分布式文件计算项目报告

1831588 张志强 1811018 王冠淞 1831587 吕金华

1. 分布式存储

1.1 系统设计

1.1.1 架构

我们设计的分布式文件系统为主从(master/slave)架构。其中，整个系统中只有一个 master 角色，其它扮演 slave 角色，整个系统在 master 与 slave、slave 与 slave 的交互下工作。其中，master 节点存储着整个系统文件的 meta 信息(每个文件的分块情况、存储位置等)，接收所有对于文件系统的访问请求，包括文件上传、下载、修改等，以及监控着集群的实时状态；slave 节点负责存储具体的文件数据，处理所有文件的上传、下载、修改等请求，并实时汇报自己的状态。整个分布式文件系统对外提供统一的接口，逻辑上等同于一个文件系统。在内部，大文件会被切割成多个分块，每个文件分块存储在不同的 slave 节点上，以实现系统的负载均衡。同时，每个文件分块会有多个副本，存储在不同 slave 节点上，避免节点宕机造成的数据损坏。所有文件的元信息存储在 master 节点上，并由它负责管理。主从架构的系统中比端到端的去中心化系统更加方便易用，但主节点的正常运行对于整个系统来说至关重要。因此对于主节点我们设计一个次主节点，定期从主节点备份元信息，一旦主节点宕机，次主节点可以接管 master 的角色，保证系统的正常运行。

1.1.2 文件分块

文件分块的策略是为了实现系统的负载均衡。生产环境的分布式文件系统往往包含数百上千个节点，存储着海量的大文件数据，文件大小甚至可以达到 TB、PB 级别。我们设计的文件系统也应当满足负载均衡的需求，避免大文件存储在单个节点上造成的不均衡现象。

我们设置系统配置默认的文件分块大小为 128MB。分块过大会导致大文件无法均匀存储在 slave 节点上，造成节点负载的不均衡；分块过小会导致系统中存在过多的分块，造成 master 节点文件元信息数据过大，管理难度增加。不过，由于系统中可能会存在很多小于 128MB 的文件，我们允许文件的最后一个分块大小小于默认分块大小，优化了这部分的存储效率。

在具体的某个上传操作中，客户端可以指定这次上传默认的分块大小。一般来说，除非有特殊的生产环境需求，一般情况下不需要改变分块大小。根据分块大小，master 节点将文件的分块策略及访问秘钥返回给客户端，由客户端将文件分成一个个分块，并和对应的 slave 节点验证秘钥后完成分块的上传。

下面图 1 是文件分块的一个设计原型。

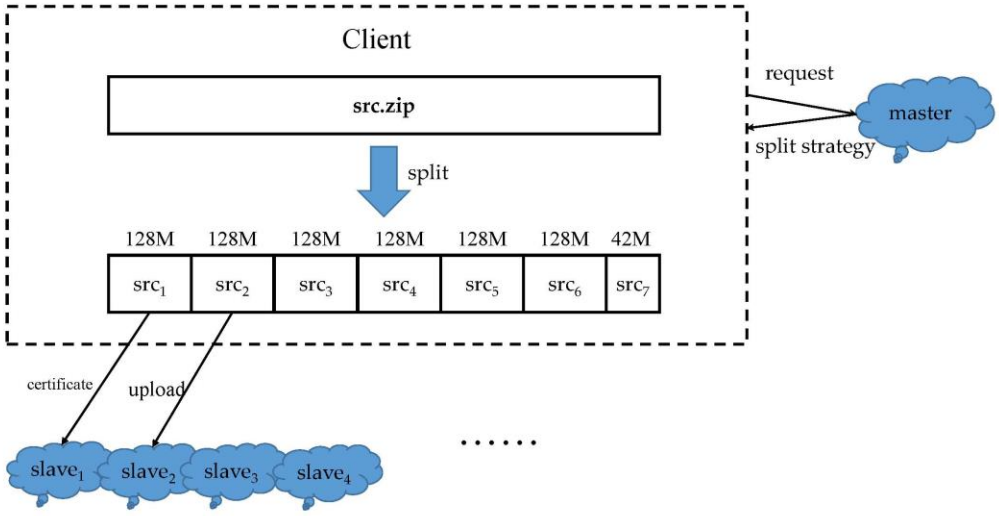


图 1. 文件系统分块设计

其中，客户端待上传文件 src.zip 大小为 810MB。客户端向 master 请求上传，master 节点返回分块策略给客户端。客户端按照默认分块大小，将数据分成 6 份 128MB 的分块和一份 42MB 的分块。最后一份 42MB 的的数据不足 128MB，为了节约空间，也作为一份分块存储。之后客户端根据分配策略，将每个分块上传到对应的 slave 节点上，当每个分块都上传成功时，客户端发送上传成功信息给 master 节点，master 将对应的元信息存储到元信息数据库中，完成整个文件上传分块过程。

数据分块策略有以下优点：

- 大文件与小文件的统一管理，便于 master 的操作
- 数据的容灾与备份，在发生意外宕机时能够从其它节点上恢复分块数据，降低数据损坏的概率
- 超大文件的有效存储，可以作为多个分块存储在不同机器上

1.1.3 文件备份

文件备份是数据容错的有效手段。当某个 slave 节点宕机时，虽然该节点上的所有分块数据都会丢失，但是其他 slave 节点上存储了它们的备份数据，因而系统可以正常运行，可以正常上传下载修改文件，不受影响。同时，每隔一段时间，每个 slave 节点会向 master 节点发送心跳数据，表明自己正常运行。这样，当

master 节点没有接收到某个节点的信号时，可以判定这个节点宕机了，通过元数据中找到这个节点的分块信息列表，将这些分块在其它机器上的备份再复制一份，存储到其它节点上，从而保持分块的备份数量不变。

具体设计而言，文件备份过程按照以下步骤完成：

1. 当客户端发送请求上传文件时，请求首先由 master 节点接收，计算分块结果、副本分配以及访问秘钥，将结果返回给客户端。秘钥具有时效性，一段时间后秘钥就会失效，因此上传请求需要在给定时间内完成。

2. 客户端接收到分块策略和访问秘钥时，将文件按照分块策略截成一段段字节流，验证秘钥通过后，通过 tcp 协议传输到对应 slave 节点的分布式文件系统中。

3. 当每个分块写入分布式文件系统中时，其副本会按照副本策略通过另外一个进程上传到对应的节点上，从而完成副本的批量写入。

4. 多个分块的多个副本作为独立的任务，在完成时会向客户端发送通知，客户端确认所有任务都顺利完成之后，关闭文件的访问，并通知 master 节点上传完毕，master 回收访问秘钥，文件上传和备份完成。

