第14章 Apache Spark分布式计算原理

Objective(本课目标)

- [x] 掌握Spark中的数据分区(Data Partition)及数据迁移(Data Suffle)原理及方法
- [x] 运用Spark Cache/Persist及Checkpointing提高 Spark应用的性能和可靠性
- [x] 学会使用Spark Shell
- [x] 练习Spark装载CSV， JSON数据文件

Two Main Spark Abstractions

- RDD – Resilient Distributed Dataset
- DAG – Direct Acyclic Graph (有向无环图)

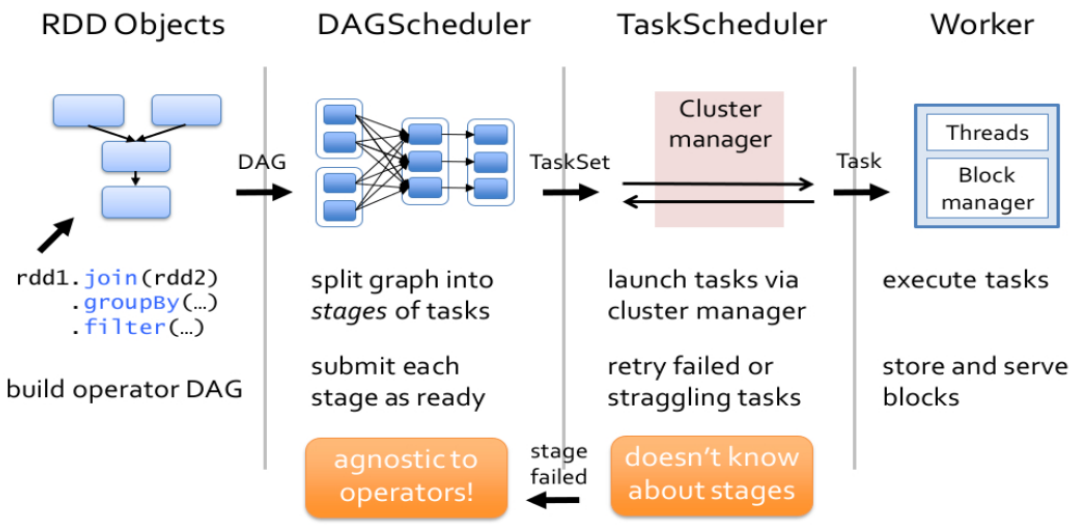
Lineage

> “
>
> RDD只支持粗粒度转换，即在大量记录上执行的单个操作。将创建RDD的一系列Lineage（即血统）记录下来，以便恢复丢失的分区。RDD的Lineage会记录RDD的元数据信息和转换行为，当该RDD的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

DAG

- Direct Acyclic Graph – sequence of computations performed on data (对数据执行的计算序列)
  - Node: RDD partition (节点：对应RDD的分区)
  - Edge: Transformation on top of data(边：基于数据进行变换)
  - Acyclic: Graph cannot return to the older partition (数据变换之后生产的数据集不能回到之前)
  - Direct: transformation is an action that transitions data partition state (from A to B) (数据转换会改变数据的分区状态)

How Spark Works by DAG(DAG是如何工作的)

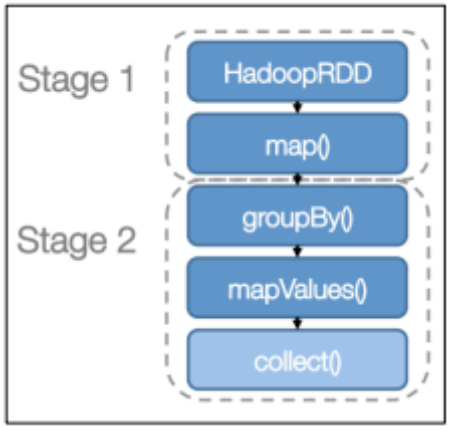


1. 将代码的在内部翻译成执行指令，但是不会真正的去执行。(这个指令就相当于一个DAG图)
2. 在翻译的过程中编译器是知道那些过程是会产生数据的shuffle
3. 根据rdd变换会生成data shuffling，划分为一个stage
4. 当stage规划完成之后就会提交给DAGScheduler
5. DAGScheduler将stage生成一个TaskSet发送给TaskScheduler
6. TaskScheduler安排，调度，每一个任务，提交给Worker来完成
7. 一个stage的所有task完成才会进入下一个Stage

