大数据高端人才专项计划

东北大学 Northeastern University

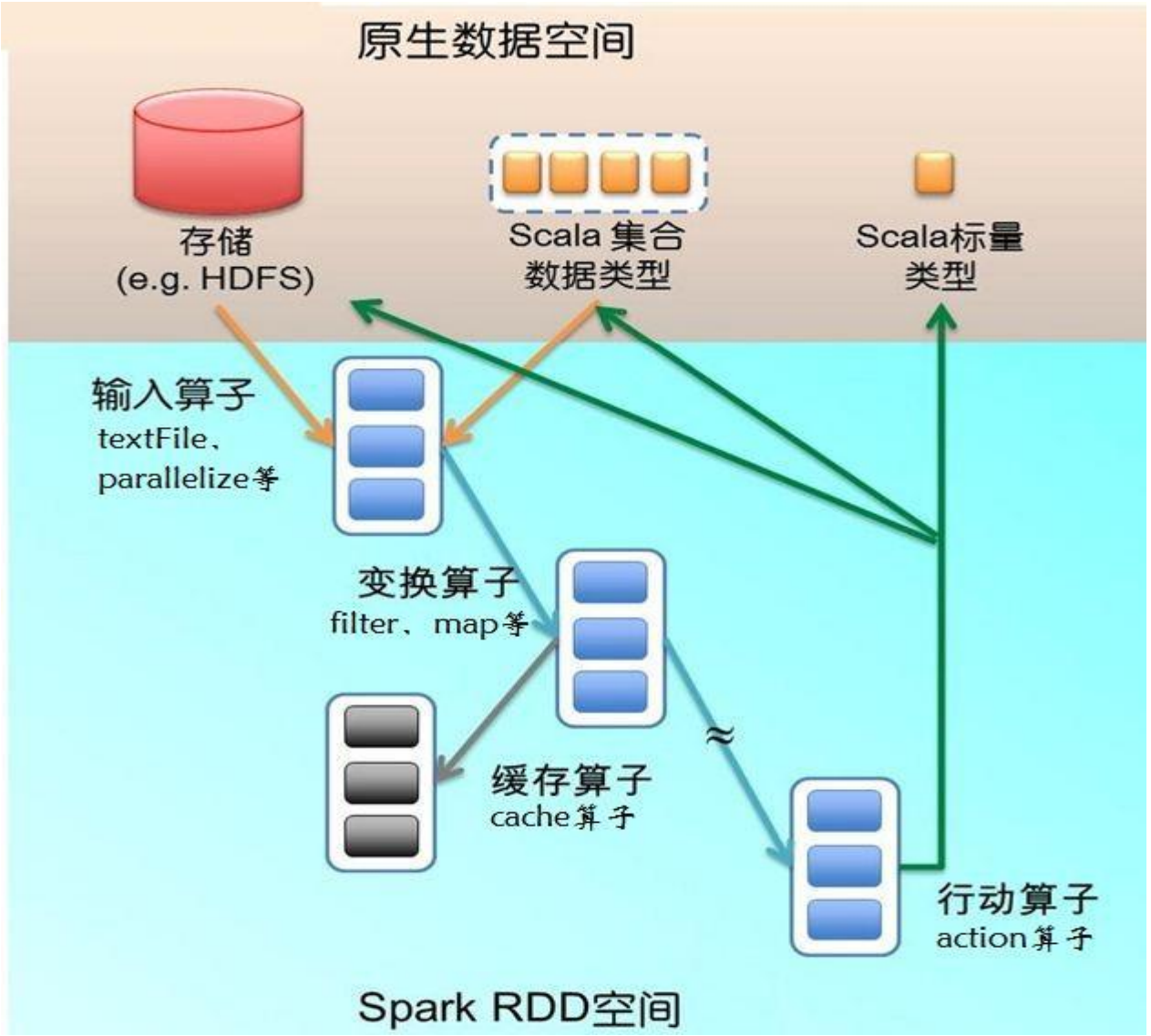
HORIZON 昊宸科技

Spark 编程模型

Spark 编程模型

- 右图给出了rdd 编程模型，并将下例中用到的四个算子映射到四种算子类型。spark程序工作在两个空间中：spark rdd空间和scala原生数据空间。在原生数据空间里，数据表现为标量（即scala基本类型，用橘色小方块表示）、集合类型（蓝色虚线 框）和持久存储（红色圆柱）。

```
val file = sc.textFile("hdfs://...")
val errors = file.filter(_.contains("ERROR"))
errors.cache()
errors.count()
```



Spark UI 解读

- 如何访问Spark UI, 如果是Local 模式，通过本地IP 地址 + 4040 端口可以访问到Spark UI

Spark 再启动其他任务的时候会监测4040 端口是否已经被占用掉，如果已经占用，spark会报一个端口占用的错误，然后通过启动其他端口比如说4041。

```
17/06/13 06:27:30 INFO SparkEnv: Registering BlockManagerMaster
17/06/13 06:27:30 INFO DiskBlockManager: Created local directory at C:\Users\Administrator.thinkpad-PC\AppData\Local\Temp\blockmgr-630d
17/06/13 06:27:30 INFO MemoryStore: MemoryStore started with capacity 1771.1 MB
17/06/13 06:27:30 INFO SparkEnv: Registering OutputCommitCoordinator
17/06/13 06:27:30 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
17/06/13 06:27:30 INFO Utils: Successfully started service 'SparkUI' on port 4041.
17/06/13 06:27:30 INFO SparkUI: Started SparkUI at http://192.168.1.100:4041
17/06/13 06:27:31 INFO Executor: Starting executor ID driver on host localhost
17/06/13 06:27:31 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 2489.
17/06/13 06:27:31 INFO NettyBlockTransferService: Server created on 2489
17/06/13 06:27:31 INFO BlockManagerMaster: Trying to register BlockManager
17/06/13 06:27:31 INFO BlockManagerMasterEndpoint: Registering block manager localhost:2489 with 1771.1 MB RAM, BlockManagerId(driver,
17/06/13 06:27:31 INFO BlockManagerMaster: Registered BlockManager
```

Spark UI 解读

- Jobs：在里面可以看到当前应用分析出来的所有任务，以及所有的excutors中action的执行时间
- Stages：在里面可以看到应用的所有stage，stage是按照宽依赖来区分的，因此粒度上要比job更细一些
- Storage：我们所做的cache persist等操作，都会在这里看到，可以看出来应用目前使用了多少缓存
- Environment：里面展示了当前spark所依赖的环境，比如jdk,lib等等
- Excutors：这里可以看到执行者申请使用的内存以及shuffle中input和output等数据
- 应用的名字可以在右上角看到

在Spark中job是根据action操作来区分的，另外任务还有一个级别是stage，它是根据宽窄依赖来区分的。

Spark UI 解读-Jobs

- Jobs：的数量和 action 算子的数量有关系

Spark 1.6.3 | Jobs | Stages | Storage | Environment | Executors | SparkUIDemo application UI

Spark Jobs (?)

Total Uptime: 18 s
Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	count at SparkUIDemo.scala:62	2017/06/13 09:39:11	16 ms	1/1 (3 skipped)	1/1 (3 skipped)
0	collect at SparkUIDemo.scala:60	2017/06/13 09:39:09	2 s	4/4	4/4

```
//第一个action 算子
userDistribution.collect.foreach(println)
//第一个action 算子
userDistribution.count
```

Spark UI 解读-Jobs-调度方式

- FIFO: Spark 调度器在默认情况下以FIFO(先进先出)方式调度Job执行，第一个Job优先获取可用资源，第二个Job还可以继续获取剩余资源，如果第一个Job没有获取全部资源，则第二个Job可以并行运行。
- FAIR: Spark 可以通过配置FAIR共享调度模式调度Job，Spark 在多任务之间以轮训方式为任务分配资源。
- *自定义调度池: 用户可以通过配置文件自定义调度池。

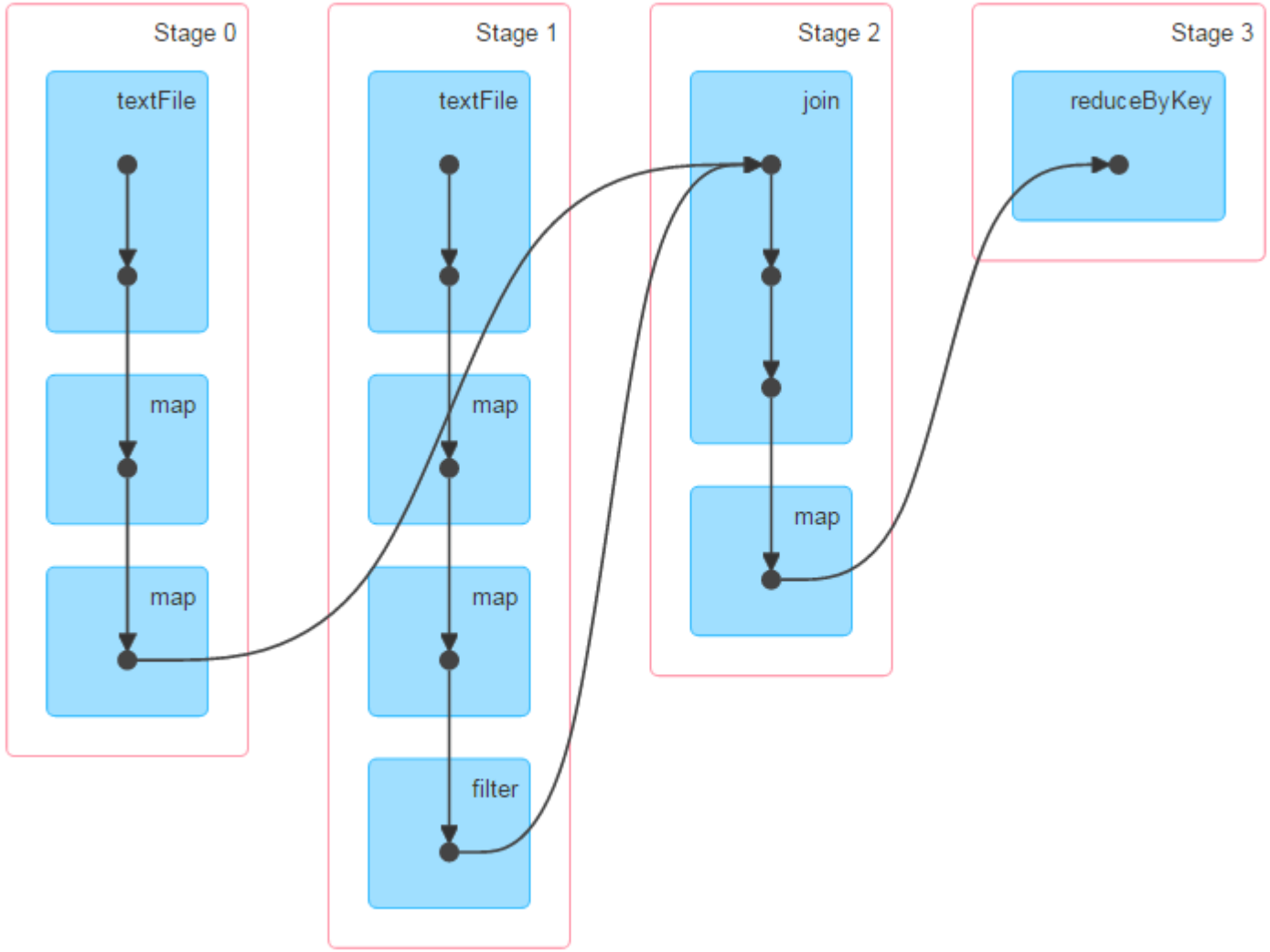
1) 调度模式（SchedulingMode）:用户可以选择FIFO或者FAIR方式进行调度。