下面图 2 是文件备份的一个设计原型。

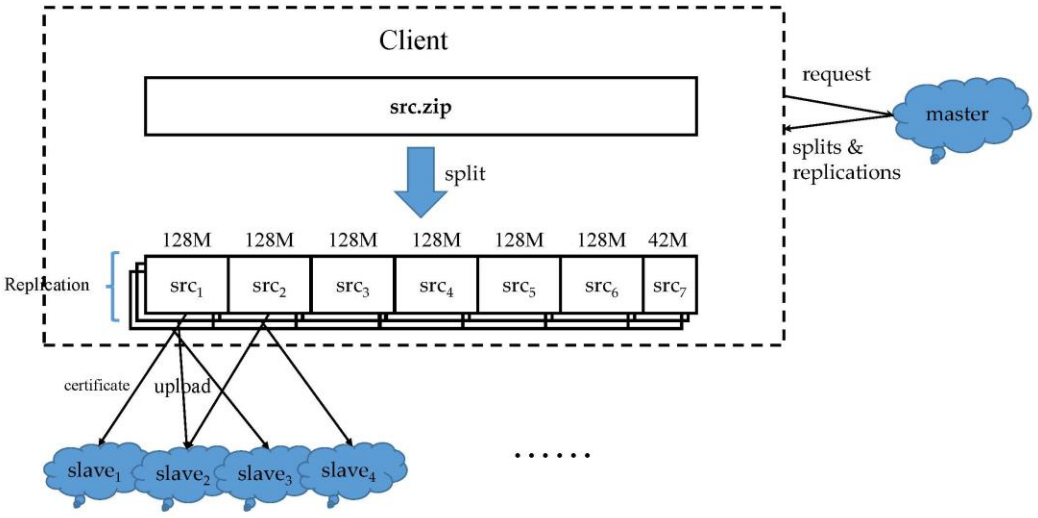


图 2. 文件系统备份设计

在文件上传时，上传副本和上传分块是同步完成的。以上图为例，每个分块有两个副本，一共存储三份，每份存储到不同的节点上，从而保证节点失效时数据的有效访问。

1.1.4 宕机备份

宕机备份是系统中节点失效时的备份手段。对于大型系统而言，宕机有效备份是保证系统在生产环境中高可用性的重要手段。对于一个 slave 节点失效的情况，需要通过宕机备份将它存储的分块转移到其它节点上，保持所有分块的副本数目一致。

一段时间内，若 master 节点接收不到某 slave 节点 A 定期发送的心跳数据时，认定这个节点 A 失效。在 master 节点的元数据库中有着对应每个 slave 节点分块的索引信息，可以通过索引高效获取每个节点存储所有分块的信息。通过索引节点 A 对应的所有分块信息，找到每个分块在其它节点上的备份。master 节点重新计算新的备份的分配策略，通过一个特殊的宕机备份恢复进程，直接从这些 slave 节点将分块传输到分配策略给定的节点上，不需要 master 节点其它的参与。

下面图 3 是宕机备份的一个设计原型。

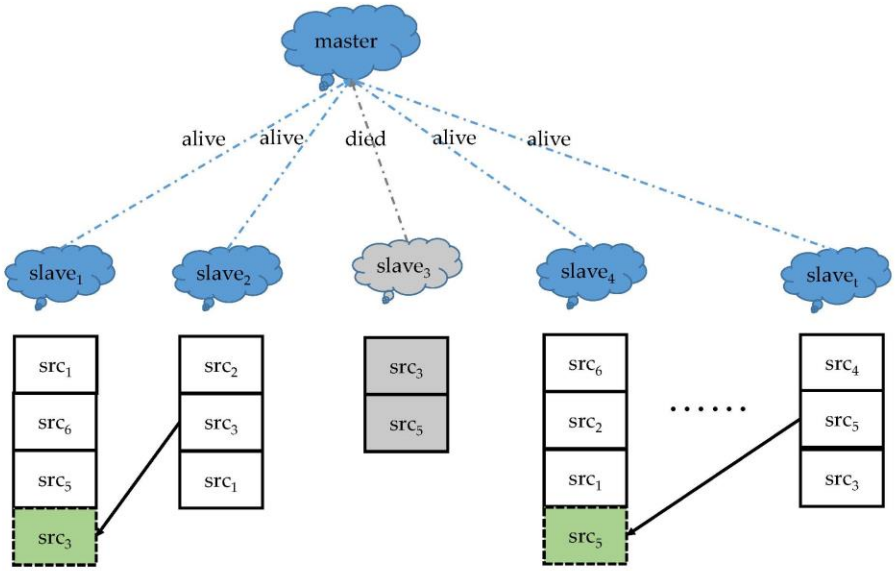


图 3. 文件系统宕机备份设计

master 定期检查节点健康状态，变为灰色表明节点宕机。slave3 宕机，该节点上所有的分块全部丢失。但是由于 master 节点上保存所有文件的元信息，可以方便的找到丢失分块的备份信息。例如丢失的 3 号和 5 号分块可以分别在 2 号和 t 号节点上找到。为了维持系统备份数量的一致性，保证系统的高可用性，需要将 3 号和 5 号分块再备份一次，绿色分块就是新创建的分块，分块的上传是直接在 slave 节点之间完成的，不需要 master 节点额外的参与。

1.1.5 文件一致性

当对分布式文件系统中的某个文件进行修改时，需要对所有的备份进行更新，保证数据一致性。数据一致性在数据库领域中有着成熟现成的应用和理论支持。数据一致性包括读一致性和写一致性，一般的应用程序需要满足支持并发的读请求和原子的写请求，这样才能满足数据的强一致性。数据的弱一致性可以允许并发的写请求，只要求最终到达数据的一致性。然而，弱一致性模型有着一定的出错概率，需要设计额外的出错修复程序，在出错严重的情况下还需要手动修复。在实时计算的需求下，数据的强一致性是计算不出错的前提。因此我们选择强一致性模型来实现我们的文件管理。

强一致性具体需要满足两个要求：

1. 写一致性：对于一个文件，同时只允许进行一个写操作。若有多个写操作并发，必须排队进行执行。单次写操作完成之后，对于修改的分块，需要同时将它的分块进行修改，修改完毕之后单次写操作才能完成，释放资源。

2. 读一致性：对于读文件请求来说，不需要等对该文件的写操作完成之后才能执行。读请求获取文件分块的所有副本中时间戳最近的一个，作为当前可见的最新版本，返回给客户端。读操作可以并发执行，不需要排队。

1.2 系统实现

为了确保开发的时效性和系统的高可用性已经扩展性，我们利用现有的 Hadoop 分布式框架进行开发，在其之上建立我们的分布式文件系统。整个系统有 1 个 master 节点，一个次 master 节点，和 10 个 slave 节点。