• Spark Execution Model(执行模型)

- Create DAG of RDDs to represent computation (创建RDD的DAG来表示计算)

- Create logical execution plan for DAG (为DAG创建逻辑执行计划)
  - Split into “stages” based on need to re-organize data (基于需要整理的数据形成stages)

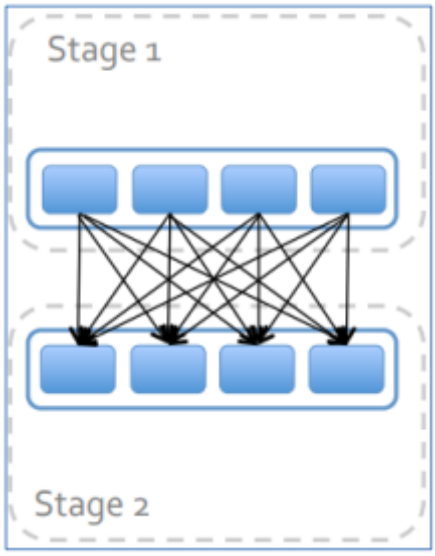


- Schedule and execute individual tasks (调度并执行每一个的任务)

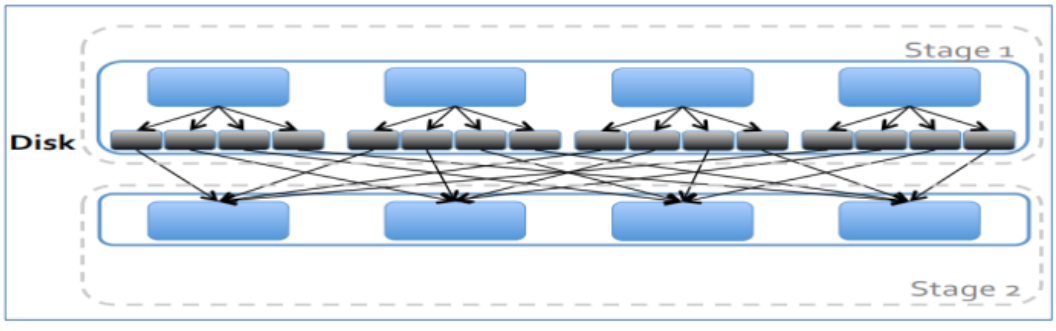
- Split each stage into tasks (per partition); (把每个阶段包含多个任务)
- A task is data + computation (包含了数据和计算)
- Execute all tasks within a stage before moving on (一个阶段的所有task任务执行完成，才会执行下一个task)

The Shuffle in Spark(数据迁移)

- Redistribute data among partitions (在分区之间重新分配数据)
  - Avoid when possible (尽量避免)
  - Partial aggregation reduces data movement (部分的集聚，减少数据的传输量)



- The Shuffle
  - Pull-based, not push-based (第二个阶段会从上一个阶段"拿"数据)
  - Write intermediate files to disk (将中间文件写入磁盘)
  - By default, shuffling doesn't change the # of partitions(默认情况下，shuffle不会改变分区数目)



Note：Mapreduce的shuffle，是由Map阶段推送到reduce阶段的这么一个过程，Spark是pull-based，rdd处理完成之后，不知道那个数据需要进入到那个分区，当下一步操作的时候，当前rdd知道需要什么样子的数据，然后到对应的地方去获取数据。(性能优化)Map阶段输出的数据首先是走缓存，当缓存达到阈值的时候，会把数据写到本地磁盘(更快)Spark也是将临时的结果写到本地磁盘，获取数据也是直接从本地磁盘。在缺省情况下rdd不会修改partition的数目。