2)权重(Weight): 这个参数控制在整个集群资源的分配上，这个调度池相对于其他调度池优先级的高低。

3)minShare： 配置minShare参数（这个参数代表多少CPU核）这个参数决定整体调度的调度池能给待调度的调度池分配多少资源就可以满足调度池的资源需求。

Spark UI 解读-Stages

- stage：是根据宽窄依赖来区分的



Spark UI 解读-Stages

- stage：是根据宽窄依赖来区分的

Completed Stages (4)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	collect at ShowStagesDemo.scala:61	+details	2017/06/14 18:56:28	36 ms	3/3			2033.0 B	
2	map at ShowStagesDemo.scala:57	+details	2017/06/14 18:56:28	0.5 s	3/3			77.2 KB	2033.0 B
1	filter at ShowStagesDemo.scala:46	+details	2017/06/14 18:56:26	1 s	1/1	23.5 MB			2.6 KB
0	map at ShowStagesDemo.scala:36	+details	2017/06/14 18:56:26	0.6 s	3/3	262.4 KB			74.6 KB

Spark UI 解读-Stages

- Spark task 类型（Spark shuffle 和 MapReduce 类似，只是做了一些优化）

  ShuffleMapTask（Map task）

  ResultTask （Result Task）

- 每个Stage的task 数目（task 数目怎么决定的）

  1)第一个Stage 由存储介质决定HDFS block 或者 hbase region数目决定

  比如要读取1GB 数据，HDFS block 默认大小为128M，那么spark 会启动 8个任务来读取这个集合

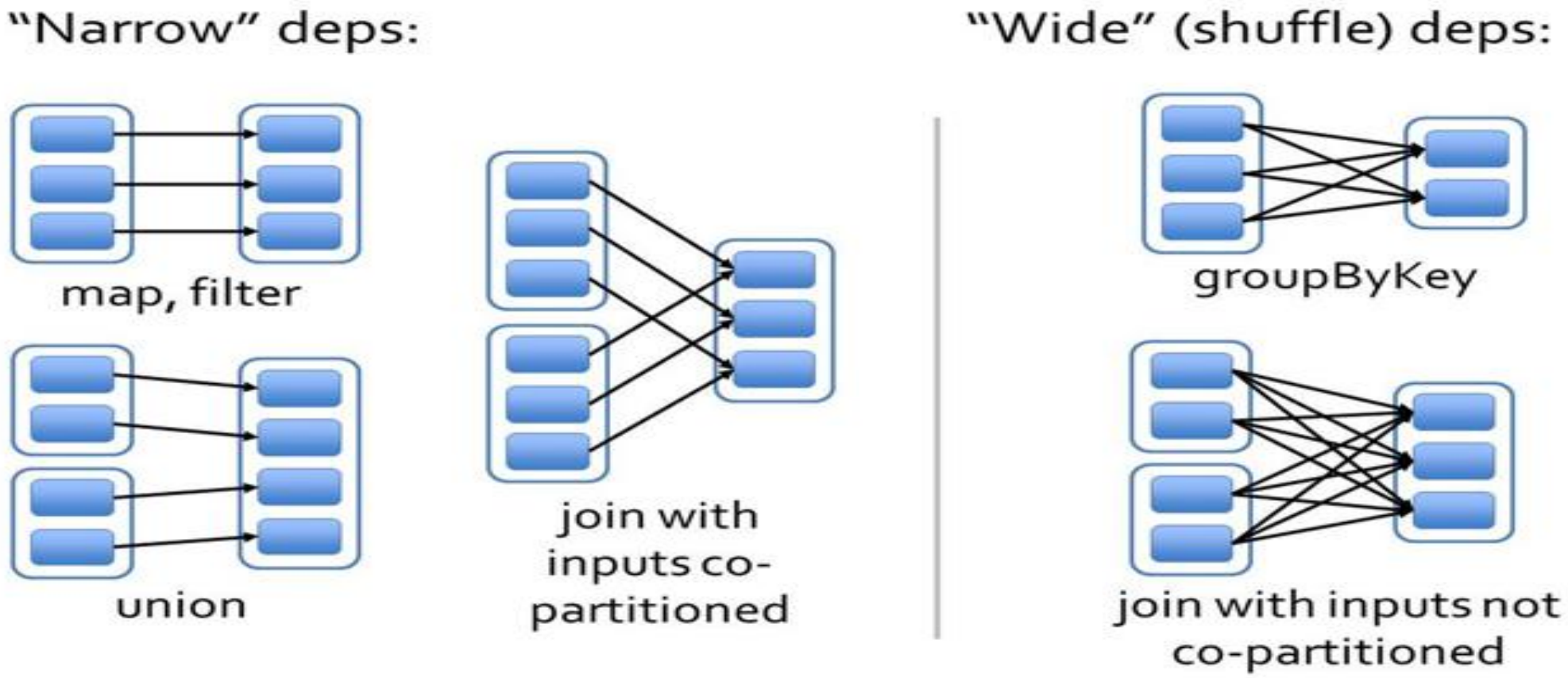
  2) 由用户自己设置，默认等于第一阶段的并行度

宽窄依赖

- Spark中RDD的高效与DAG图有着莫大的关系，在DAG调度中需要对计算过程划分stage，而划分依据就是RDD之间的依赖关系。针对不同的转换函数，RDD之间的依赖关系分为窄依赖（narrow dependency）和宽依赖（wide dependency, 也称 shuffle dependency）.
- 窄依赖是指父RDD的每个分区只被子RDD的一个分区所使用，子RDD分区通常对应常数个父RDD分区(O(1)，与数据规模无关)
- 宽依赖是指父RDD的每个分区都可能被多个子RDD分区所使用，子RDD分区通常对应所有的父RDD分区(O(n)，与数据规模有关)

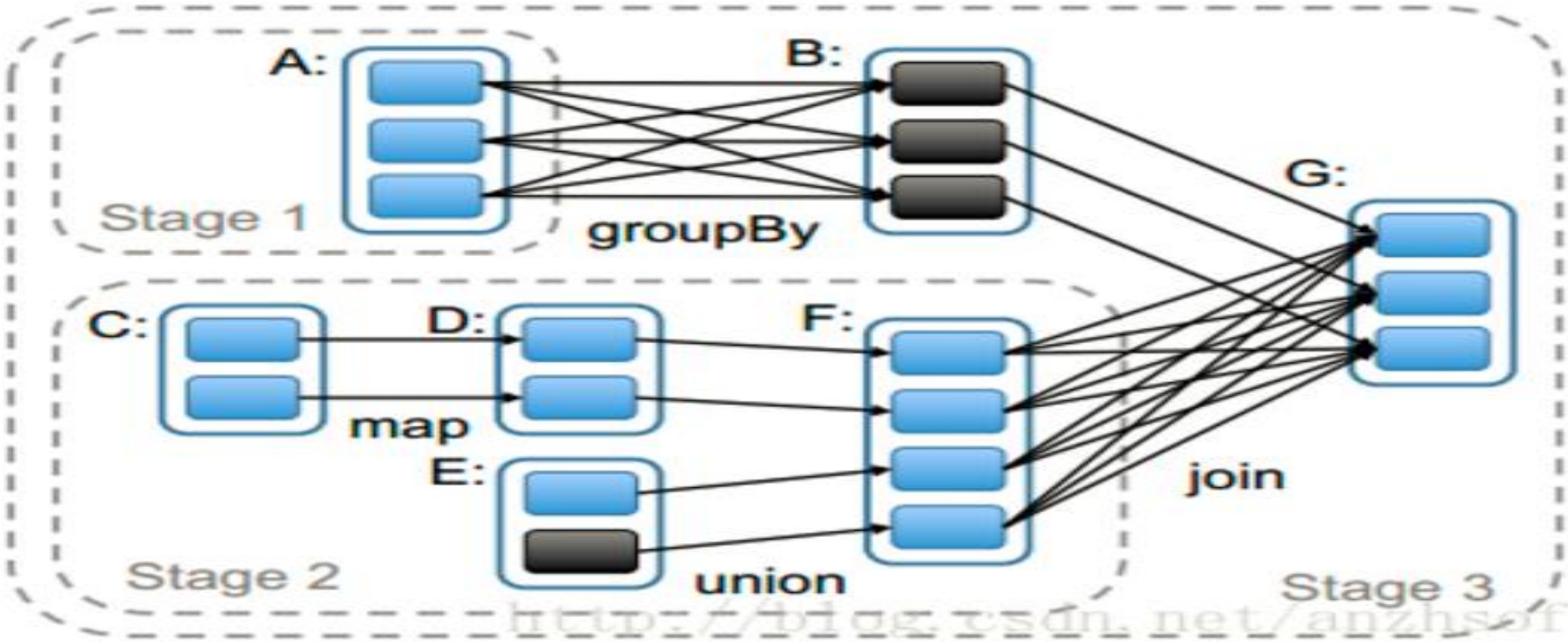
宽窄依赖

- 针对不同的转换函数，RDD之间的依赖关系分类窄依赖（ narrow dependency ）和宽依赖（ wide dependency, 也称 shuffle dependency ）.



