3,"c","gz"),(4,"d","bj"),(5,"e","gz"))).toDF("id","name","addr");

df.registerTempTable("stu");

sqc.sql("select * from stu where addr = 'bj'").show()
```

(3)排序查询

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);

val df =

sc.makeRDD(List((1,"a","bj"),(2,"b","sh"),(3,"c","gz"),(4,"d","bj"),(5,"e","gz"))).toDF("id","name","addr");

df.registerTempTable("stu");

sqlContext.sql("select * from stu order by addr").show()
```

```
sqlContext.sql("select * from stu order by addr  desc").show()
```

(4)分组查询

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
val df =
sc.makeRDD(List((1,"a","bj"),(2,"b","sh"),(3,"c","gz"),(4,"d","bj"),(5,"e","gz"))).toDF("id","name","addr");
df.registerTempTable("stu");
sqlContext.sql("select addr,count(*) from stu group by addr").show()
```

(5)连接查询

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
val dept=sc.parallelize(List((100,"财务部"),(200,"研发部"))).toDF("deptid","deptname")
val emp=sc.parallelize(List((1,100,"张财务"),(2,100,"李会计"),(3,300,"王艳发"))).toDF("id","did","name")
dept.registerTempTable("deptTab");
emp.registerTempTable("empTab");
sqlContext.sql("select deptname,name from deptTab inner join empTab on deptTab.deptid =
empTab.did").show()
```

(6)执行运算

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
val df = sc.makeRDD(List(1,2,3,4,5)).toDF("num");
df.registerTempTable("tabx")
sqlContext.sql("select num * 100 from tabx").show();
```

(7)分页查询

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);

val df = sc.makeRDD(List(1,2,3,4,5)).toDF("num");

df.registerTempTable("tabx")

sqlContext.sql("select * from tabx limit 3").show();
```

(8)查看表

```
sqlContext.sql("show tables").show
```

(9)类似hive方式的操作

```
scala>val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

scala>hiveContext.sql("create table if not exists zzz (key int, value string) row format delimited fields

terminated by '|'")

scala>hiveContext.sql("load data local inpath 'file:///home/software/hdata.txt' into table zzz")

scala>hiveContext.sql("select key,value from zzz").show
```

(10)案例

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);

val df = sc.textFile("file:///root/work/words.txt").flatMap{ _.split(" ") }.toDF("word")

df.registerTempTable("wordTab")

sqlContext.sql("select word,count(*) from wordTab group by word").show
```

SparkSql API

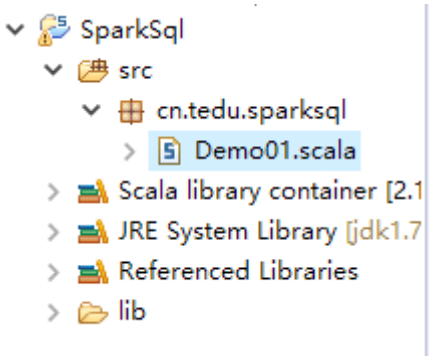
2018年2月11日 18:05

通过api使用sparksql

实现步骤：

1）打开scala IDE开发环境，创建一个scala工程

2）导入spark相关依赖jar包



3）创建包路径以object类

4）写代码

代码示意：

```
package cn.tedu.sparksql


import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.sql.SQLContext




object Demo01 {


 def main(args: Array[String]): Unit = {


  val conf=new SparkConf().setMaster("spark://hadoop01:7077").setAppName("sqlDemo01");
```

```
    val sc=new SparkContext(conf)
    val sqlContext=new SQLContext(sc)

    val rdd=sc.makeRDD(List((1,"zhang"),(2,"li"),(3,"wang")))

    import sqlContext.implicits._
    val df=rdd.toDF("id","name")
    df.registerTempTable("tabx")