Fixing the Mistake(解决错误)

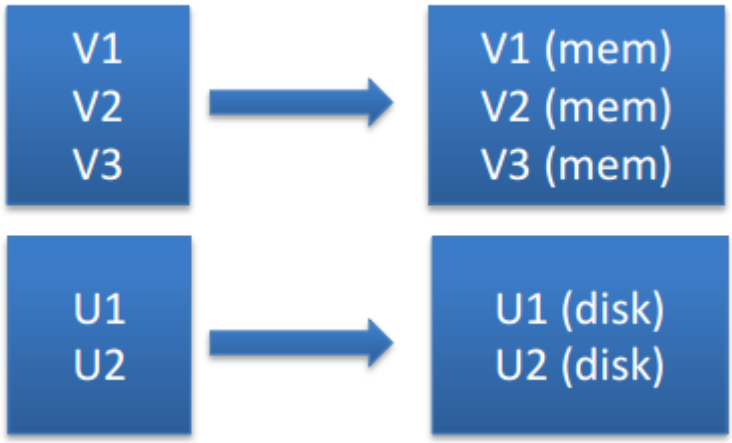
```
//比较性能
sc.textFile("hdfs:/names")
.map(name = > (nane.charAt(0),name))
.groupByKey()
.mapValues{ names => names.toSet.size}
.collect

// 至少快2-3倍
sc.textFile("hdfs:/names")
.distinct(numPartitions = 5)
.map(name => (name.charAt(0),1))
.reduceByKey(_+_)
.collect

// 如果先过滤数据而不影响结果的话，尽量先对数据进行过滤。
```

RDD Optimization(RDD优化)

- RDD Cache / Persist (缓存)
- RDD Partition Size (分区大小)
- RDD Broadcast (广播)
- RDD Checkpointing (保存点设置)

RDD Persistence – cache/persist(持久化)

- Cache data to memory/disk. It is a lazy operation (缓存数据是属于延迟操作)
- cache = persist(MEMORY) 2.0之前， persist(MEMORY_AND_DISK) 2.0之后
- 为什么需要Persistence， RDD不是可以自动计算出来吗?




```
//案例1
def main(args: Array[String]): Unit = {
        Logger.getLogger("org").setLevel(Level.ERROR)
```

```
        val sparkConf = new
SparkConf().setAppName(this.getClass.getSimpleName).setMaster("local")
        val sparkContext = new SparkContext(sparkConf)

        val rdd = sparkContext.makeRDD(1 to 10)
        /**
         * 不缓存
         */
        val nocacheRDD = rdd.map(_.toString + "[" + System.currentTimeMillis +
"]")
        println(nocacheRDD.collect.mkString(","))
        println(nocacheRDD.collect.mkString(","))

        println("-----------------------------------")

        /**
         * 有缓存
         */
        val cacheRDD = rdd.map(_.toString + "[" + System.currentTimeMillis + "]")
        cacheRDD.cache()     // 在第一执行得到cacheRDD的数据的时候。 cache()方法就会把这
个cacheRDD 放到内存
        println(cacheRDD.collect.mkString(","))    // 第一次触发执行，所以map操作执行
了，得到了cacheRDD
        println(cacheRDD.collect.mkString(","))    // 第二次action触发执行的时候，就直
接从内存中，获取cacheRDD的值
        //释放缓存
        cacheRDD.unpersist()
    }


//案例2，rdd2的值为多少?
val data = Array(100, 200, 300, 400) //定义一个数组
val rdd1 = sc.parallelize(data) //定义rdd1
val rdd2 = rdd1.map(x => x / 100) //通过rdd1 生成rdd2
data(2) = 900 //修改 data的值
```