Stage划分

- C D E F 都是窄依赖，划分为一个Stage，A和 B 是宽依赖，所以需要shuffle，A是一个Stage；从后往前看从G发启。



Spark UI 解读-Storage

- storage页面能看出目前使用的缓存，点击进去可以看到具体在每个机器上，使用的block的情况

Spark 1.6.3 Jobs Stages Storage Environment Executors **ShowStagesDemo** application UI

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
G:\spark\workspaces\SparkCoreOpt\resource\data\ml-1m\users.dat	Memory Deserialized 1x Replicated	3	100%	1056.9 KB	0.0 B	0.0 B

```
val usersRdd = sc.textFile(DATA_PATH + "/users.dat",3)
usersRdd.cache()
```

Spark UI 解读-Executors

- Executors这个页面比较常用了，一方面通过它可以看出来每个excutor是否发生了数据倾斜，另一方面可以具体分析目前的应用是否产生了大量的shuffle，是否可以通过数据的本地性或者减小数据的传输来减少shuffle。

Executors (5)

Memory: 173.6 KB Used (3.2 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
1	idh102:46715	2	57.9 KB / 511.5 MB	0.0 B	0	0	5	5	3.6 s	12.0 MB	9.2 KB	51.6 KB	stdout stderr	Thread Dump
2	idh103:55613	1	28.9 KB / 511.5 MB	0.0 B	0	0	1	1	2.9 s	11.8 MB	0.0 B	1347.0 B	stdout stderr	Thread Dump
3	idh101:40014	1	28.9 KB / 511.5 MB	0.0 B	0	0	3	3	2.8 s	151.5 KB	17.9 KB	25.7 KB	stdout stderr	Thread Dump
4	idh105:44575	0	0.0 B / 511.5 MB	0.0 B	0	0	2	2	1.6 s	0.0 B	26.3 KB	669.0 B	stdout stderr	Thread Dump
driver	192.168.88.104:56265	2	57.9 KB / 1247.6 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

```
sh /usr/hdp/2.5.0.0-1245/spark/bin/spark-submit --class com.horizon.opt.stages.ShowStagesDemo --master yarn-client --driver-memory 2g --executor-memory 1g --num-executors 4 --executor-cores 1 ./sparkcoreopt_2.10-1.0.jar hdfs://192.168.88.104:8020/out/ml-1m/
```

Spark UI 解读-Executors

Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebo
	9	6	12 GB	176 GB	0 B	6	100	0	4	0	0	0	0

Scheduling Resource Type	Minimum Allocation	Maximum Allocation
[MEMORY, CPU]	<memory:2048, vCores:1>	<memory:45056, vCores:25>

Search:

ser	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	% of Queue	% of Cluster	Progress	Tracking UI
ark	ShowStagesDemo	SPARK	default	0	Wed Jun 14 19:15:12 +0800	N/A	RUNNING	UNDEFINED	5	5	10240	11.4	5.7		ApplicationMaster

yarn-client

--driver-memory 2g

--executor-memory 1g

--num-executors 4 --executor-cores 1 ./sparkcoreopt_2.10-1.0.jar hdfs://192.168.88.104:8020/out/ml-1m/

Spark概念总结

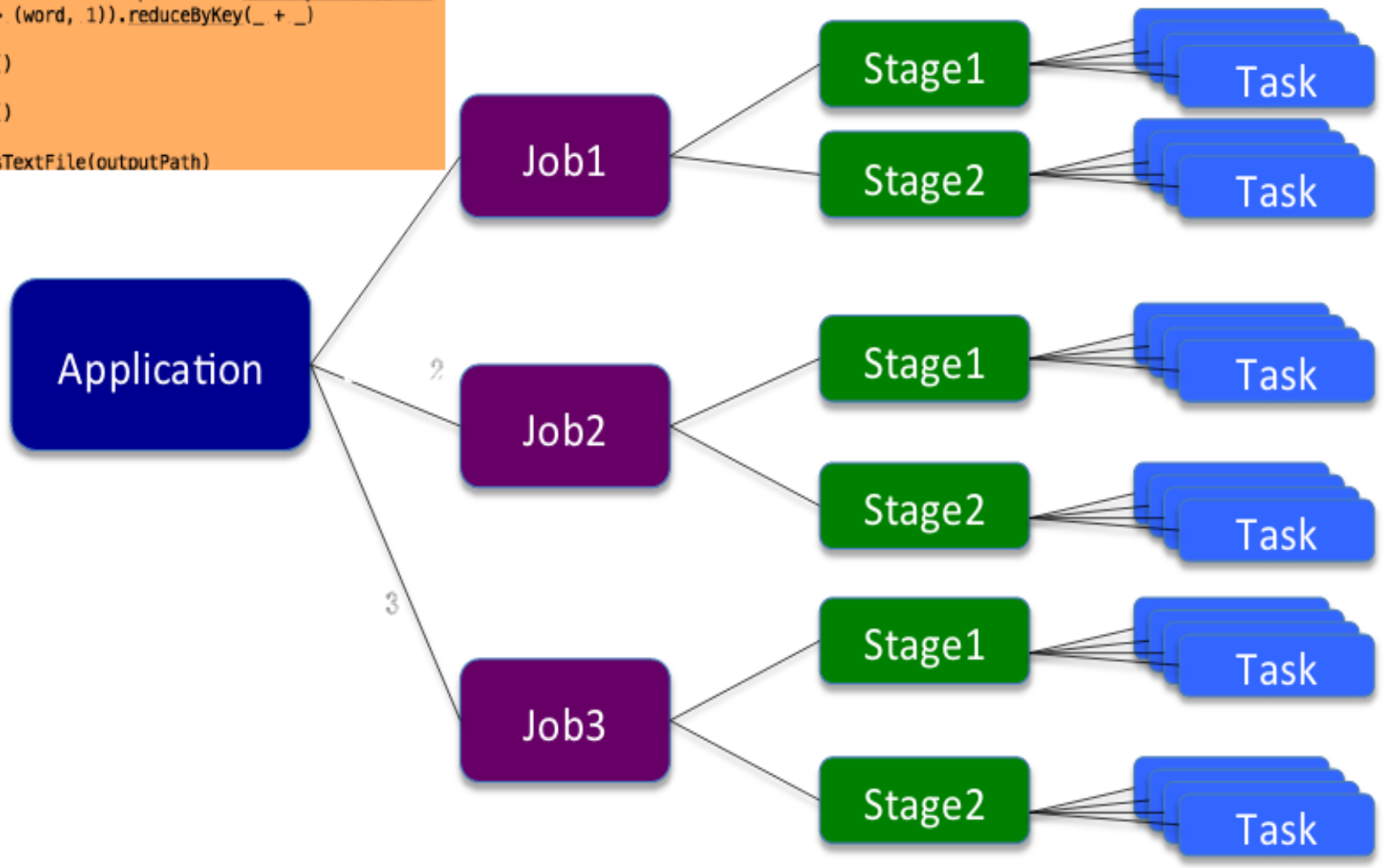
```
val rowRdd = sc.textFile(inputPath)
val resultRdd = rowRdd.flatMap(line => line.split("\\s+"))
    .map(word => (word, 1)).reduceByKey(_ + _)

resultRdd.count()

resultRdd.first()

resultRdd.saveAsTextFile(outputPath)
```

- Application：每一个应用，我们写的Scala object
- Job：有多少个action，调度器并行执行
- Stage：每个job 被切分为多个阶段
- Task：每个阶段都是并发启动多个task执行的




Spark 算子

Spark 算子

- Transformation 变换/转换算子：这种变换并不触发提交作业，完成作业中间过程处理

Transformation 操作是延迟计算的，也就是说从一个RDD 转换生成另一个 RDD 的转换操作不是马上执行，需要等到有 Action 操作的时候才会真正触发运算。

- Action 行动算子

Action 算子会触发 Spark 提交作业（Job），并将数据输出 Spark系统。

为了更好的掌握这些Spark 算子我们将Spark 算子分分类

①Value数据类型的Transformation算子，这种变换并不触发提交作业，针对处理的数据项是Value型的数据。

②Key-Value数据类型的 Transfromation 算子，这种变换并不触发提交作业，针对处理的数据项是Key-Value型的数据对。

③Action算子，这类算子会触发 SparkContext 提交Job作业。

Value数据类型的Transformation算子

- 1.输入分区与输出分区一对一型

  map算子
  flatMap 算子
  mapPartitions 算子
  glom算子

- 2.输入分区与输出分区多对一型

  union 算子
  cartesian 算子

- 3.输入分区与输出分区多对多型

  grouBy算子

Value数据类型的Transformation算子

- 4.输出分区为输入分区子集型

  filter 算子
  distinct 算子
  subtract 算子
  sample 算子
  takeSample 算子

- 5.Cache型

  cache算子
  persist算子

Key-Value数据类型的Transfromation算子

- 1.输入分区与输出分区一对一

  mapValues算子

- 2.对单个RDD或两个RDD聚集单个RDD聚集

  combineByKey 算子
  reduceByKey 算子
  partitionBy 算子

- 3.两个RDD聚集

  Cogroup算子

- 4.连接

  join算子
  leftOutJoin 和 rightOutJoin 算子

Action 算子

- 1.无输出

  foreach算子

- 2. HDFS

  saveAsTextFile 算子
  saveAsObjectFile 算子

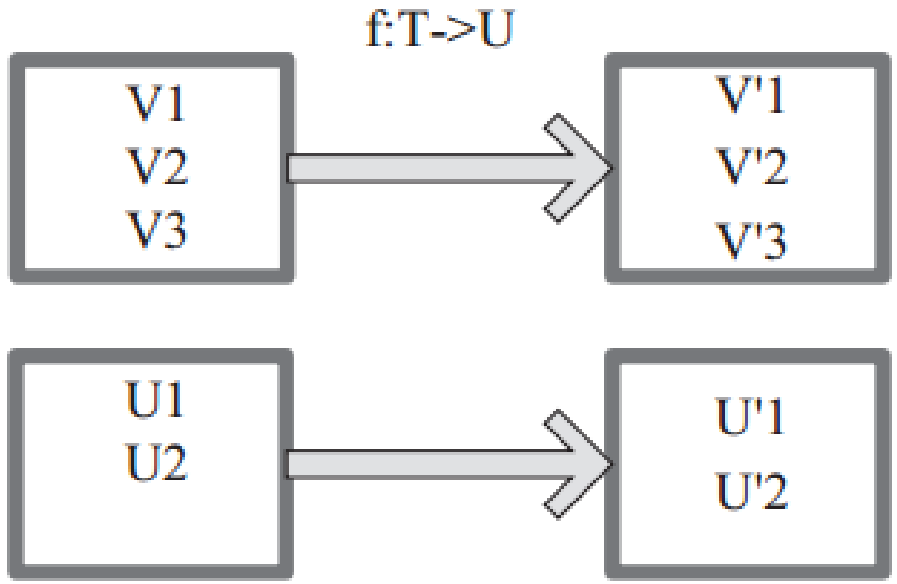
- 3. Scala 集合和数据类型

  collect算子
  collectAsMap 算子
  reduceByKeyLocally 算子
  lookup算子
  count算子
  top算子
  reduce算子
  fold算子
  aggregate算子