    val df2=sqlContext.sql("select * from tabx order by name");
    val rdd2=df2.toJavaRDD;
    //将结果输出到linux的本地目录下，当然，也可以输出到HDFS上
    rdd2.saveAsTextFile("file:///home/software/result");
  }
}
```

5）打jar包，并上传到linux虚拟机上

6）在spark的bin目录下

执行：sh spark-submit --class cn.tedu.sparksql.Demo01 ./sqlDemo01.jar

7）最后检验

SparkStreaming介绍

2018年2月11日 18:08

概述

Spark Streaming是一种构建在Spark上的**实时计算**框架，它扩展了Spark处理大规模流式数据的能力，以**吞吐量高**和**容错能力**强著称。

SparkStreaming VS Storm

大体上两者非常接近，而且都处于快速迭代过程中，即便一时的对比可能某一方占优势。

在Spark老版本中，SparkStreaming的延迟级别达到秒级，而Storm可以达到毫秒级别。而在最新的2.0版本之后，SparkStreaming能够达到毫秒级。

但后者可能很快就追赶上来。比如在性能方面，Spark Streaming刚发布不久，有基准测试显示性能超过Storm几十倍，原因是Spark Streaming采用了小批量模式，而Storm是一条消息一条消息地计算。但后来Storm也推出了称为Trident的小批量计算模式，性能应该不是差距

了。而且双方都在持续更新，底层的一个通信框架的更新或者某个路径的代码优化都可能让性能有较大的提升。

目前，sparkStreaming还不能达到一条一条记录的精细控制，还是以batch为单位。所以像Storm一般用于金融领域，达到每笔交易的精细控制。

但是两者的基因不同，更具体地说就是核心数据抽象不同。这是无法改变的，而且也不会轻易改变，这样的基因也决定了它们各自最适合的应用场景。

Spark Streaming的核心抽象是**DSTream**，里面是RDD，下层是Spark核心DAG调度，所以Spark Streaming的这一基因决定了其粒度是小批量的，无法做更精细地控制。数据的可靠性也是以批次为粒度的，但好处也很明显，就是有可能实现更大的吞吐量。

另外，得益于Spark平台的良好整合性，完成相同任务的流式计算程序与历史批量处理程序的代码基本相同，而且还可以使用平台上的其他模块比如SQL、机器学习、图计算的计算能力，在开发效率上占有优势。而Storm更擅长细粒度的消息级别的控制，比如延时可以实现毫秒级，数据可靠性也是以消息为粒度的。