1.2.1 架构

整个系统的架构如下图 4 所示。

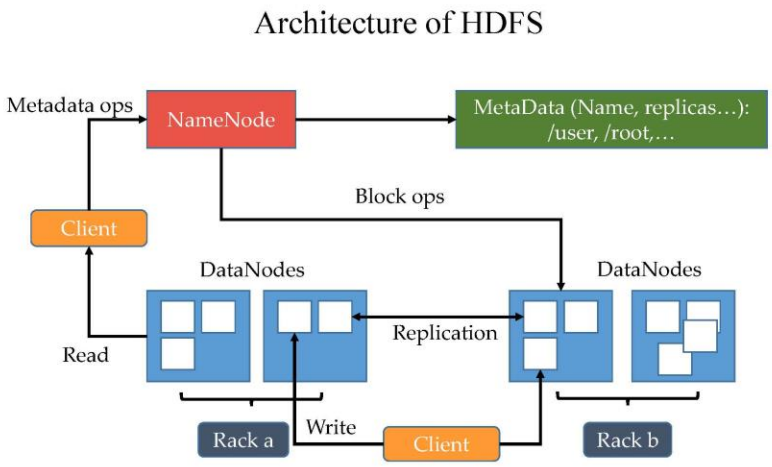


图 4. HDFS 系统架构

值得注意的是，与我们的设计不同，hdfs 利用机架感知将数据节点划分为不同的分组，数据在相同分组内传输更快。倾向于将分块的备份存储在不同的分组内，以实现容灾备份。

1.2.2 文件分块备份

整个分布式文件系统统一对外提供 FileSystem 类，完成客户端的各类请求。然而，在系统内部，通过 DistributedFileSystem (DFS)类完成内部操作。HDFS 的文件分块备份流程如下：

1. HDFS 客户端程序通过 DFS API 向 NameNode 发送一个创建文件请求。这个请求以 RPC 调用的方式传递到 NameNode

2. NameNode 执行各种检查，包括名称检查，权限检查等。若满足要求时，NameNode 记录下这条文件创建请求记录；若要求不满足，客户端 DFS API 抛出 IOException 异常

3. DFS API 向客户端返回一个 FSDataOutputStream 类来写入文件数据。FSDataOutputStream 类负责将文件数据切成分块，包装成 tcp 数据包，放入内部的数据队列，供数据调度器 DataStreamer 调用。DataStreamer 类和 NameNode 沟通，分配对应的 DataNode 队列来上传分块。对应备份数量不为 1 的情况，分配的对应备份数量的 DataNode 队列，同时将分块备份上传。

4. DataNode 队列形成一个管道(Pipeline)，管道中有 Replication 项。DataStreamer 将流式 tcp 数据包传输到 Pipeline 中的第一个 DataNode，该数

据节点存储数据包并将它转发到 Pipeline 中的第二个数据节点，如此重复，直到传递到第 Replication 个 DataNode。

5. DataNode 存储好数据后，会返回确认信息给 FSDataOutputSteam 类。一个数据包只有在对应 Pipeline 消耗完时才会从队列中删除。

6. 客户端写完数据后，在流上调用 close()，文件上传结束。

HDFS 分块备份过程如下图 5 所示。

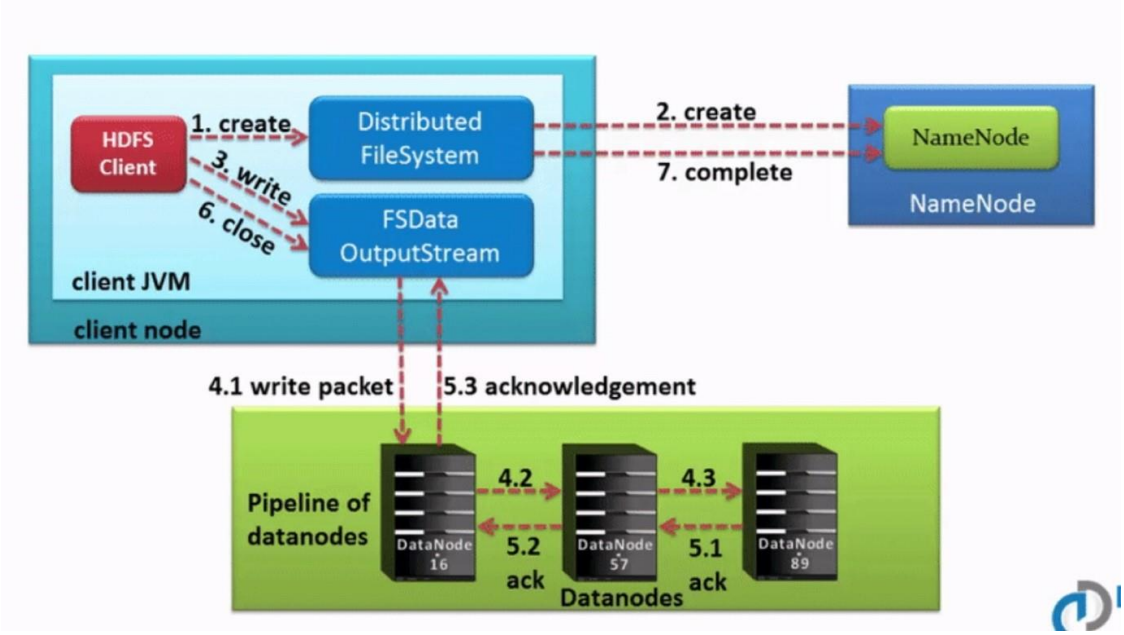


图 5. HDFS 分块备份

操作代码示意如下：

```
Configuration conf = new Configuration();
conf.addResource("core-site.xml");
conf.set("dfs.blocksize", blockSize);
conf.set("dfs.replication", replication);
FileSystem hdfs = FileSystem.get(conf);

if (dest.charAt(dest.length() - 1)!= "/")
    dest += "/" + newName;
else
    dest += newName;

Path path = new Path(dest);
if (hdfs.exists(path)){
    if (!overwrite){
        System.out.println("File " + dest + "already exists");
```

```
            return false;
        }else{
            System.out.println("File" + dest + "already exists, but will be
 overwritten")
        }
}

FSDataOutputStream out = hfds.create(path);
InputStream in = new BufferedInputStream(new FileInputStream(new File(s
rc)));
byte []b = new byte[1024];
int numBytes = 0;
while((numBytes = in.read(b)) > 0){
    out.write(b, 0, numBytes);
}

in.close();
out.close();
hdfs.close();
```