Map 算子

- 将原来 RDD 的每个数据项通过 map 中的用户自定义函数 f 映射转变为一个新的元素。

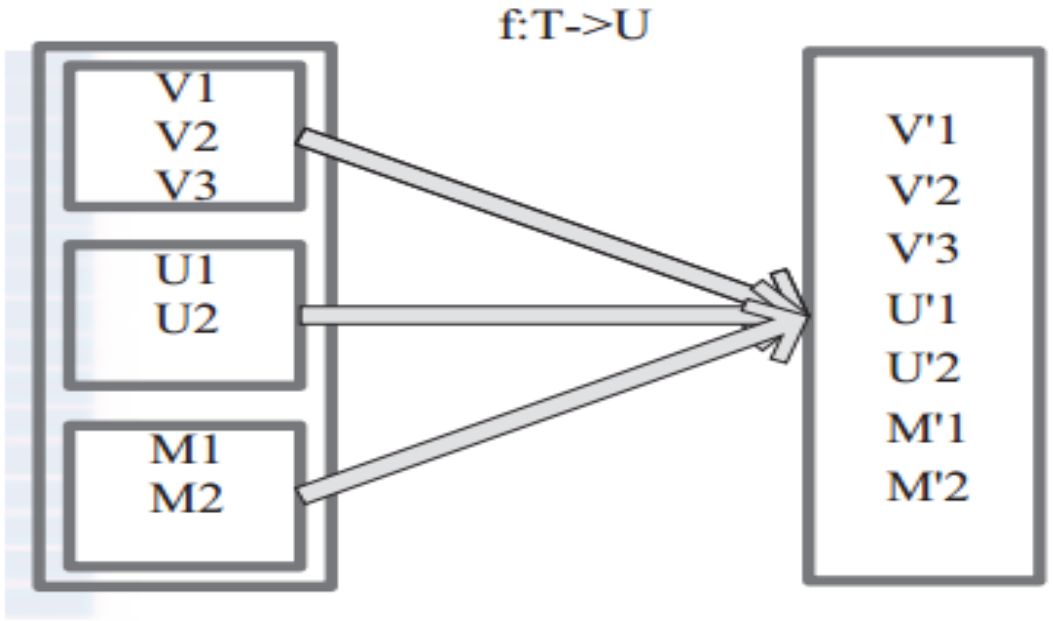
图中每个方框表示一个 RDD 分区，左侧的分区经过用户自定义函数 f:T->U 映射为右侧的新 RDD 分区。但是，实际只有等到 Action算子触发后，这个 f 函数才会和其他函数在一个stage 中对数据进行运算。在图 1 中的第一个分区，数据记录 V1 输入 f，通过 f 转换输出为转换后的分区中的数据记录 V'1。



FlatMap 算子

- 将原来 RDD 中的每个元素通过函数 f 转换为新的元素，并将生成的 RDD 的每个集合中的元素合并为一个集合。

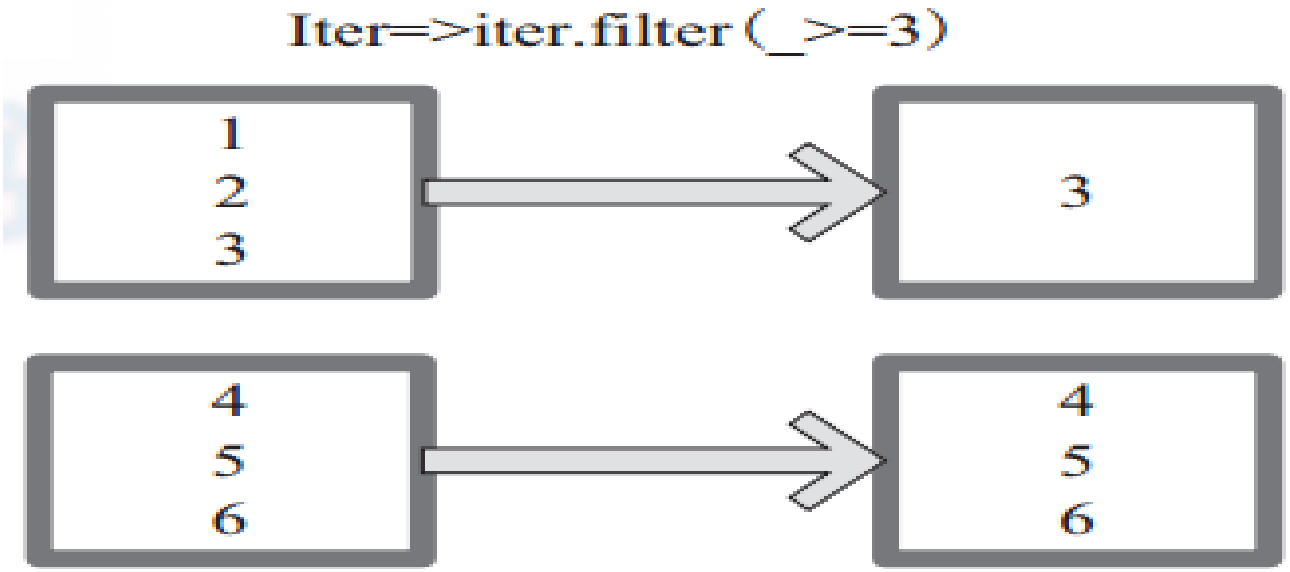
RDD 的一个分区，进行 flatMap函数操作， flatMap 中传入的函数为 f:T->U， T和 U 可以是任意的数据类型。将分区中的数据通过用户自定义函数 f 转换为新的数据。外部大方框可以认为是一个 RDD 分区，小方框代表一个集合。 V1、V2、 V3 在一个集合作为 RDD 的一个数据项，可能存储为数组或其他容器，转换为V'1、 V'2、 V'3 后，将原来的数组或容器结合拆散，拆散的数据形成为 RDD 中的数据项



MapPartitions 算子

- mapPartitions 函数获取到每个分区的迭代器，在函数中通过这个分区整体的迭代器对整个分区的元素进行操作。

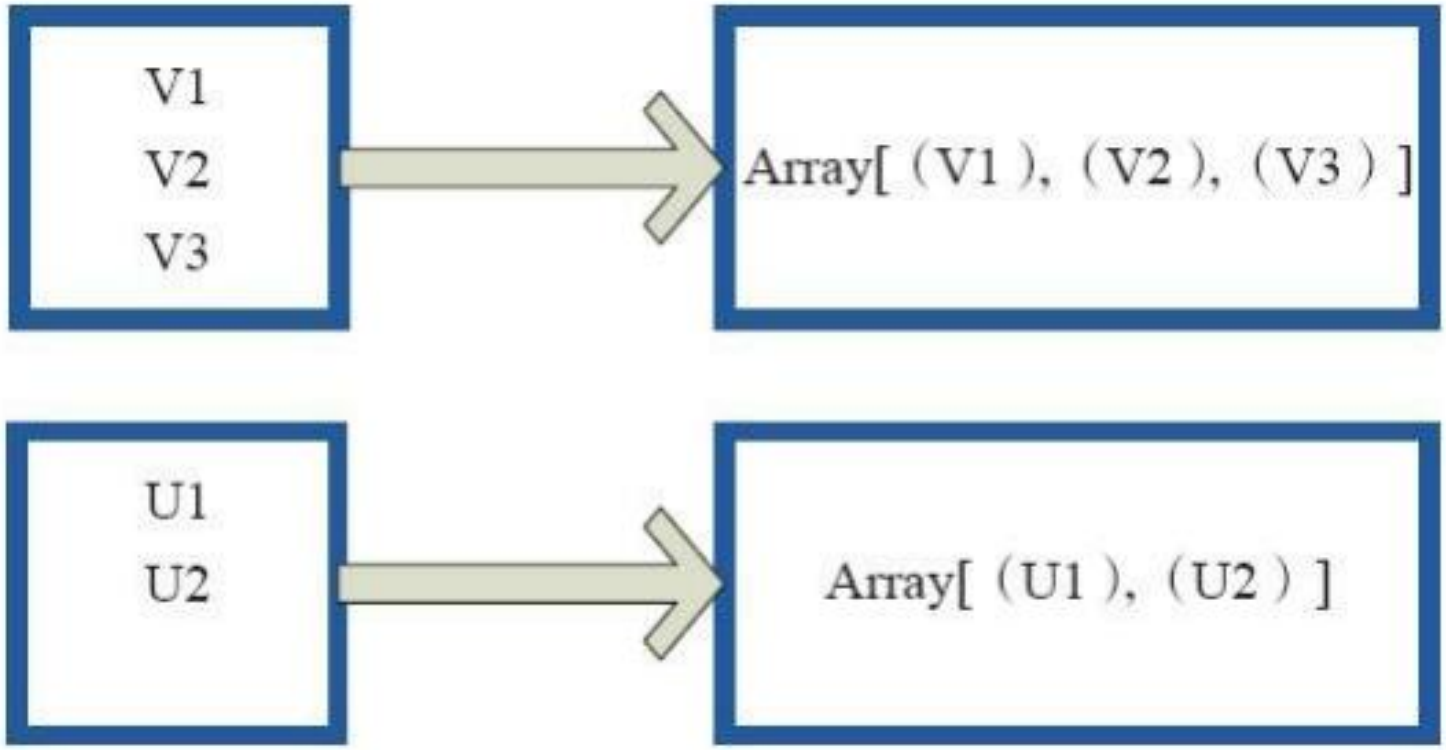
该函数和map函数类似，只不过映射函数的参数由RDD中的每一个元素变成了RDD中每一个分区的迭代器。如果在映射的过程中需要频繁创建额外的对象，使用mapPartitions要比map高效的多。比如，将RDD中的所有数据通过JDBC连接写入数据库，如果使用map函数，可能要为每一个元素都创建一个 connection，这样开销很大，如果使用mapPartitions，那么只需要针对每一个分区建立一个connection。参数preservesPartitioning表示是否保留父RDD的partitioner分区信息。