核心数据抽象的不同导致了它们在计算模式上的本质区别。Spark Streaming在本质上其实是像MR一样的批处理计算，但将批处理的周期从常规的几十分钟级别尽可能缩短至秒级（毫秒级），也算达到了实时计算的延时指标。而且，它支持各类数据源，基本可以实现流式计算的功能，但延时无法进一步缩短了。但Storm的设计初衷就是实时计算，毫秒级的计算当然不在话下，而且后期通过更高级别的Trident也实现了小批次处理功能。

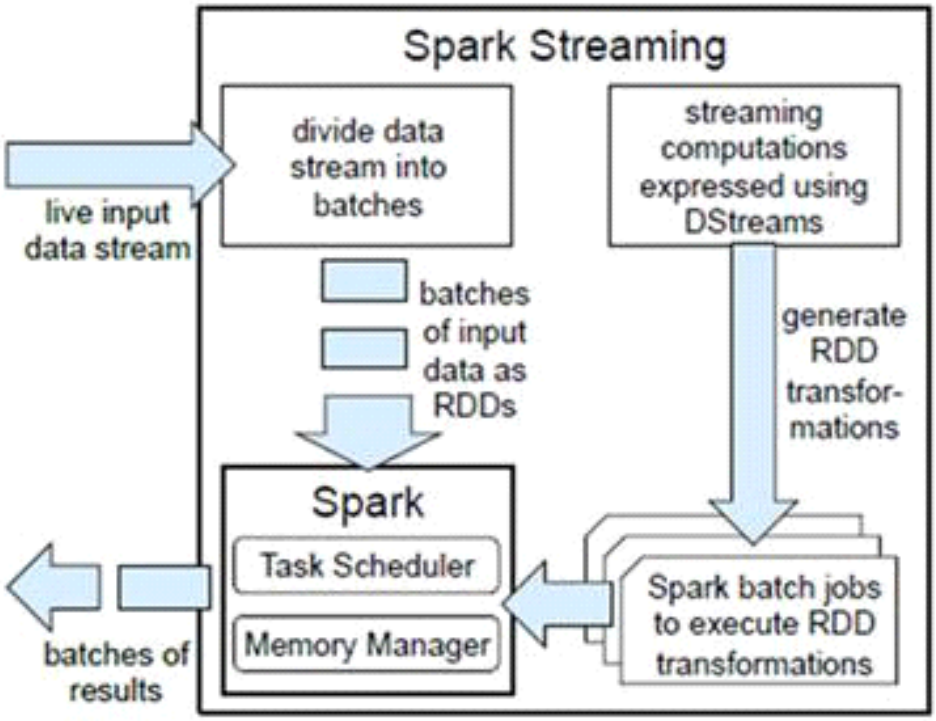
	Spark Streaming	Storm
计算模型	小批量计算	实时数据流计算
实时性	秒级	最快10毫秒级
数据可靠性	精确只被处理一次	核心支持最少一次或最多一次； Trident支持全部最少一次、最多一次、精确一次
性能	性能测试视硬件、网络、程序等多方因素而定，无法直接对比，这里只是引用一些公开的数据： Spark Streaming：40万记录/秒/节点 Storm：100万/秒/节点	
实现语言	Scala	Clojure
编程语言	Scala、Java、Python	Trident只支持Clojure、Java、Scala，核心还支持Python、Ruby等其他语言
相同点	分布式、容错、可扩展	

架构及原理

2018年2月11日 18:10

架构设计

SparkStreaming是一个对实时数据流进行高通量、容错处理的流式处理系统，可以对多种数据源（如Kafka、Flume、Twitter、ZeroMQ和TCP 套接字）进行类似Map、Reduce和Join等复杂操作，并将结果保存到外部文件系统、数据库或应用到实时仪表盘。



Spark Streaming是将流式计算分解成一系列短小的批处理作业，也就是把Spark Streaming的输入数据按照batch size（如1秒）分成一段一段的数据DStream（Discretized-离散化Stream），每一段数据都转换成Spark中的RDD（Resilient Distributed Dataset），然后将Spark Streaming中对DStream的Transformations操作变为针对Spark中对RDD的Transformations操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加或者存储到外部设备。

对DStream的处理，每个DStream都要按照数据流到达的**先后顺序依**次进行处理。这样使所有的批处理具有了一个顺序的特性，其本质是转换成RDD的血缘关系。所以，

SparkStreaming对数据天然具有容错性保证。

为了提高SparkStreaming的工作效率，你应该合理的配置批的时间间隔， 最好能够实现上一个批处理完某个算子，下一个批子刚好到来。

入门基础案例

2018年2月11日 22:41

WordCount案例

案例一：