1.2.3 文件一致性

HDFS 中遵从一次写多次读的原则，在 HDFS 中更新文件有以下两种策略：

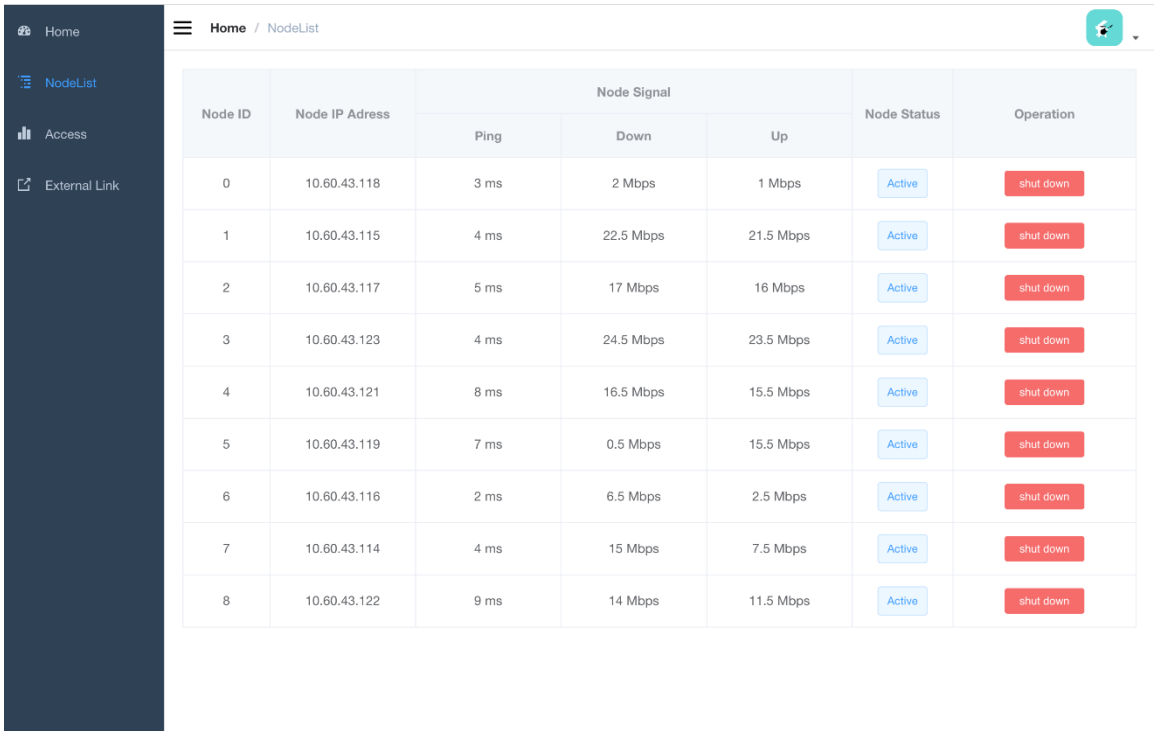
A 追加到文件尾

由于追加到文件尾，影响的是文件的最后一个分块，只需要将追加数据写入到最后一个分块就可以完成追加数据的需求。注意，HDFS 中数据分块和本地文件系统的格式兼容，可以直接通过本地文件系统操作添加数据。写操作是原子性的，若有并发请求，则排队执行。

B 下载到本地修改放回

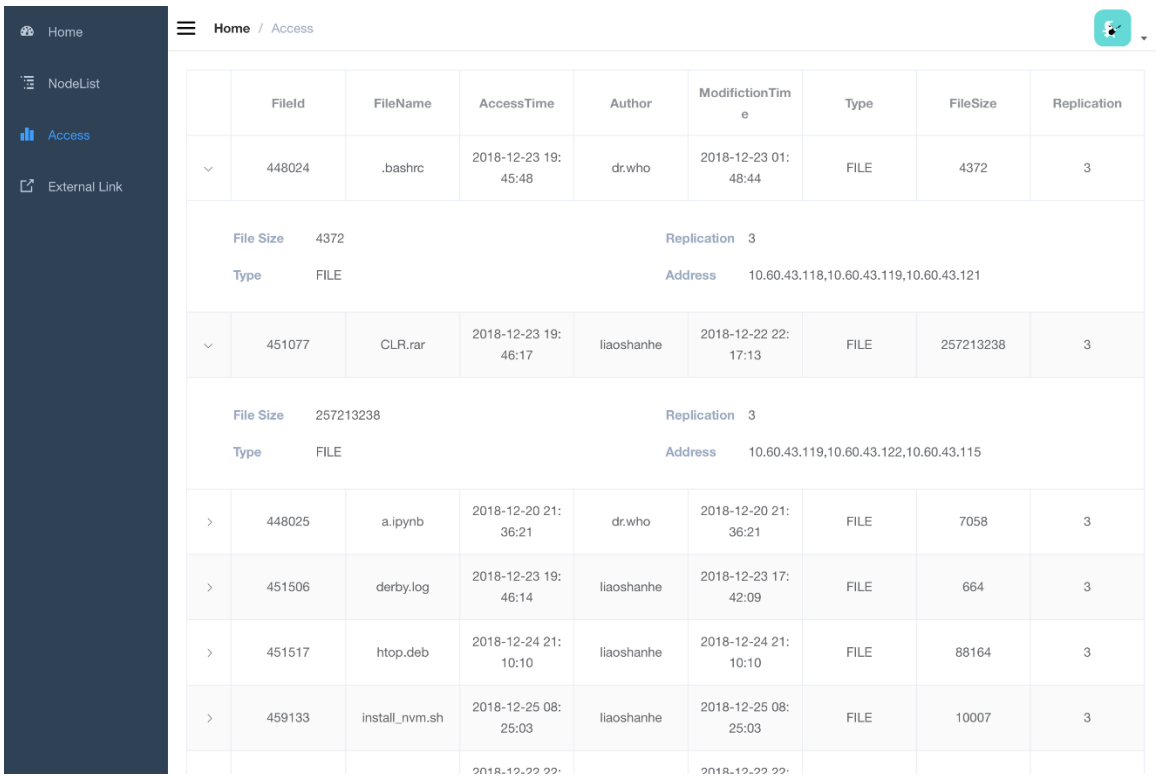
注意，由于 hdfs 不支持同名文件覆盖，因此在修改后放回前，需要将 hdfs 中对应文件删除，或者重命名文件。

1.2.4 界面可视化



Node ID	Node IP Adress	Node Signal			Node Status	Operation
		Ping	Down	Up		
0	10.60.43.118	3 ms	2 Mbps	1 Mbps	Active	shut down
1	10.60.43.115	4 ms	22.5 Mbps	21.5 Mbps	Active	shut down
2	10.60.43.117	5 ms	17 Mbps	16 Mbps	Active	shut down
3	10.60.43.123	4 ms	24.5 Mbps	23.5 Mbps	Active	shut down
4	10.60.43.121	8 ms	16.5 Mbps	15.5 Mbps	Active	shut down
5	10.60.43.119	7 ms	0.5 Mbps	15.5 Mbps	Active	shut down
6	10.60.43.116	2 ms	6.5 Mbps	2.5 Mbps	Active	shut down
7	10.60.43.114	4 ms	15 Mbps	7.5 Mbps	Active	shut down
8	10.60.43.122	9 ms	14 Mbps	11.5 Mbps	Active	shut down