- 如果生成的rdd是shuffleling(wide)操作，可以cache，为了避免后面需要使用到，而重复的操作。(繁重的操作)

Storage Level for persist(缓存级别)

Storage Level	Purpose
MEMORY_ONLY	Keep RDD in available cluster memory as de-serialized Java objects. Some partitions may not be cached if there is not enough cluster memory. Those partitions will be recalculated on the fly as needed.
MEMORY_AND_DISK(Default level)	This option stores RDD as de-serialized Java objects. If RDD does not fit in cluster memory, then store those partitions on the disk and read them as needed.(内存不够放到硬盘)
MEMORY_ONLY_SER	This options stores RDD as serialized Java objects (One byte array per partition). This is more CPU intensive but saves memory as it is more space efficient. Some partitions may not be cached. Those will be recalculated on the fly as needed.(序列化之后放到内存)
MEMORY_AND_DISK_SER	This option is same as above except that disk is used when memory is not sufficient.(序列化之后放到内存和磁盘)
DISK_ONLY	This option stores the RDD only on the disk(只存在硬盘)
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as other levels but partitions are replicated on 2 slave nodes (缓存之后再做一次备份)

```
# 调用persist方法必须手动指定缓存方式
rdd.persist(StorageLevel.MEMORY_AND_DISK)

# catch底层也是persist  persist(): this.type = persist(StorageLevel.MEMORY_ONLY)
def cache(): this.type = persist()
```

RDD Checkpointing(保存点)

- Checkpointing is a process of truncating RDD lineage(Rdd的生成步骤) graph, and saving it to a storage for failure recovery (将rdd的执行状态存储起来，便于失败恢复)
  - Reliable – checkpointing saves RDD to HDFS in Spark Core (Spark Core存储在HDFS)
  - Local – checkpointing saves RDD to local FS in Spark Streaming or GraphX (Spark Streaming or GraphX存储在local FileSystem)
  - setCheckpointDir – set the directory under which RDDs are going to be checkpointed (设置保存点目录)
    - The directory must be a HDFS path if running on a cluster (必须是一个HDFS目录)
  - checkpoint() – mark a RDD for checkpointing to be saved to a file (将RDD标记为要保存到文件的检查点)
  - getCheckpointFile() – get the file name of the RDD checkpointed (获取checkpointed RDD的文件)

```
#案例
```

checkpointing vs persist(保存点和缓存和区别)

- checkpointing：
  - Remove RDD Lineage (不保存RDDLineage)
  - Developer's responsibility to remove or keep the data on disk (开发人员管理数据)
  - in order to enhance application failure tolerance (增强程故障序容错能力)
- Cache/Persist：
  - Keep RDD Lineage (保持RDDLineage)
  - Spark's responsibility to remove the data on disk/memory (spark负责管理数据)
  - performance enhancement (提高性能)

1. 应用程序执行完成之后，会删除checkpoint和persist产生的临时数据。(存放在临时录)
2. checkpoint 存储的文件我们需要自己管理
3. persist会由程序自动清除
4. persist存储的数据，是包含了RDD的 RDD's lineage(RDD如何生成的)
5. Checkpoint 没有保存 RDD's lineage，只保存数据
6. Checkpoint vs Persist都是lazy的

Spark Variable - Broadcast(广播变量)