```
import org.apache.spark.streaming._
val ssc = new StreamingContext(sc,Seconds(5));
val lines = ssc.textFileStream("file:///home/software/stream");
//val lines = ssc.textFileStream("hdfs://hadoop01:9000/wordcount");
val words = lines.flatMap(_.split(" "));
val wordCounts = words.map((_,1)).reduceByKey(_+_);
wordCounts.print();
ssc.start();
```

基本概念

1. StreamingContext

StreamingContext是Spark Streaming编程的最基本环境对象，就像Spark编程中的SparkContext一样。StreamingContext提供最基本的功能入口，包括从各途径创建最基本的对象DStream（就像Spark编程中的RDD）。

创建StreamingContext的方法很简单，生成一个SparkConf实例，设置程序名，指定运行周期（示例中是5秒），这样就可以了：

```
val conf = new SparkConf().setAppName("SparkStreamingWordCount")
val ssc = new StreamingContext(conf, Seconds(5))
```

运行周期为5秒，表示流式计算每间隔5秒执行一次。这个时间的设置需要综合考虑程序的延时需求和集群的工作负载，应该大于每次的运行时间。

StreamingContext还可以从一个现存的org.apache.spark.SparkContext创建而来，并保持关联，比如上面示例中的创建方法：

```
val ssc = new StreamingContext(sc, Seconds(5))
```

StreamingContext创建好之后，还需要下面这几步来实现一个完整的Spark流式计算：

（1）创建一个输入DStream，用于接收数据；

（2）使用作用于DStream上的Transformation和Output操作来定义流式计算（Spark程序是使用Transformation和Action操作）；

（3）启动计算，使用streamingContext.start()；

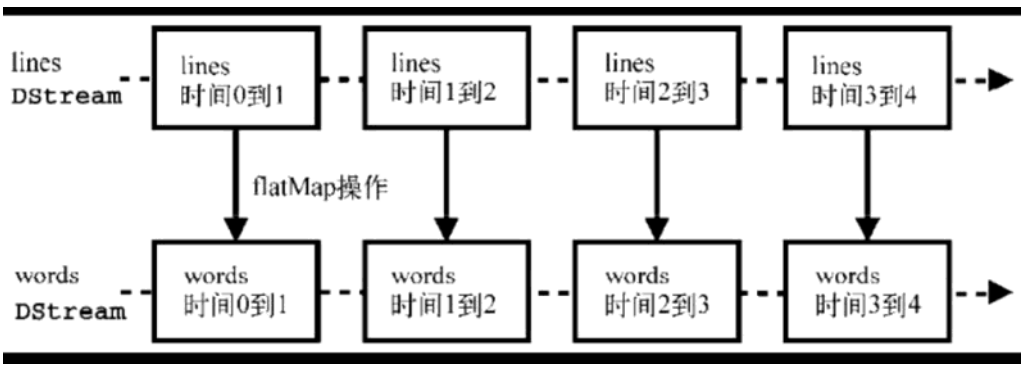
（4）等待计算结束（人为或错误），使用streamingContext.awaitTermination()；

（5）也可以手工结束计算，使用streamingContext.stop()。

2. DStream抽象

DStream（discretized stream）是Spark Streaming的核心抽象，类似于RDD在Spark编程中的地位。DStream表示连续的数据流，要么是从数据源接收到的输入数据流，要求是经过计算产生的新数据流。DStream的内部是一个RDD序列，每个RDD对应一个计算周期。比如，在上面的WordCount示例中，每5秒一个周期，那么每5秒的数据都分别对应一个RDD，如图所示，图中的时间点1、2、3、4代表连续的时间周期。

所有应用在DStream上的操作，都会被映射为对DStream内部的RDD上的操作，比如上面的WordCount示例中对lines DStream的flatMap操作，如下图



RDD操作将由Spark核心来调度执行，但DStream屏蔽了这些细节，给开发者更简洁的编程体验。当然，我们也可以直接对DStream内部的RDD进行操作（后面会讲到）。

案例二：

经过测试，案例一代码确实可以监控指定的文件夹处理其中产生的新的文件

但数据在每个新的周期到来后，都会重新进行计算

而**如果需要对历史数据进行累计处理 该怎么做呢**？

SparkStreaming提供了**checkPoint**机制，首先需要设置一个检查点目录，在这个目录，存储了历史周期数据。通过在临时文件中存储中间数据 为历史数据累计处理提供了可能性

```
import org.apache.spark.streaming._
val ssc = new StreamingContext(sc,Seconds(5));
ssc.checkpoint("file:///home/software/chk");
val lines = ssc.textFileStream("file:///home/software/stream");
val  result= lines.flatMap(_.split(" ")).map((_,1)).updateStateByKey{(seq, op:Option[Int]) => { Some(seq.sum +op.getOrElse(0)) }}
result.print();
ssc.start();
```

updateStateByKey 方法说明：

1.seq:是一个序列，存的是某个key的历史数据

2.op:是一个值，是某个key当前的值

比如： (hello,1)

①seq里是空的，Some(1)=>Some（返回的是历史值的和+当前值）

②(hello,2)，seq(1) op=2 Some(1+2)

③(hello,1)，seq(1,2) op=1 Some(3+1)

案例三：

但是这上面的例子里所有的数据不停的累计 一直累计下去

很多的时候我们要的也不是这样的效果 我们希望能够每隔一段时间重新统计下一段时间的数据，并且能够对设置的批时间进行更细粒度的控制，这样的功能可以通过滑动窗口的方式来实现。

在DStream中提供了如下的和滑动窗口相关的方法：

window(windowLength, slideInterval)

windowLength：窗口长度

slideInterval：滑动区间

countByWindow(windowLength, slideInterval)

reduceByWindow(func, windowLength, slideInterval)

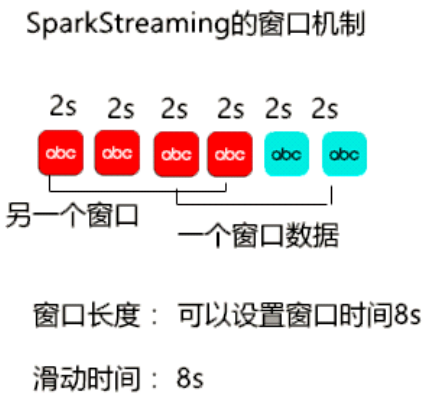
reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])

reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])

countByValueAndWindow(windowLength, slideInterval, [numTasks])

可以通过以上机制改造案例