Glom 算子

- glom函数将每个分区形成一个数组。

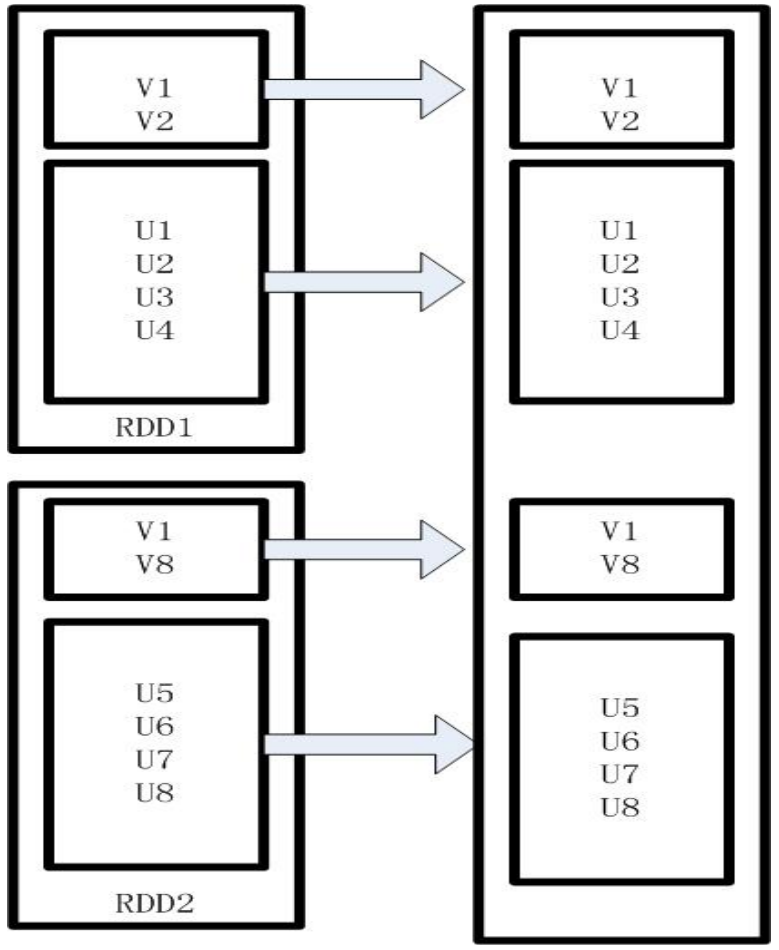
每个方框代表一个RDD分区。图中的方框代表一个分区。 该图表示含有V1、 V2、 V3的分区通过函数glom形成一数组Array[（V1），（V2），（V3）]。



Union 算子

- 使用 union 函数时需要保证两个 RDD 元素的数据类型相同，返回的 RDD 数据类型和被合并的 RDD 元素数据类型相同，并不进行去重操作，保存所有元素。

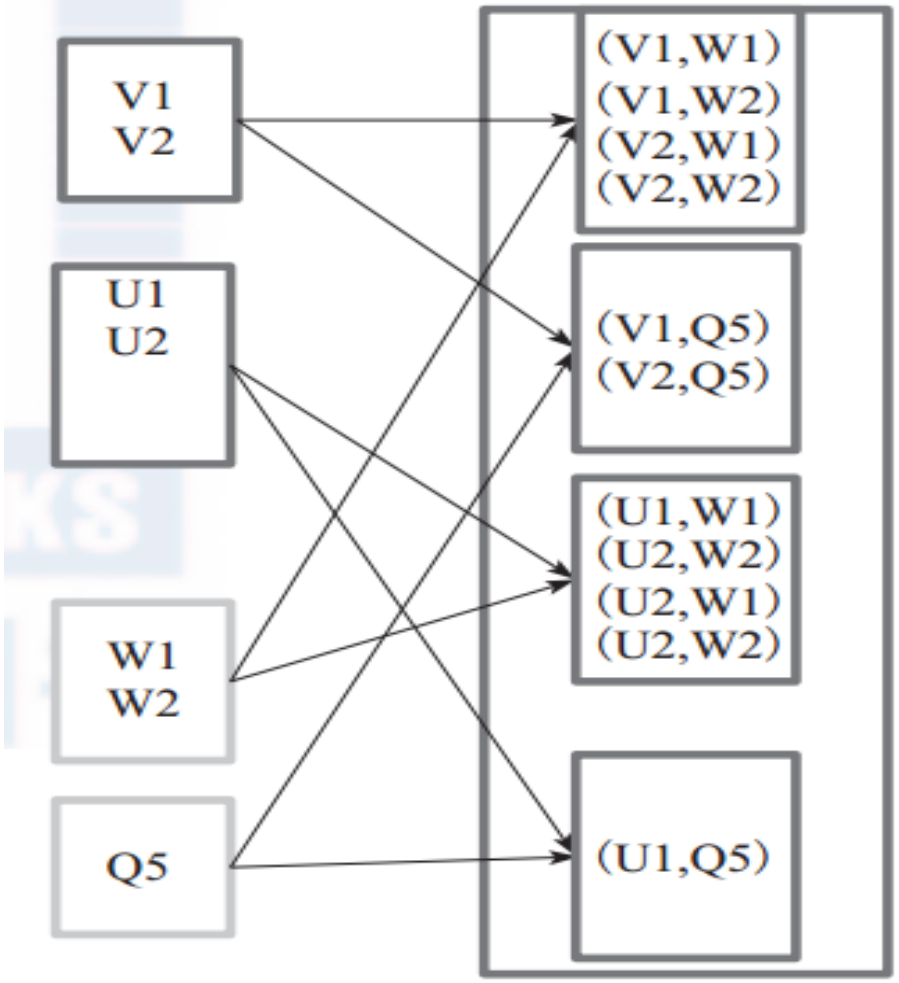
图中左侧大方框代表两个 RDD，大方框内的小方框代表 RDD 的分区。右侧大方框代表合并后的 RDD，大方框内的小方框代表分区。



Cartesian 算子

- 对两个RDD内的所有元素进行笛卡尔积操作。

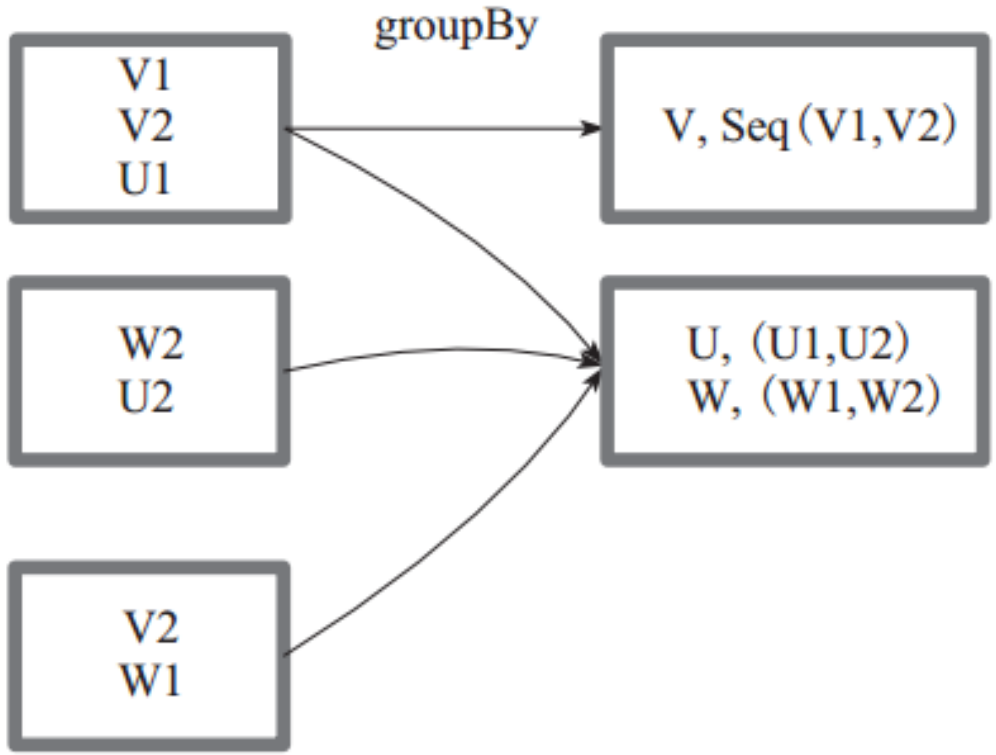
图中左侧大方框代表两个 RDD，大方框内的小方框代表 RDD 的分区。右侧大方框代表合并后的RDD，大方框内的小方框代表分区。图6中的大方框代表RDD，大方框中的小方框代表RDD分区。例如： V1 和另一个 RDD 中的 W1、 W2、Q5 进行笛卡尔积运算形成 (V1,W1)、(V1,W2)、 (V1,Q5) 。



GroupBy 算子

- 将元素通过函数生成相应的 Key，数据就转化为 Key-Value 格式，之后将 Key 相同的元素分为一组。

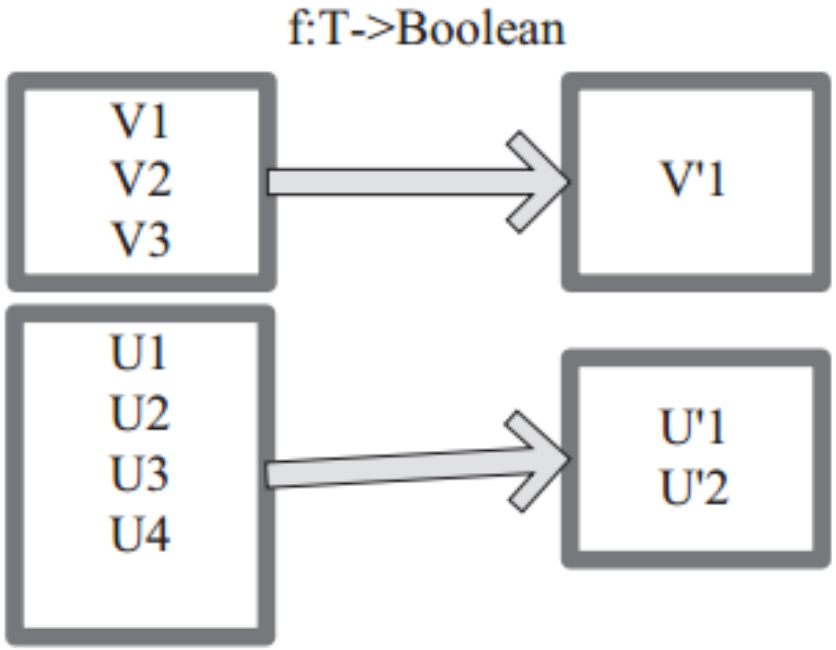
图中方框代表一个 RDD 分区，相同key 的元素合并到一个组。例如 V1 和 V2 合并为 V， Value 为 V1,V2。形成 V,Seq(V1,V2) 。