这是我们的集群配置，可以看到一共有 9 台物理机器作为数据节点，同时显示机器的网速带宽等信息。



	FileId	FileName	AccessTime	Author	ModifictionTime	Type	FileSize	Replication
⌄	448024	.bashrc	2018-12-23 19:45:48	dr.who	2018-12-23 01:48:44	FILE	4372	3
	File Size 4372				Replication 3			
	Type FILE				Address 10.60.43.118,10.60.43.119,10.60.43.121			
⌄	451077	CLR.rar	2018-12-23 19:46:17	liaoshanhe	2018-12-22 22:17:13	FILE	257213238	3
	File Size 257213238				Replication 3			
	Type FILE				Address 10.60.43.119,10.60.43.122,10.60.43.115			
›	448025	a.ipynb	2018-12-20 21:36:21	dr.who	2018-12-20 21:36:21	FILE	7058	3
›	451506	derby.log	2018-12-23 19:46:14	liaoshanhe	2018-12-23 17:42:09	FILE	664	3
›	451517	htop.deb	2018-12-24 21:10:10	liaoshanhe	2018-12-24 21:10:10	FILE	88164	3
›	459133	install_nvm.sh	2018-12-25 08:25:03	liaoshanhe	2018-12-25 08:25:03	FILE	10007	3

这是分布式文件系统的界面，我们实现一层扁平化的文件系统。界面展示了所有文件的元数据信息，包括其创建时间、大小、分块数量、每个分块副本的位置等。

1.2.5 系统配置

- dfs.blocksize：默认块大小，可以用一个字母表示单位，例如 k、m、g 等，如果没有表示以字节为单位。

```
<property>
    <name>dfs.blocksize</name>
    <value>128m</value>
</property>
```

- dfs.replication: 默认分块存储数量。对于一次上传操作，可以指定分块存储数量，否则使用默认值。

```
  <property>
    <name>dfs.replication</name>
    <value>3</value>
</property>
```

2. 分布式计算

我们在 Hadoop 分布式文件系统的基础上搭建了 spark 分布式计算框架，并利用 yarn 来管理分布式计算资源。Spark Driver 首先 作为一个 ApplicationMaster 在 Yarn 集群中启动，其实就是把 spark 作为一个客户端提交作业给 Yarn,实际运行程序的是 Yarn。并且我们使用的 Yarn 模式是 cluster 模式，即我们的 sparkdriver 是运行在 NodeManager 的 ApplicationMaster 中，而不是客户端的进程，具体关系如图 6。

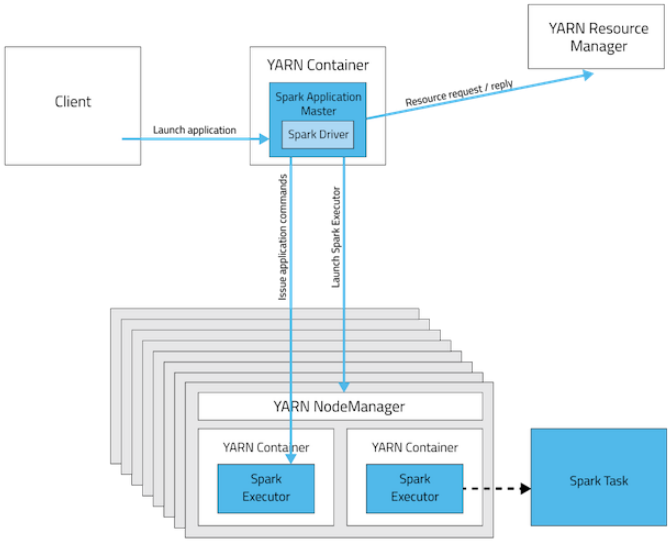


图 6. yarn cluster mode

我们使用的 Apache spark 版本是 2.2.0.

2.1 Spark 框架介绍

Apache Spark 是专为大规模数据处理而设计的快速通用的计算引擎，是一个开源的类 Hadoop MapReduce 的通用并行框架。Spark 拥有 Hadoop MapReduce 所具有的优点；但不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中，从而不再需要读写 HDFS，因此计算速度提升巨大。并且它对 MapReduce 做了更高一层的抽象，提供了更丰富的 API，更有利于开发者进行使用。

想要了解 spark，就要明白它的核心概念——RDD (Resilient Distributed Datasets) 。虽然 spark 在 2.1 版本后使用速度更快的 DataFrame，我们本文使用的也是 DataFrame 来表示数据和进行计算。但是 DataFrame 其实是基于 RDD 实现的，只不过 DataFrame 提供了详细的结构信息 (Schema) 来帮助 Spark SQL 了解数据集中包含哪些列及其名称和数据类型。RDD 是一个只读的数据集合，被分块存储在多个机器之上，对数据集进行操作和变为对 RDD 的转换。

如果 RDD 有个分块丢失，丢失的部分可以被重建。对于很多相同的数据项使用同一种操作 (如 map/filter/join) ，这种方式能够通过记录 RDD 之间的转换从而刻画 RDD 的继承关系 (lineage) ，而不是真实的数据，最终构成一个 DAG (有向无环图) ，而如果发生 RDD 丢失，RDD 会有充足的信息来得知怎么从其他 RDDs 重新计算得到。

在 spark 中，对 RDD 的操作可以分为两种，即 Transformation 和 Action：

2.1.1 Transformation

对于 Transformation 操作是指由一个 RDD 生成新 RDD 的过程，其代表了是计算的中间过程，其并不会触发真实的计算 (Transformation 是惰性的) 。其中一些典型的操作如下：

- map (f:T=>U) : RDD[T]=>RDD[U]

返回一个新的分布式数据集，由每个原元素经过 func 函数转换后组成

- filter (f:T=>Bool) : RDD[T]=>RDD[T]

返回一个新的数据集，由经过 func 函数后返回值为 true 的原元素组成

- flatMap (f:T=>Seq[U]) : RDD[T]=>RDD[U])

类似于 map，但是每一个输入元素，会被映射为 0 到多个输出元素 (因此，func 函数的返回值是一个 Seq，而不是单一元素)

- sample (withReplacement: Boolean, fraction: Double, seed: Long) : RDD[T]=>RDD[T]

sample 将 RDD 这个集合内的元素进行采样，获取所有元素的子集。用户可以设定是否有放回的抽样、百分比、随机种子，进而决定采样方式。

- groupByKey ([numTasks]) :RDD[(K,V)]=>RDD[(K,Seq[V])]

在一个由 (K,V) 对组成的数据集上调用，返回一个 (K，Seq[V]) 对的数据集。

- reduceByKey (f: (V,V) =>V, [numTasks]) : RDD[(K, V)]=>RDD[(K, V)]

在一个 (K，V) 对的数据集上使用，返回一个 (K，V) 对的数据集，key 相同的值，都被使用指定的 reduce 函数聚合到一起。和 groupbykey 类似，任务的个数是可以通过第二个可选参数来配置的。

- union (otherDataset) : (RDD[T],RDD[T]) =>RDD[T]

返回一个新的数据集，由原数据集和参数联合而成

- join (otherDataset, [numTasks]) : (RDD[(K,V)],RDD[(K,W)]) =>RDD[(K, (V,W))]