```
import org.apache.spark.streaming._
val ssc = new StreamingContext(sc,Seconds(1));
ssc.checkpoint("file:///home/software/chk");
val lines = ssc.textFileStream("file:///home/software/stream");
val result = lines.flatMap(_.split(" ")).map((_,1)).reduceByKeyAndWindow((x:Int,y:Int)=>x+y, Seconds(5), Seconds(5) ).print()
ssc.start();
```



注意：窗口长度和滑动长度必须是batch size的整数倍
此外，使用窗口机制，必须要设定检查点目录

输入输出相关方法

2018年2月19日 12:59

输出相关方法

输出	解释及案例
print()	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging.

saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as text files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.
	
	保存为文本文件、本地磁盘系统或HDFS
	
	代码一：
	import org.apache.spark.streaming._
	val ssc = new StreamingContext(sc,Seconds(10));
	val lines = ssc.textFileStream("file:///home/software/stream");
	val words = lines.flatMap(_.split(" "));
	val wordCounts = words.map((_,1)).reduceByKey(_+_);
	wordCounts.saveAsTextFiles("file:///home/software/result/part-r")
	
	ssc.start ( )
	
	代码二：
	import org.apache.spark.streaming._
	val ssc = new StreamingContext(sc,Seconds(10));
	val lines = ssc.textFileStream("file:///home/software/stream");
	val words = lines.flatMap(_.split(" "));
	val wordCounts = words.map((_,1)).reduceByKey(_+_);
	wordCounts.saveAsTextFiles("hdfs://hadoop01:9000/result/part-r")
	ssc.start

foreachRDD(func)	The most generic output operator that applies a function, func, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function func is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs. 最灵活的Output操作，传入一个函数，直接作用在RDD上。传入的函数可以实现任意功能，比如写

文件、写DB，或者通过网络发送出去

代码一：

```
import org.apache.spark.streaming._
val ssc = new StreamingContext(sc,Seconds(10));
val lines = ssc.textFileStream("file:///home/software/stream");
val words = lines.flatMap(_.split(" "));
val wordCounts = words.map((_,1)).reduceByKey(_+_);

wordCounts.foreachRDD(rdd => {
val count = rdd.count()
print(" line count is " + count)
rdd.saveAsTextFile("/home/software/result")
})