filter 算子

- filter 函数功能是对元素进行过滤，对每个元素应用 f 函数，返回值为 true 的元素在 RDD 中保留，返回值为 false 的元素将被过滤掉。

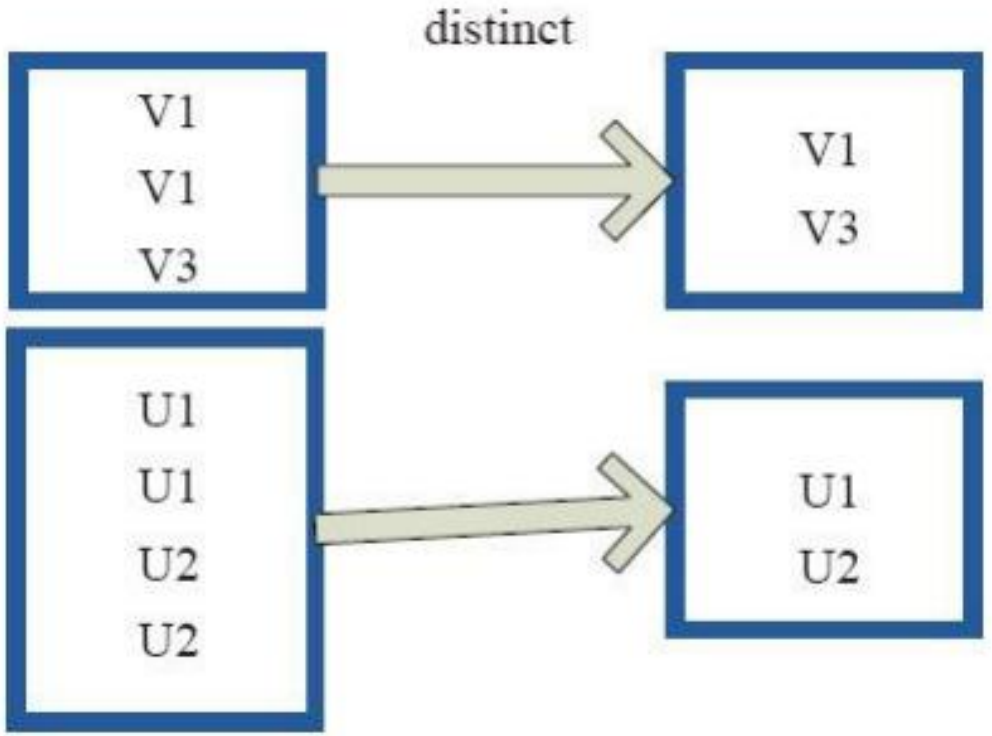
中每个方框代表一个 RDD 分区， T 可以是任意的类型。通过用户自定义的过滤函数 f，对每个数据项操作，将满足条件、返回结果为 true 的数据项保留。例如，过滤掉 V2 和 V3 保留了 V1，为区分命名为 V'1。



Distinct 算子

- distinct将RDD中的元素进行去重操作。

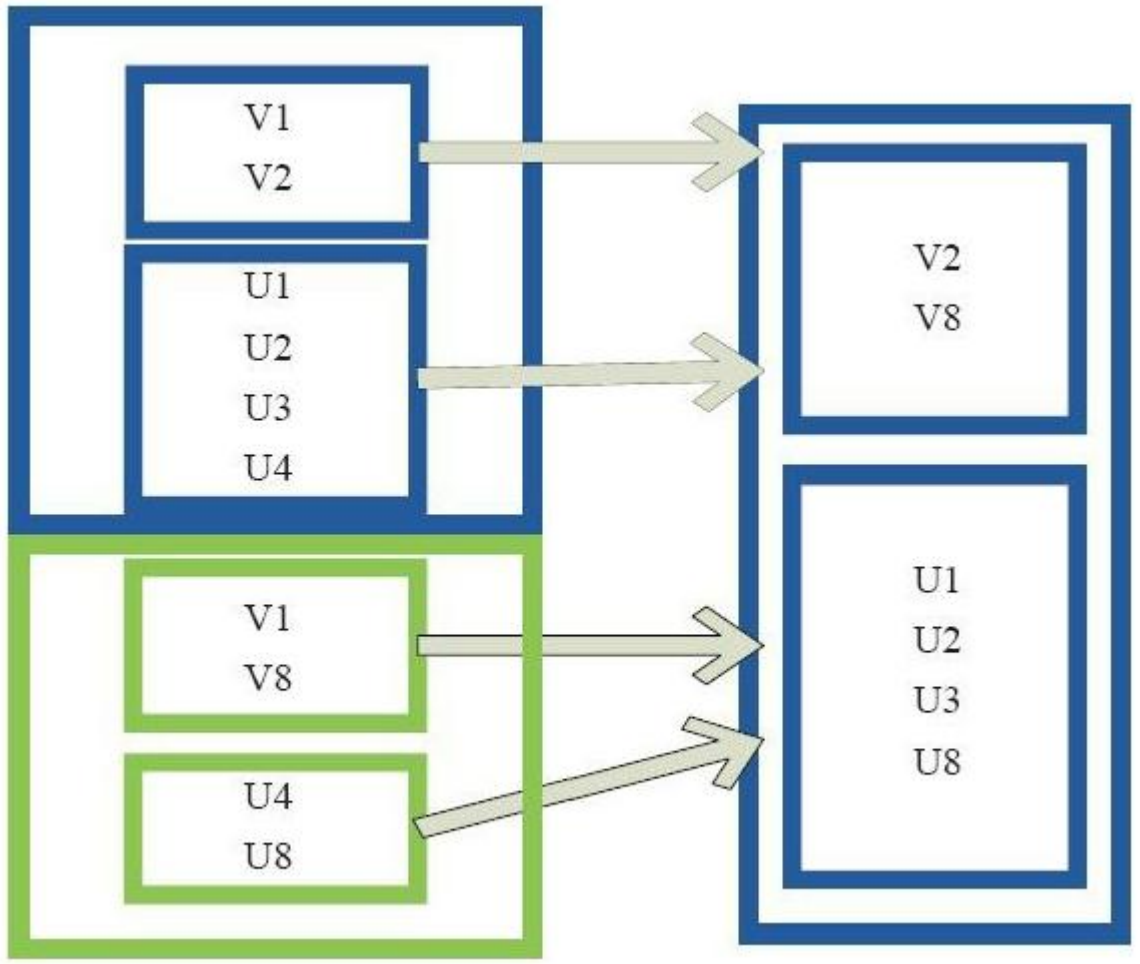
图中的每个方框代表一个RDD分区，通过distinct函数，将数据去重。 例如，重复数据V1、 V1去重后只保留一份V1 。



Subtract 算子

- Subtract相当于进行集合的差操作。

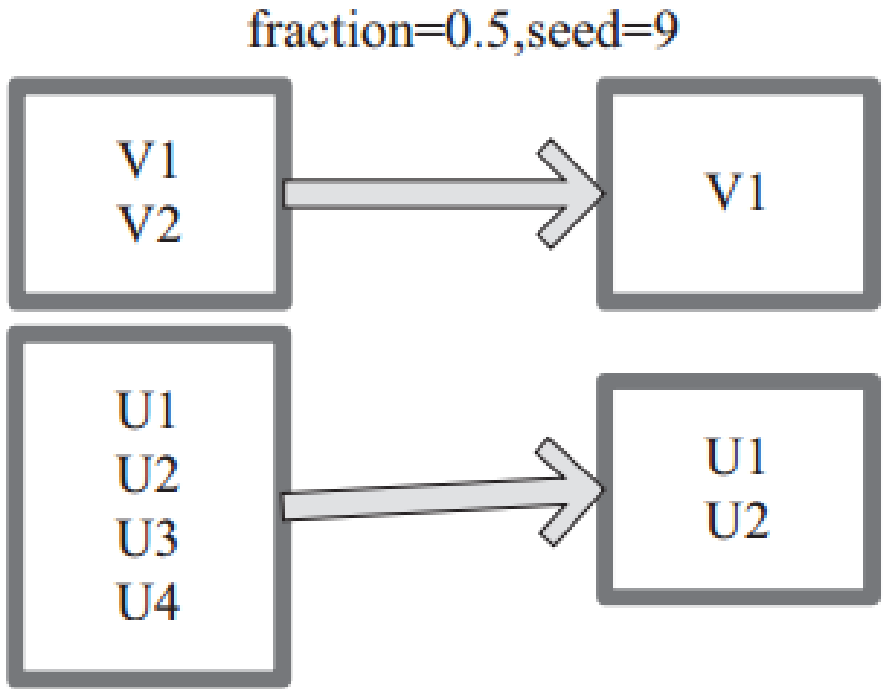
RDD 1去除RDD 1和RDD 2交集中的所有元素。图中左侧的大方框代表两个RDD，大方框内的小方框代表RDD的分区。右侧大方框代表合并后的RDD，大方框内的小方框代表分区。V1在两个RDD中均有，根据差集运算规则，新RDD不保留，V2在第一个RDD有，第二个RDD没有，则在新RDD元素中包含V2。



Sample 算子

- 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。

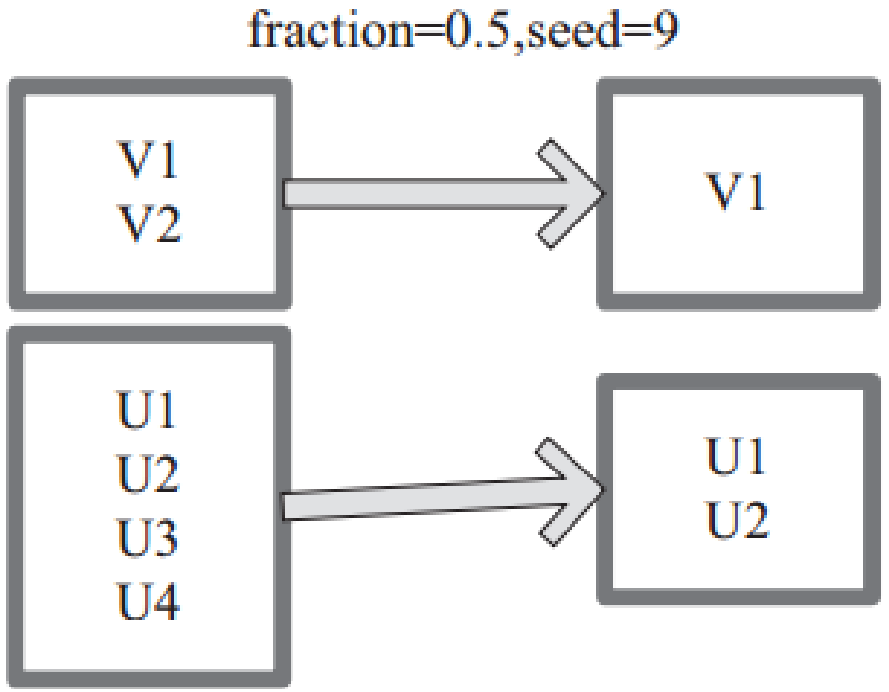
图中的每个方框是一个 RDD 分区。通过 sample 函数，采样 50% 的数据。V1、V2、U1、U2、U3、U4 采样出数据 V1 和 U1、U2 形成新的 RDD