返回 key 值相同的所有匹配对

……

2.1.2 Action

Action 操作代表一次计算的结束，不再产生新的 RDD，将结果返回到 Driver 程序。所以 Transformation 只是建立计算关系，而 Action 才是实际的执行者。每个 Action 都会调用 SparkContext 的 runJob 方法向集群正式提交请求，所以每个 Action 对应一个 Job。典型的 Action 操作有下面几个：

- count() : RDD[T]=>Long

  返回数据集的元素个数

- countByKey() : RDD[T]=>Map[T, Long]

  对(K,V)类型的 RDD 有效，返回一个(K，Int)对的 Map，表示每一个 key 对应的元素个数

- collect() : RDD[T]=>Seq[T]

  在 Driver 中，以数组的形式，返回数据集的所有元素。这通常会在使用 filter 或者其它操作并返回一个足够小的数据子集后再使用会比较有用。

- reduce(f:(T,T)=>T) : RDD[T]=>T

  通过函数 func（接受两个参数，返回一个参数）聚集数据集中的所有元素。这个功能必须可交换且可关联的，从而可以正确的被并行执行。

通过了解一些基本的 RDD 操作，我们可以进一步了解 RDD 之间关系的宽依赖和窄依赖。简单概括，区分宽窄依赖主要就是看父 RDD 的一个 Partition 的流向，要是流向一个的话就是窄依赖，流向多个的话就是宽依赖。

比如下面的图 7 就是**宽依赖(Shuffle 依赖)**，因为父 RDD 中一个分区内的数据会被分割，发送给子 RDD 的所有分区，即存在 shuffle 的过程。

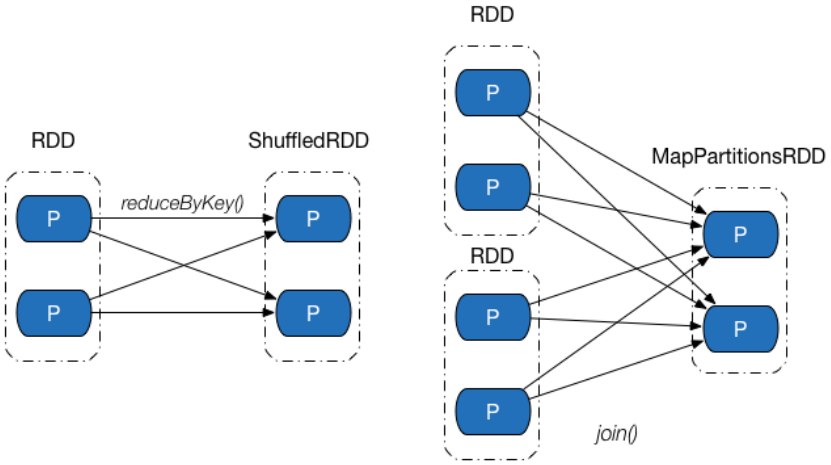


图 7. RDD 宽依赖

与之相对的，**窄依赖**中的一个分区最多只会被子 RDD 中的一个分区使用父 RDD 中，一个分区内的数据是不能被分割的。例子如图 3 所示：

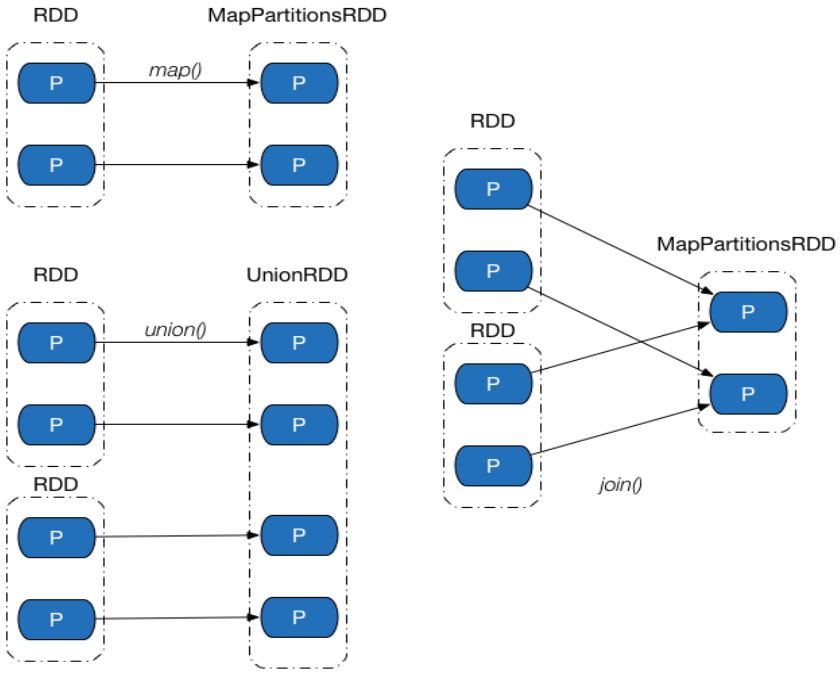


图 8. RDD 窄依赖

了解了 RDD 之间的宽窄依赖，我们可以学习 spark 里面每个 job 的 stage 的划分。DAGScheduler 会把 DAG 划分相互依赖的多个 stage，划分 stage 的依据就是 RDD 之间的宽窄依赖。遇到宽依赖就划分 stage,每个 stage 包含一个或多个 task 任务。然后将这些 task 以 taskSet 的形式提交给 TaskScheduler 运行。stage 是由一组并行的 task 组成。运行到每个 stage 的边界时，数据在父 stage 中按照 Task 写到磁盘上，而在子 stage 中通过网络按照 Task 去读取数据。这些操作会导致很重的网络以及磁盘的 I/O，所以 stage 的边界是非常占资源的，在编写 Spark 程序的时候需要尽量避免的 。

2.2 计算任务及结果

根据附录中的数据文件（一个表格，内容为通话记录，主要包含字段：主叫号码、通话开始时间、通话时长、通话类型、主叫号码运营商等），使用分布式系统的计算资源，完成以下计算：

2.2.1 平均通话次数

计算思路是将源数据先根据主叫号码和日期 id（day_id）分组聚合，得到每个主叫号码每天的通话数，让后将结果作为子查询再根据主叫号码分组聚合计算每个号码总共的通话天数和总通话次数，这两个结果相除就可以得出平均每天的通话次数。核心代码为：

```
def qes_1(spark):
    print("Start computing question 1: ")
    qes1=spark.sql("""
    select calling_nbr, sum(call_num)/count(1) as avg_calling
    from
    (
        select calling_nbr, day_id as day,count(1) as call_num
        from calling
        group by calling_nbr,day_id
    )
    group by calling_nbr
    """)
    qes1.write.csv("dc2019/qes_1.csv",mode='overwrite')
print("Question 1 completed")
```

从图 9 的任务 1 的 RDD 关系（DAG，有向无环图）可以看出这个任务的 Job 被分为了 3 个阶段——stage，这是因为代码中 RDD 经过了 2 次 shuffle 依赖变换，即 group 语句，其余的 map 操作(图 10)都对应与 SQL 查询中的 select 语句。