ssc.start
```

```
scala>  line count is 0 line count is 4 line count is 0 line count is 0 line c
unt is 0 line count is 0 line count is 0 line count is 0 line count is 0
```

输入控制

输入DStream是DStream的起始

基本型。在StreamingContext的API中直接提供，比如文件系统

高级型。需要通过额外的库来实现，比如Kafka、Flume、Kinesis、Twitter，而且在编译时

需要设置依赖

文件输入

最常用的是从文件生成输入DStream，方法是使用方法textFileStream，比如接收一个HDFS

文件系统兼容的目录作为参数，Spark会实时监控这个目录，以文件为单位。新出现的文件会

被按照文本文件格式读取到Spark内存中。对于文件的几点要求如下：

1）目录中的所有文件的格式相同；

2）只处理目录下的文件，不包括子目录下的文件；

3）文件进入目录的方式必须是原子的，比如被从其他目录移动或重命名得来；

注意：文件进入目录之后，建议不要再被修改，因为只会被读取一次，新追加的内容不会被读取。

代码示意：

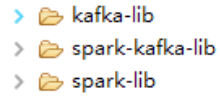
```
val lines = ssc.textFileStream("file:///home/software/stream");
或
val lines = ssc.textFileStream("hdfs://hadoop01:9000/word");
```

Kafka整合

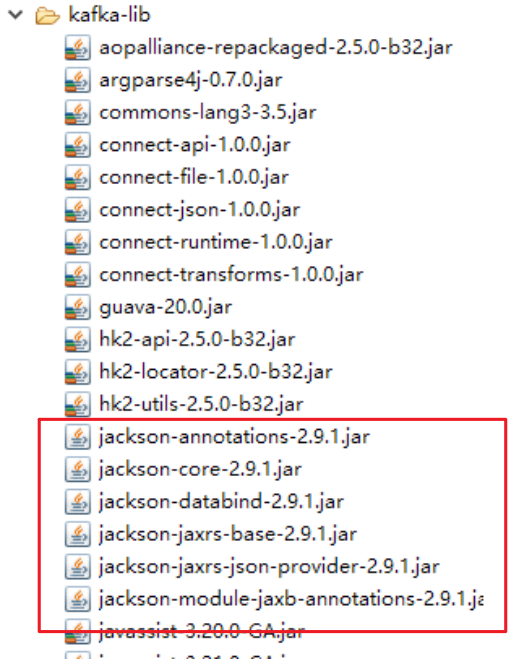
2018年2月20日 15:48

实现步骤：

1.导入相关依赖jar包（注意jar包版本问题）



2.如果报jackson错误，将相关jar包依赖移除



3.启动Zookeeper集群

4.启动Kafka

进入bin目录，执行：sh kafka-server-start.sh ../config/server.properties

5.创建主题

进入bin目录，执行：sh kafka-topics.sh --create --zookeeper hadoop01:2181 --replication-factor 1 --partitions 1 --topic parkx

6.启动生成者线程，用于发送消息

进入bin目录，执行：sh kafka-console-producer.sh --broker-list hadoop01:9092 --topic parkx

7.编写spark代码：

代码示例：

```
object Demo01 {

 def main(args: Array[String]): Unit = {
  val conf = new SparkConf();
  conf.setAppName("spark_streaming_Kafka_demo");
  conf.setMaster("local[2]");
  val sc = new SparkContext(conf);


  val ssc = new StreamingContext(sc,Seconds(5));

  //连接kafka 消费数据
  val kstream = KafkaUtils.createStream(ssc, "hadoop01:2181,hadoop02:2181,hadoop03:2181", "gx1",  Map("parkx"->1)).map(_._2);
  kstream.print();
```

```
    //启动streaming
    ssc.start();

  //保持streaming线程一直开启
    ssc.awaitTermination()


  }
}
```

课后作业：Spark和Hbase整合

2018年2月20日 15:48

实现步骤：

1.导入hbase相关依赖包

2.启动zookeeper集群

3.启动hbase集群

在主节点的bin目录下，执行： sh start-hbase.sh

4.进入hbase

在bin目录执行：sh hbase shell

5.建立测试表

2.编写spark代码：

代码示例（向hbase插入数据）：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.hadoop.mapreduce.Job
import org.apache.hadoop.fs.shell.find.Result
import org.apache.hadoop.hbase.mapreduce.TableOutputFormat
import org.apache.hadoop.hbase.client.Put
import org.apache.hadoop.hbase.util.Bytes
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
object Demo01 {

 def main(args: Array[String]): Unit = {
   val conf = new SparkConf();
  conf.setAppName("spark_hbase");
  conf.setMaster("local");
  val sc = new SparkContext(conf);
```

```
  sc.hadoopConfiguration.set("hbase.zookeeper.quorum","hadoop01,hadoop02,hadoop03")
        sc.hadoopConfiguration.set("hbase.zookeeper.property.clientPort", "2181")
        sc.hadoopConfiguration.set(TableOutputFormat.OUTPUT_TABLE, "tabx")

        val job = new Job(sc.hadoopConfiguration)
        job.setOutputKeyClass(classOf[ImmutableBytesWritable])
        job.setOutputValueClass(classOf[Result])
        job.setOutputFormatClass(classOf[TableOutputFormat[ImmutableBytesWritable]])
        val confx = job.getConfiguration();


  val indataRDD = sc.makeRDD(Array("rk1,zhang,19","rk2,li,29","rk3,wang,39"))
        val rdd = indataRDD.map(_.split(',')).map{arr=>{
          val put = new Put(Bytes.toBytes(arr(0)))
          put.add(Bytes.toBytes("cf1"),Bytes.toBytes("name"),Bytes.toBytes(arr(1)))
          put.add(Bytes.toBytes("cf1"),Bytes.toBytes("age"),Bytes.toBytes(arr(2).toInt))
          (new ImmutableBytesWritable(), put)
        }}

        rdd.saveAsNewAPIHadoopDataset(confx)
 }
}
```