- Like Hadoop DC, broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. For example, to give every node a copy of a large input dataset efficiently (与Hadoop DC一样，broadcast变量允许程序员在每台机器上缓存一个只读变量，而不是将其副本与任务一起发送。例如，要有效地为每个节点提供一个大型输入数据集的副本)
- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost (Spark还尝试使用高效的广播算法分发广播变量，以降低通信成本)

```
// 案例1 仅在驱动程序节点上可用
val factor = 100
val rdd = sc.parallelize(Arrray(1,2,3,4))
rdd.map(x => {
val y = facotr * x
y * y
})

// 案例2 在WorkNode上执行，在WorkNode上面没有 factor变量. 使用Broadcast广播到WordNode
val f = sc.broadcast(factor) // should be used on worker node
rdd.map(x => {
val factor = f.value
```

```
val y = x * factor
y * y
}
)

//driver 不可能定义全局变量，全局变量只能是driver node 上广播到 worker node 上的
//MapSiteJoin -> Spark broadcast(rdd)
//rdd2.join(rdd1) ,可以把rdd1广播到rdd2所在的分区节点.这样性能会更好。类似于MapJoin
//需要广播吗?
foreach { x => {
val y = factory * x
println(y)
} }

// 案例3 rdd广播变量使用
val r = sc.broadcast(rdd1)
rdd2.join(r.value)
join: rdd1.join(rdd2)  // rdd1 should be larger than rdd2 (rdd2需要被 shuffle)
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
val fn= sc.textFile("hdfs://filename").map(_.split("|"))
val fn_bc= sc.broadcast(fn)
```

RDD Partition Size(分区大小)?

- The Limit of Partition Size is 2GB (分区大小的限制是2GB)
  - Overflow exception if shuffle block size > 2 GB.(分区shuffle的块数据量太大会内存溢出)
  - Spark uses ByteBuffer as anstraction for blocks (Spark使用ByteBuffer作为块的)

```
val buf = ByteBuffer.allocate(length.toInt)
//ByteBuffer is limited by Integer.MAX_SIZE (2GB)
```

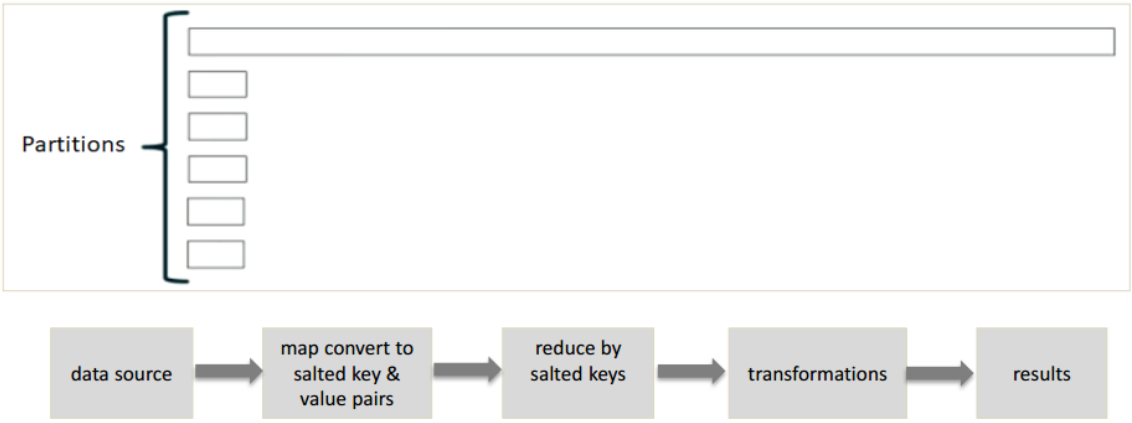
- Don’t have too few partitions (不要有太少的分区)
  - Jobs will be slow, not making use of parallelism(效率低，并行不够)
- Rule of thumb (经验法则): ~128MB per partition (建议一个partition128M)
- If # of partitions is less but close to 2000, bump to just > 2000. (如果分区数小于但接近2000，则大约 > 2000)
- Spark uses different data structure for bookkeeping during shuffles, when the # of partitions is less than 2000 vs. more than 2000
- SPARK-6235: 2 GB limit https://issues.apache.org/jira/browse/SPARK-6235

Importance of Partition Tuning (调整分区的重要性)