TakeSample 算子

- takeSample（ ）函数和上面的sample函数是一个原理，但是不使用相对比例采样，而是按设定的采样个数进行采样，同时返回结果不再是RDD，而是相当于对采样后的数据进行Collect（ ），返回结果的集合为单机的数组。

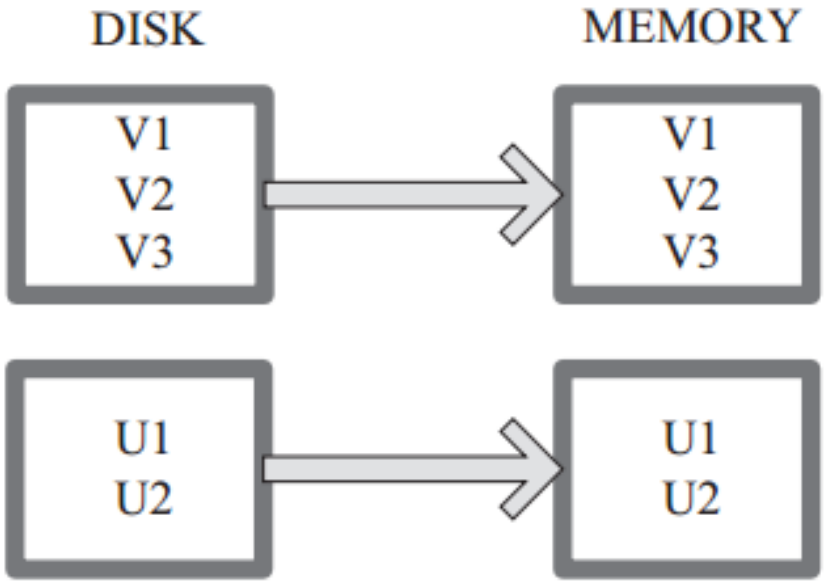
图中的每个方框是一个 RDD 分区。通过 sample 函数，采样 50% 的数据。V1、V2、U1、U2、U3、U4 采样出数据 V1 和 U1、U2 形成新的 RDD



Cache 算子

- Cache 将 RDD 元素从磁盘缓存到内存。 相当于 persist(MEMORY_ONLY) 函数的功能。

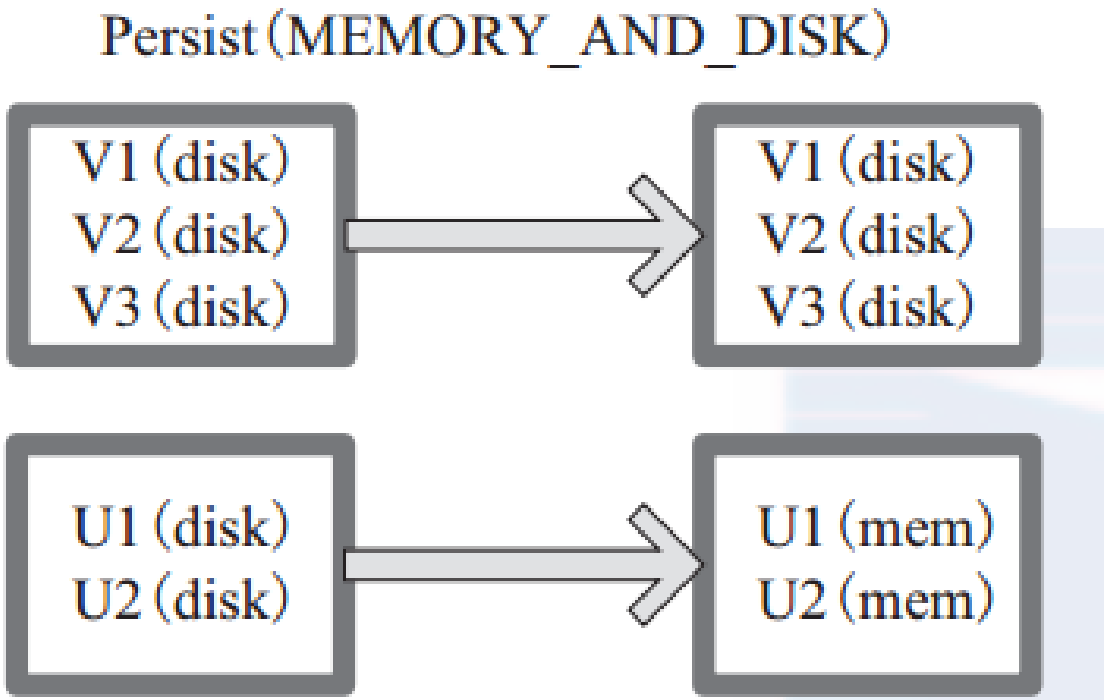
中每个方框代表一个 RDD 分区，左侧相当于数据分区都存储在磁盘，通过 cache 算子将数据缓存在内存。



Persis 算子

- Persist 函数对 RDD 进行缓存操作。数据缓存在哪里依据 StorageLevel 这个枚举类型进行确定。有以下几种类型的组，DISK 代表磁盘，MEMORY 代表内存， SER 代表数据是否进行序列化存储。

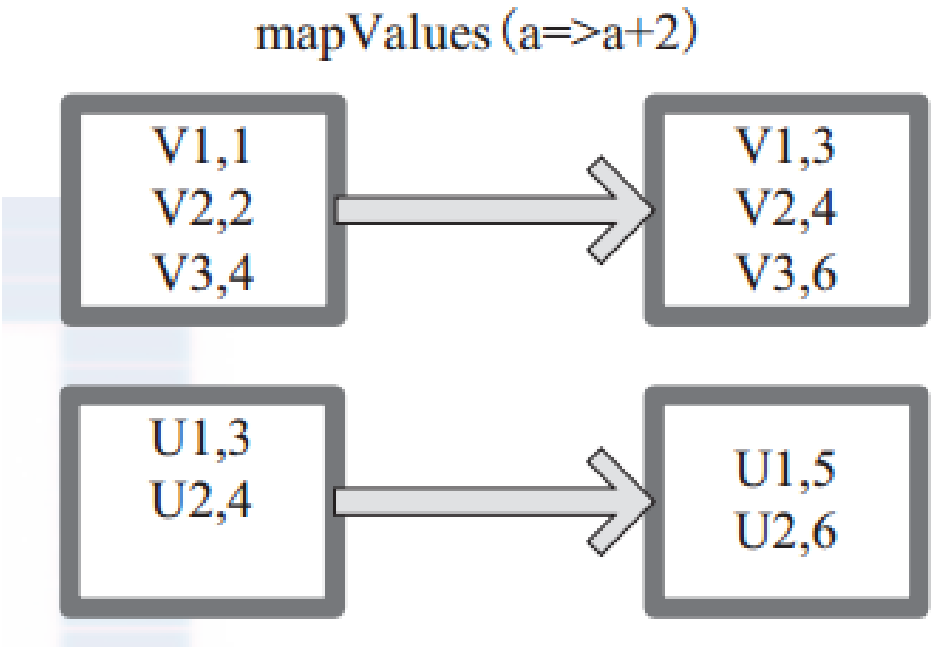
val **DISK_ONLY**: StorageLevel
val **DISK_ONLY_2**: StorageLevel
val **MEMORY_AND_DISK**: StorageLevel
val **MEMORY_AND_DISK_2**: StorageLevel
val **MEMORY_AND_DISK_SER**: StorageLevel
val **MEMORY_AND_DISK_SER_2**: StorageLevel
val **MEMORY_ONLY**: StorageLevel
val **MEMORY_ONLY_2**: StorageLevel
val **MEMORY_ONLY_SER**: StorageLevel
val **MEMORY_ONLY_SER_2**: StorageLevel
val **NONE**: StorageLevel
val **OFF_HEAP**: StorageLevel



MapValues 算子

- MapValues ：针对（Key， Value）型数据中的 Value 进行 Map 操作，而不对 Key 进行处理。

图 中的方框代表 RDD 分区RDD 分区。 a=>a+2 代表对 (V1,1) 这样的 Key Value 数据对，数据只对 Value 中的 1 进行加 2 操作，返回结果为 3。



CombineByKey 算子

- createCombiner:(V) =>C,

组合器函数，用于将V类型转换成C类型，输入参数为RDD[K,V]中的V,输出为C; 在C不存在情况下，可以通过V创建C

- mergeValue:(C,V) =>C,

合并值函数，将一个C类型和一个V类型值合并成一个C类型，输入参数为(C,V)，输出为C;当C存在的时候需要merge

- mergeCombiners:(C,C) =>C,

合并两个 C

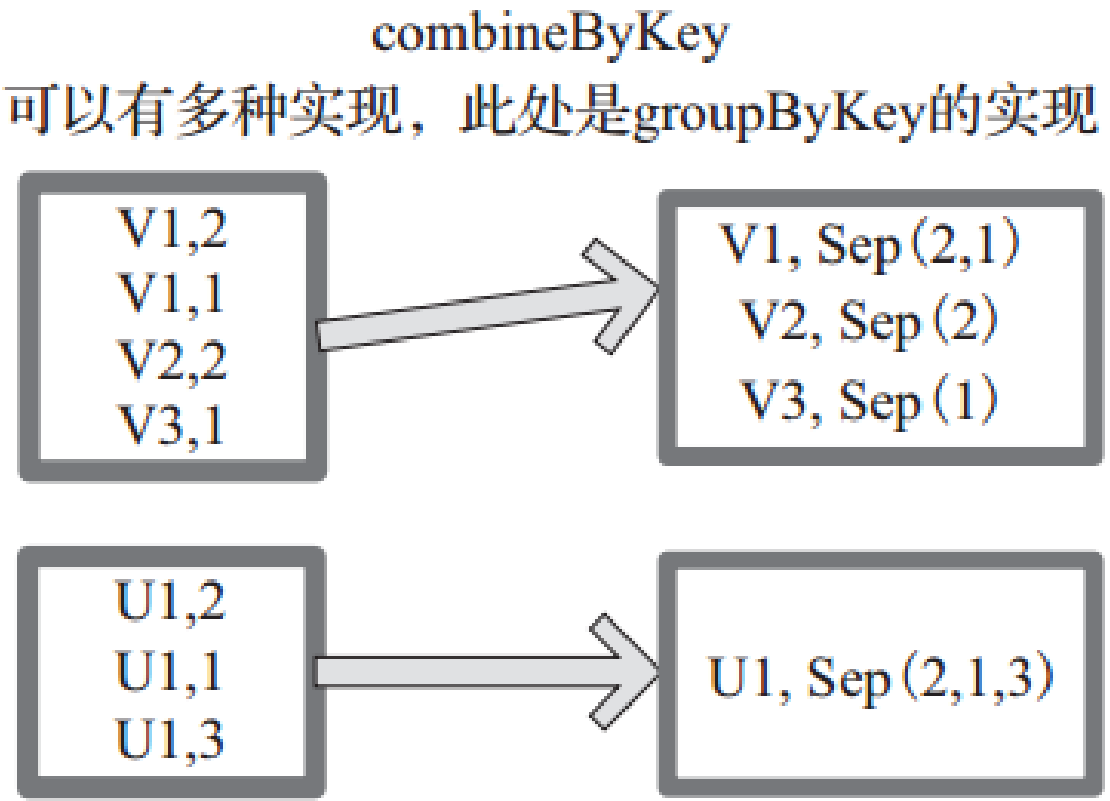
- Partitioner : Partitioner,

Partitioner, Shuff le 时需要的 Partitioner。

- mapSideCombine : Boolean=true,
- 为了减小传输量，很多 combine 可以在 map端先做，比如叠加,可以先在一个 partition 中把所有同的 key 的 value 叠加，再 shuff le。
- serializer : Serializer=null):RDD[(K,C)] 。　　//传输需要序列化，用户可以自定义序列化类

