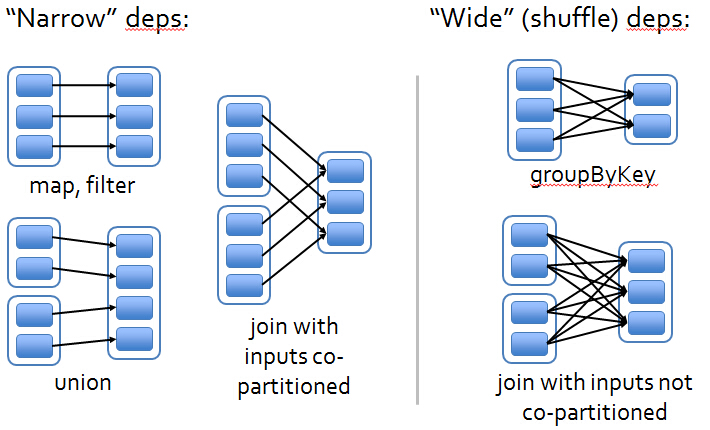
**第二十二课：RDD依赖关系彻底解密**

概要：

1. RDD依赖关系的本质内幕
2. 依赖关系下的数据流视图
3. 经典的RDD依赖关系解析
4. RDD依赖关系源码内幕

**1. RDD依赖关系的本质内幕**



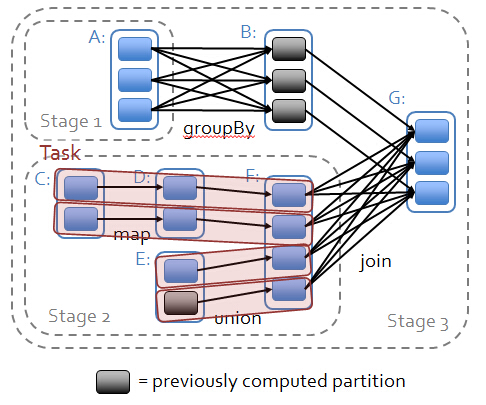
1,窄依赖是指每个父RDD的一个Partition最多被子RDD的一个Partition所使用，例如map、filter、union等都会产生窄依赖；union是将多个RDD合并成一个RDD，父RDD不变化

2，宽依赖是指一个父RDD的Partition会被多个子RDD的Partition所使用，例如groupByKey、reduceByKey、sortByKey等操作都会产生宽依赖

总结：如果父RDD的一个Partition被一个子RDD的Partition所使用就是窄依赖，否则的话就是宽依赖。**如果子RDD中的Partition对父RDD的Partition依赖的数量不会随着RDD数据规模的改变而改变的话，就是窄依赖，否则的话就是宽依赖。**

特别说明：对join操作有两种情况，如果说join操作的使用每个partition仅仅和已知的Partition进行join，此时的依赖是固定的，不会随着数据的规模而改变的，此时是join操作就是窄依赖；其它情况的join操作就是宽依赖；因为是确定的partition数量的依赖关系，所以就是窄依赖，得出一个推论，窄依赖不仅包含一对一的窄依赖，还包含一对固定个数的窄依赖（也就是说对父RDD的依赖的Partition的数量不会随着RDD数据规模的改变而改变）

**2. 依赖关系下的数据流视图**



**为什么不可以将所有的RDD划分成一个stage？**

遇到Shuffle级别的依赖关系必须计算依赖的RDD的所有的Partitions,并且都发生在一个Task中计算，所以task会过大。RDD在计算的过程中会回溯，RDD之间存在血统关系，因此就会造成重复计算。**上面两种假设的核心问题是在遇到Shuffle依赖的时候无法进行pipeline.**

注意：

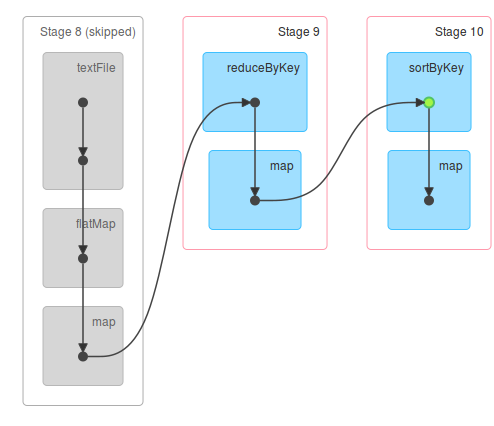
1. 从后往前推理，遇到宽依赖就断开，遇到窄依赖就把当前的RDD加入到该Stage中；

**2，** **每个Stage里面的Task的数量是由该Stage中最后一个RDD的Partition的数量所决定的！**

3，最后一个Stage里面的任务的类型是ResultTask，前面其它所有的Stage里面的任务的类型都是ShuffleMapTask；

4，**代表当前Stage的算子一定是该Stage的最后一个计算步骤！！！**

**3. 经典的RDD依赖关系解析**



补充：Hadoop中的MapReduce操作中的Mapper和Reducer在Spark中基本等量算子是：map、reduceByKey；

**表面上是数据在流动，实质上算子在流动：**

1. 数据不动代码动；
2. 在一个Stage内幕算子为何会流动（Pipeline）？首先是算子合并，也就是所谓的函数式编程的执行的时候最终进行函数的展开从而把一个Stage内部的多个算子合并成为一个大算子（其内部包含了当前Stage中所有算子对数据的计算逻辑）；其次是由于Tranformation操作的Lazy特性！！！在具体算子交给集群的Executor计算之前首先会通过Spark Framework（DAGScheduler）进行算子的优化**（基于数据本地性的Pipeline）**