- Main issue: too few partitions (分区较少)
  - Less concurrency (并发性较低)
  - More susceptible to data skew (更容易受到数据偏斜的影响)
  - Increased memory pressure for groupBy, reduceByKey, sortByKey, etc. (增加了groupBy，reduceByKey，sortByKey等的内存压力)
- Secondary issue: too many partitions(分区较多)
  - Overhead of data shuffling (数据shuffling的开销过大)
  - Overhead of process creation (进程创建的开销)

Data Skew(数据倾斜)

- Data Skew slows performance significantly (数据倾斜会显著降低性能)
- Data Skew is normally created after group by, joins, etc. (数据倾斜通常是在group by、join等之后创建的)
- Solution – Repartition through new hashing / salting: (解决方案-salting)




```
//案例1
val u1 = sc.parallelize(Array("A","B","A","B","B","B","B","B"),5)
u1.saveAsTextFile("/root/spark/skew")
-rw-r--r-- 1 root root 2 May 14 10:53 part-00000
-rw-r--r-- 1 root root 4 May 14 10:53 part-00001
-rw-r--r-- 1 root root 2 May 14 10:53 part-00002
-rw-r--r-- 1 root root 4 May 14 10:53 part-00003
-rw-r--r-- 1 root root 4 May 14 10:53 part-00004
-rw-r--r-- 1 root root 0 May 14 10:53 _SUCCESS

//案例2
val u2 = u1.map(x => (x,1)).groupByKey()
Array[(String, Iterable[Int])] = Array((A,CompactBuffer(1, 1)),
(B,CompactBuffer(1, 1, 1, 1, 1,1)))
-rw-r--r-- 1 root root 24 May 14 10:57 part-00000
-rw-r--r-- 1 root root 36 May 14 10:57 part-00001
-rw-r--r-- 1 root root 0 May 14 10:57 part-00002
-rw-r--r-- 1 root root 0 May 14 10:57 part-00003
```

```
-rw-r--r-- 1 root root 0 May 14 10:57 part-00004

// 案例3 salted keys
val u2 = u1.map(x => (Random.nextInt(100).toString+x ,1))
val u2 = u1.map(x => (x+Random.nextInt(10).toString,1))
u2.groupByKey().saveAsTextFile("/root/skew6")
```

- **Load CSV files(加载CSV文件)**

```
// 案例1
val lines = sc.textFile("file:///usr/data/users.csv")

// 案例2
val fields = lines.mapPartitionsWithIndex((idx, iter) => if (idx == 0)
iter.drop(1) else
iter).map(l => l.split(","))

//案例3
val df =
spark.read.format("csv").option("delimiter",",").option("header","true").load("hd
fs://user/data/users.csv").select("event_id","user_id","start_time").show

//案例4
val df =
spark.read.format("csv").option("header","false").load("hdfs://user/wayne/data/us
ers.csv")
.withColumnRenamed("_c0", "user_id")
.withColumnRenamed("_c1", "user_name")
.withColumnRenamed("_c2", "user_gender")
```

- **Load JSON Files(加载JSON文件)**

```
//案例1
{"string":"string1","int":1,"array":[1,2,3],"dict": {"key": "value1"}}
{"string":"string2","int":2,"array":[2,4,6],"dict": {"key": "value2"}}
{"string":"string3","int":3,"array":[3,6,9],"dict": {"key": "value3",
"extra_key": "extra_value3"}}
val df = spark.read.json("single.json")
df.printSchema

//案例2
[
    {"string":"string1","int":1,"array":[1,2,3],"dict": {"key": "value1"}},
    {"string":"string2","int":2,"array":[2,4,6],"dict": {"key": "value2"}},
    {
        "string": "string3",
        "int": 3,
        "array": [
```

```
            3,
            6,
            9
        ],
        "dict": {
            "key": "value3",
            "extra_key": "extra_value3"
        }
    }
]

val mdf = spark.read.option("multiline", "true").json("multi.json")
mdf.show(false)
```

作业 (Assignment)

- 给一篇文章，找到那些单词是重复的
- 看作业文件