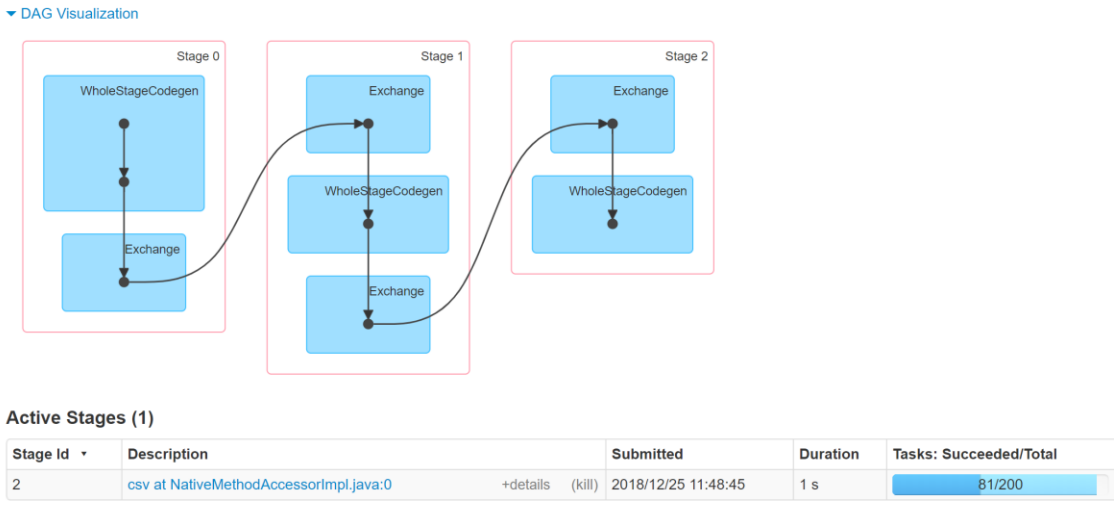


图 9. 任务 1 的 RDD 关系 DAG

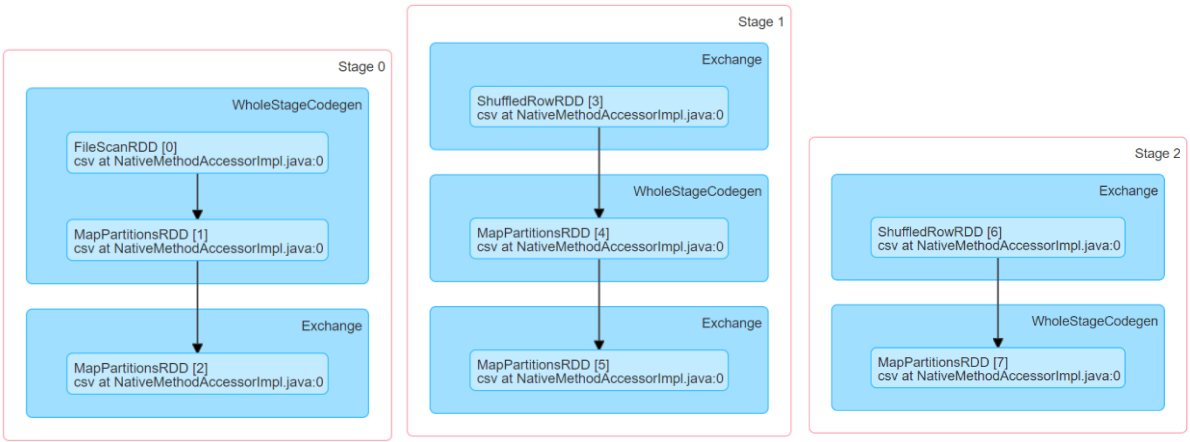


图 10. 任务 1 的 3 个 stage 里面具体 RDD 转换操作

2.2.2 运营商占比

任务 2 的计算思路是先根据通话类型和运营商分组聚合，得到不同运营商和通话类型下不同的用户数量。与之并行的另一个 RDD 只计算不同通话类型下的不重复用户个数。将两个查询结果 join 后 map 相当于第一个查询 RDD 多了一列。两列相除就得到不同通话类型下各个运营商的用户数占比。核心代码如下：

```
def qes_2(spark):
    print("Start computing question 2: ")
    qes2=spark.sql("""
    select
    ta.call_type,
    called_optr,
    user_num/user_num_2*100 as percent
    from (
        select call_type,called_optr, count (distinct called_nbr) as us
er_num
        from calling
        group by call_type,called_optr
    )as ta join
    (
        select call_type,count(distinct called_nbr) as user_num_2
        from calling
        group by call_type
    )as tb
    on (ta.call_type=tb.call_type)
    order by ta.call_type,called_optr
    """)
    qes2.write.csv("dc2019/qes_2.csv",mode='overwrite')
print("Question 2 completed")
```

结果饼状图如图 11 所示：

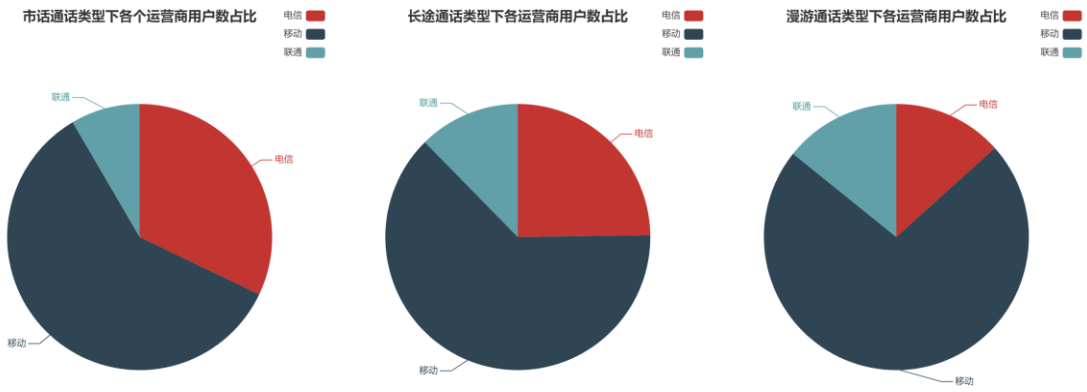


图 11. 不同通话类型下各个运营商的用户数占比

从图 12 中我们可以看到这个查询被 SQL 引擎分为了 5 个 stage，其中包括了 2 个可以并行的执行序列。第一子查询经过 group 和 join 操作（stage0，1）到达 stage5，第二个子查询经过 group，sort 和 join 操作同样到达 stage5. Stage5 中的 RDD 经过 map 操作和合并后再去除重复号码，这样得到最后的结果 RDD。