**为啥进行pipeline不会产生中间结果呢？**

我们从实验的角度来看：在stage内部，它会把多个算子合并成一个算子，这是函数式编程决定的。

**为啥为什么写了很多函数它不执行呢？**

是因为框架帮我们管理了这些算子，开始的时候这些算子还没有执行，框架根据自己的调度算法，这也就是框架的引擎了，引擎会将这些算子展开变成一个算子，然后它又不产生中间结果，而且是基于数据本地性。

**4. RDD依赖关系源码内幕**

**reduceByKey的源码内幕：**

|  |
| --- |
| **def** combineByKeyWithClassTag[C](  createCombiner: V => C,  mergeValue: (C, V) => C,  mergeCombiners: (C, C) => C,  partitioner: Partitioner,  mapSideCombine: Boolean = **true**, |

\* - `createCombiner`, which turns a V into a C (e.g., creates a one-element list)

\* - `mergeValue`, to merge a V into a C (e.g., adds it to the end of a list)

\* - `mergeCombiners`, to combine two C's into a single one.

combineByKeyWithClassTag是把包含相同K的record aggregation聚合在一起，然后再在聚合后的结果执行不同的计算逻辑，reduceByKey是边抓数据边计算，这样会提升效率。

我们假设说有K相同的(K,V)记录，数据不断的流向我们的combineByKeyWithClassTag的话，createCombiner会将第一个record流过来的数据初始化，记录到value初始化，然后第二个record进来的时候，它就开始mergeValue，以此来更新记录的值。这样的话就变抓数据边更新值。等到相同K的value全部进行mergeValue之后，得出最后一个结果。同样的原理，也会有不同K的value流进来，最后我们会把不同的结果，但是相同的K，然后再进行最后的mergeCombiners。

**依赖关系源码解读**

NarrowDependency 返回子RDD的partitionId依赖的所有的parent RDD Partitions

|  |
| --- |
| **abstract** **class** NarrowDependency[T](\_rdd: RDD[T]) **extends** Dependency[T] {  /\*\*  \* Get the parent partitions for a child partition.  \* @param partitionId a partition of the child RDD  \* @return the partitions of the parent RDD that the child partition depends upon  \*/  **def** getParents(partitionId: Int): *Seq*[Int] //**返回Seq序列** |

**窄依赖分为两种：**

OneToOneDependency：

Represents a one-to-one dependency between partitions of the parent and child RDDs.

|  |
| --- |
| **class** OneToOneDependency[T](rdd: RDD[T]) **extends** NarrowDependency[T](rdd) {  **override** **def** getParents(partitionId: Int): *List*[Int] = List(partitionId) **//直接将id返回回来**  } |

RangeDependency：对父RDD的依赖关系是有范围的，例如union

将两个RDD联合在一块。通过注释可以很好的看出

|  |
| --- |
| /\*\*  \* Represents a one-to-one dependency between ranges of partitions in the parent and child RDDs.  \* @param rdd the parent RDD  \* @param inStart the start of the range in the parent RDD  \* @param outStart the start of the range in the child RDD  \* @param length the length of the range  \*/  @DeveloperApi  **class** RangeDependency[T](rdd: RDD[T], inStart: Int, outStart: Int, length: Int)  **extends** NarrowDependency[T](rdd) {  **override** **def** getParents(partitionId: Int): *List*[Int] = {  **if** (partitionId >= outStart && partitionId < outStart + length) {  List(partitionId - outStart + inStart)  } **else** {  **Nil**  }  }  } |

ShuffleDependency：就是子RDD依赖parent RDD所有的partitioner

|  |
| --- |
| **class** ShuffleDependency[K: ClassTag, V: ClassTag, C: ClassTag](  @transient **private** **val** \_rdd: RDD[\_ <: Product2[K, V]],  **val** partitioner: Partitioner,  **val** serializer: Option[Serializer] = **None**,  **val** keyOrdering: Option[*Ordering*[K]] = **None**,  **val** aggregator: Option[**Aggregator**[K, V, C]] = **None**,  **val** mapSideCombine: Boolean = **false**)  **extends** Dependency[Product2[K, V]] {  **override** **def** rdd: RDD[Product2[K, V]] = \_rdd.asInstanceOf[RDD[Product2[K, V]]]  **private**[spark] **val** keyClassName: *String* = reflect.classTag[K].runtimeClass.getName  **private**[spark] **val** valueClassName: *String* = reflect.classTag[V].runtimeClass.getName  // Note: It's possible that the combiner class tag is null, if the combineByKey  // methods in PairRDDFunctions are used instead of combineByKeyWithClassTag.  **private**[spark] **val** combinerClassName: Option[*String*] =  Option(reflect.classTag[C]).map(\_.runtimeClass.getName)  **val** shuffleId: Int = \_rdd.context.newShuffleId()  **val** shuffleHandle: ShuffleHandle = \_rdd.context.env.shuffleManager.registerShuffle(  shuffleId, \_rdd.partitions.size, **this**)  \_rdd.sparkContext.cleaner.foreach(\_.registerShuffleForCleanup(**this**))  } |