代码示例（从hbase读取数据）：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
import org.apache.hadoop.hbase.util.Bytes

object Demo02 {

 def main(args: Array[String]): Unit = {
  //1.创建sc
  val conf = new SparkConf();
  conf.setAppName("spark_hbase");
  conf.setMaster("local");
```

```
    val sc = new SparkContext(conf);

    //2.指定相关配置信息
    val confx = HBaseConfiguration.create()
    confx.set("hbase.zookeeper.quorum","hadoop01,hadoop02,hadoop03")
    confx.set("hbase.zookeeper.property.clientPort", "2181")
    confx.set(TableInputFormat.INPUT_TABLE,"tabx")

    //3.读取hbase
    val rdd = sc.newAPIHadoopRDD(confx,  classOf[TableInputFormat],
                        classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],
                        classOf[org.apache.hadoop.hbase.client.Result]);

    //4.获取记录数
    val count = rdd.count
    println(count)

    //5.遍历数据
    rdd.foreach(t=>{
      val result = t._2
      val key = Bytes.toString(result.getRow)
      val name = Bytes.toString(result.getValue("cf1".getBytes,"name".getBytes));
      val age = Bytes.toInt(result.getValue("cf1".getBytes,"age".getBytes));
      println(key+"~"+name+"~"+age)
    })



    sc.stop()
  }
}
```

代码示例（使用hbase的过滤器查询）：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
import org.apache.hadoop.hbase.util.Bytes
import org.apache.hadoop.hbase.client.Scan
```

```
import org.apache.hadoop.hbase.protobuf.ProtobufUtil
import org.apache.hadoop.hbase.util.Base64
import org.apache.hadoop.hbase.filter.RandomRowFilter
object Demo03 {

  def main(args: Array[String]): Unit = {
    //1.创建sc
    val conf = new SparkConf();
    conf.setAppName("spark_hbase");
    conf.setMaster("local");
    val sc = new SparkContext(conf);

    //2.指定相关配置信息
    val confx = HBaseConfiguration.create()
    confx.set("hbase.zookeeper.quorum","hadoop01,hadoop02,hadoop03")
    confx.set("hbase.zookeeper.property.clientPort", "2181")
    confx.set(TableInputFormat.INPUT_TABLE,"tabx1")
//  confx.set(TableInputFormat.SCAN_ROW_START , "rk1")
//  confx.set(TableInputFormat.SCAN_ROW_STOP , "rk3")

    val scan = new Scan()
    //scan.setStartRow("rk1".getBytes)
    //scan.setStopRow("rk2".getBytes)
    scan.setFilter(new RandomRowFilter(0.5f))
    confx.set(TableInputFormat.SCAN, Base64.encodeBytes(ProtobufUtil.toScan(scan).toByteArray))

    //3.读取hbase
    val rdd = sc.newAPIHadoopRDD(confx,  classOf[TableInputFormat],
                        classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],
                        classOf[org.apache.hadoop.hbase.client.Result]);

    //4.获取记录数
    val count = rdd.count
    println(count)

    //5.遍历数据
    rdd.foreach(t=>{
      val result = t._2
```

```
      val key = Bytes.toString(result.getRow)
      val name = Bytes.toString(result.getValue("cf1".getBytes,"name".getBytes));
      val age = Bytes.toInt(result.getValue("cf1".getBytes,"age".getBytes));
      println(key+"~"+name+"~"+age)
    })

    sc.stop()
  }
}
```