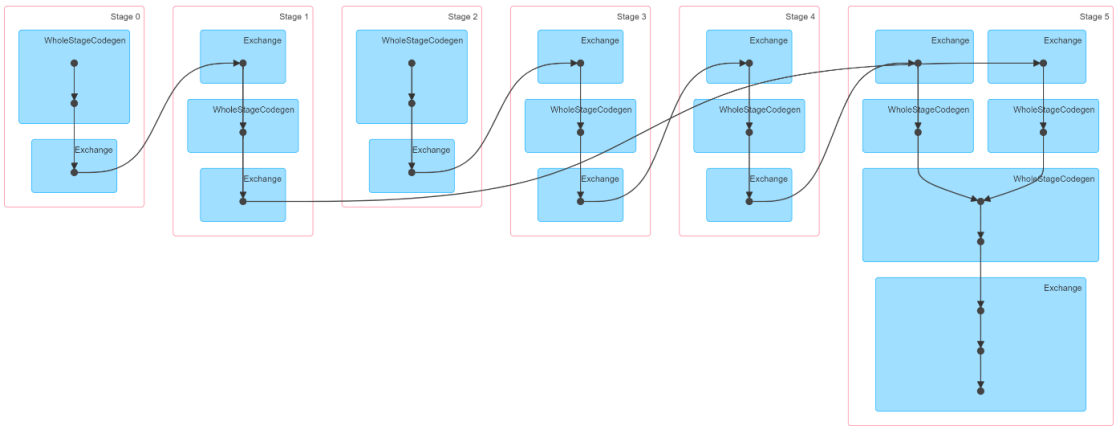


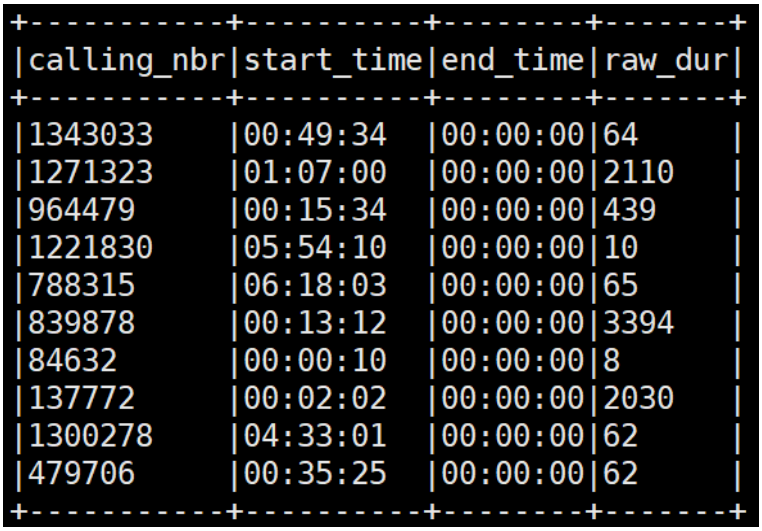
图 12. 任务 2 的 RDD 转换关系 DAG

2.2.3 通话时长占比

时间段名称	时间段的起止时间
时间段 1	0:00-3:00
时间段 2	3:00-6:00
时间段 3	6:00-9:00
时间段 4	9:00-12:00
时间段 5	12:00-15:00
时间段 6	15:00-18:00
时间段 7	18:00-21:00
时间段 8	21:00-24:00

图 13. 时间段划分

对于任务 3，我们定义了并注册了一个 UDF 函数：time*distribute*udf(time*id, start, end, raw*dur)。这个函数输入数据对应的通话时间段，开始结束时间和通话时长。该函数返回此次通话落在该通话时间段内的时间。如果通话在凌晨进行并跨天则算入第二天的对应时间段。这里我们发现原始的数据有不合法的数据，如下图所示，结束时间或者是开始的时间没有记录（00:00:00），这里我们根据通话时长重新处理了时间段的起止时间，并过滤了通话时长为 0 的数据。



calling_nbr	start_time	end_time	raw_dur
1343033	00:49:34	00:00:00	64
1271323	01:07:00	00:00:00	2110
964479	00:15:34	00:00:00	439
1221830	05:54:10	00:00:00	10
788315	06:18:03	00:00:00	65
839878	00:13:12	00:00:00	3394
84632	00:00:10	00:00:00	8
137772	00:02:02	00:00:00	2030
1300278	04:33:01	00:00:00	62
479706	00:35:25	00:00:00	62

图 14. 不合法样本示例

定义好这个函数后，就把源数据的每条记录进行 map 操作，每一条都计算 8 个时间段时间，然后根据主叫号码分组，聚合计算总共的通话时间和 8 个时间段占比。核心代码如下，结果如图 15 所示：

```
def qes_3(spark):
    print("Start computing question 3: ")
    qes3=spark.sql("""
    select
    calling_nbr,
    100*period_1/total_call as period_1,100*period_2/total_call as peri
od_2,
    100*period_3/total_call as period_3,100*period_4/total_call as peri
od_4,
    100*period_5/total_call as period_5,100*period_6/total_call as peri
od_6,
    100*period_7/total_call as period_7,100*period_8/total_call as peri
od_8
    from
    (
        select
        calling_nbr,
        sum(period_1) as period_1,sum(period_2) as period_2,
        sum(period_3) as period_3,sum(period_4) as period_4,
        sum(period_5) as period_5,sum(period_6) as period_6,
        sum(period_7) as period_7,sum(period_8) as period_8,
        sum(raw_dur) as total_call
        from
        (
            select calling_nbr,
            time_distribute(1,start_time,end_time,raw_dur) as period_1,
```

```
time_distribute(2,start_time,end_time,raw_dur) as period_2,
            time_distribute(3,start_time,end_time,raw_dur) as period_3,
time_distribute(4,start_time,end_time,raw_dur) as period_4,
            time_distribute(5,start_time,end_time,raw_dur) as period_5,
time_distribute(6,start_time,end_time,raw_dur) as period_6,
            time_distribute(7,start_time,end_time,raw_dur) as period_7,
time_distribute(8,start_time,end_time,raw_dur) as period_8,
            raw_dur
            from calling
            where raw_dur!=0
        )
        group by calling_nbr
    )
    """)
    qes3.write.csv("dc2019/qes_3.csv",mode='overwrite')
print("Question 3 completed")
```

主叫号码	每日平均通话次数	时间段1(%)	时间段2(%)	时间段3(%)	时间段4(%)	时间段5(%)	时间段6(%)	时间段7(%)	时间段8(%)
921356	5.9	0	0	0	0	100	0	0	0
638735	5.9	0	0	14.5	26.5	0	30.6	28.4	0
492140	4	0	0	0	0	0	100	0	0
46870	4	0	0	0	17.2	65.4	17.4	0	0
818989	4	0	0	19	0	0	34.2	46.8	0
145079	4	0	0	0	0	100	0	0	0
909595	4	0	0	0	0	100	0	0	0
1010766	4	0	0	0	54.3	0	45.7	0	0
403087	4	0	0	0	0	0	0	100	0
415766	4	0	0	0	13.9	74.4	10	1.6	0

图 15. 任务 1 和 3 结果——用户每天平均通话次数和不同时间段占比