CombineByKey 算子

相当于将元素为 (Int， Int) 的 RDD 转变为了 (Int， Seq[Int]) 类型元素的 RDD。图中的方框代表 RDD 分区。通过 combineByKey， 将 (V1,2)， (V1,1)数据合并为（ V1,Seq(2,1)）。

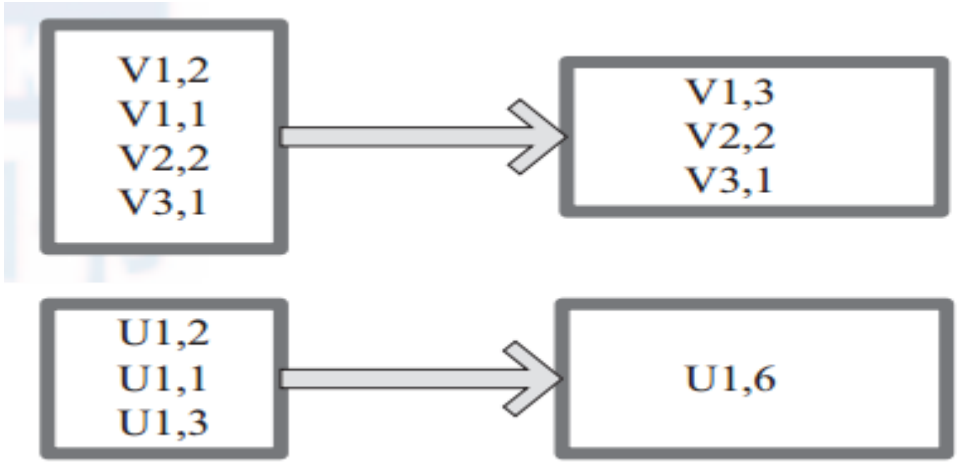


ReduceByKey 算子

- ReduceByKey 比 combineByKey 更简单的一种情况，只是两个值合并成一个值，（ Int， Int V ）to （Int， Int C），比如叠加。所以 createCombiner reduceBykey 很简单，就是直接返回 v，而 mergeValue和 mergeCombiners 逻辑是相同的，没有区别。
- 函数实现：

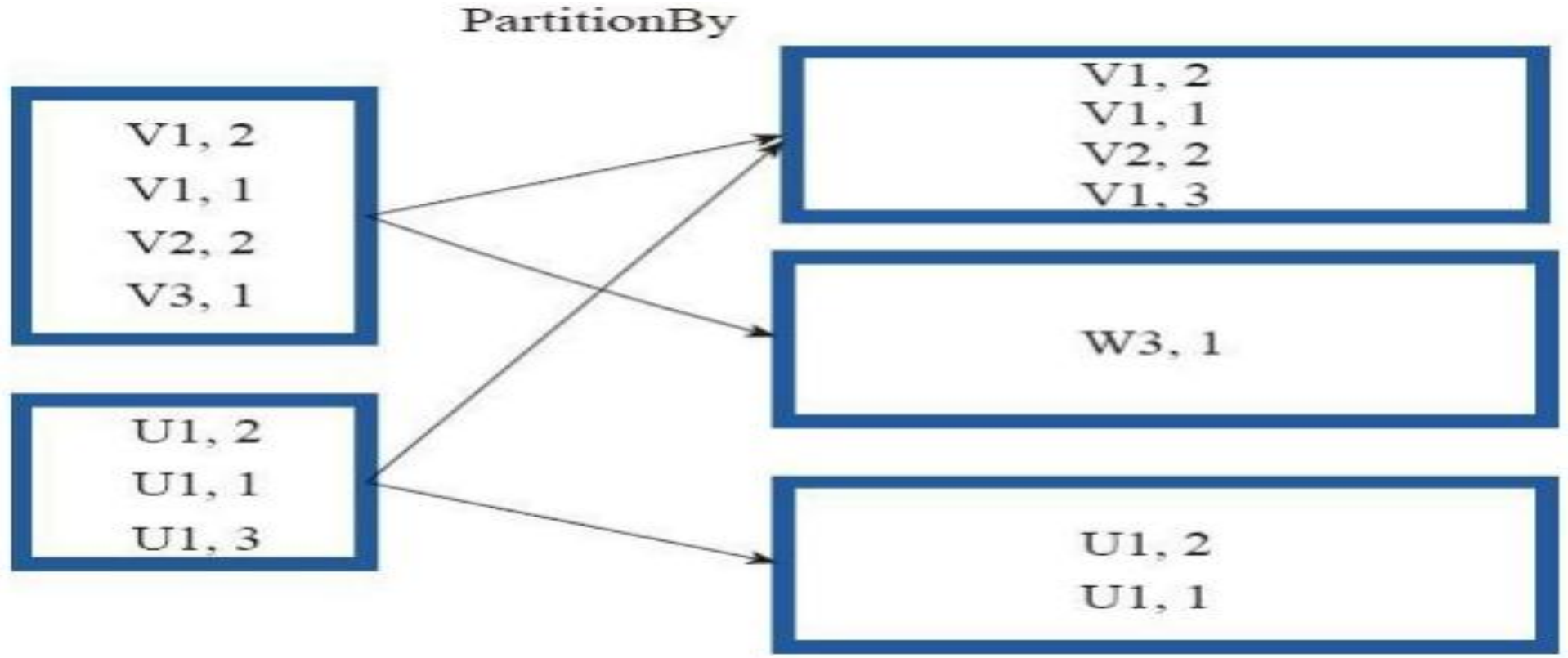
```
def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)] = self.withScope {
  combineByKeyWithClassTag[V]((v: V) => v, func, func, partitioner)
}
```

图中的方框代表 RDD 分区。通过用户自定义函数 (A,B) => (A + B) 函数，将相同 key 的数据 (V1,2) 和 (V1,1) 的 value 相加运算，结果为（ V1,3）。



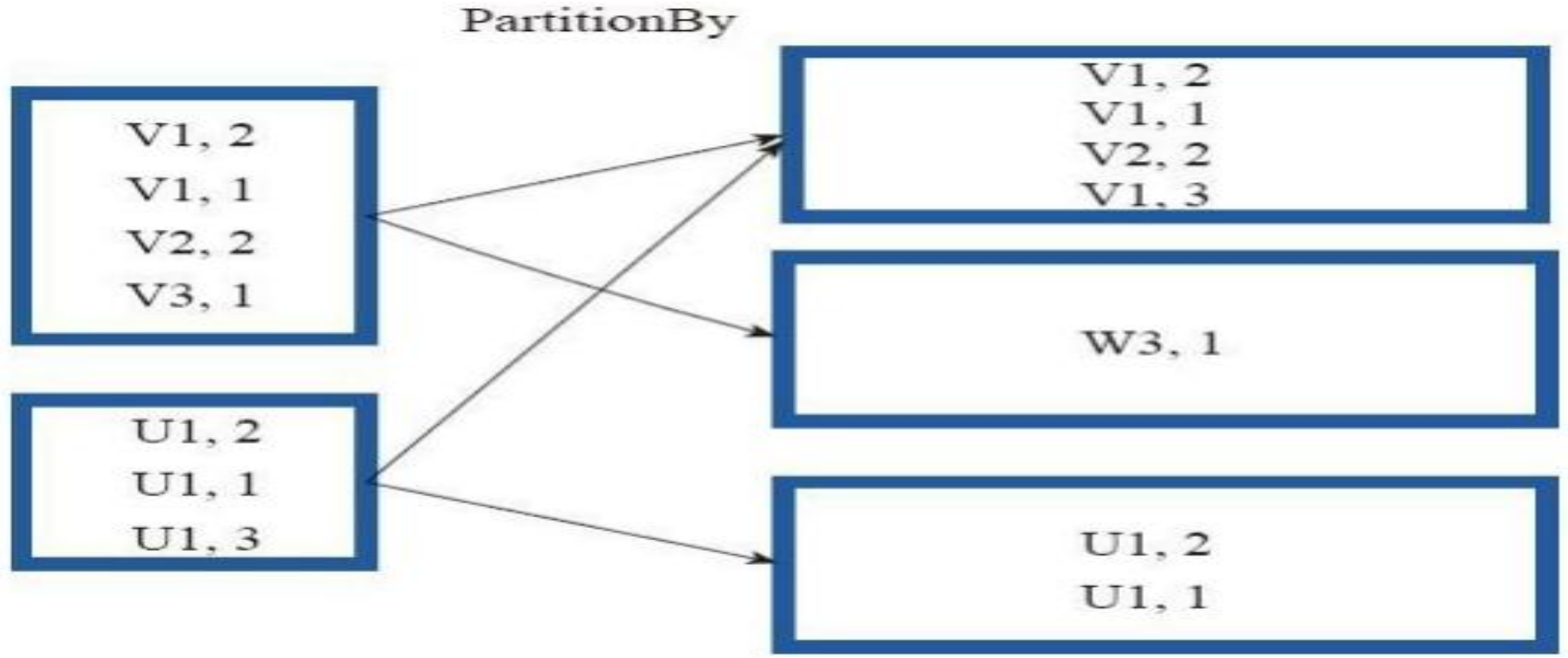
PartitionBy 算子

- PartitionBy函数对RDD进行分区操作。函数定义如下。partitionBy（partitioner：Partitioner）如果原有RDD的分区器和现有分区器（partitioner）一致，则不重分区，如果不一致，则相当于根据分区器生成一个新的ShuffledRDD。



PartitionBy 算子

- PartitionBy函数对RDD进行分区操作。函数定义如下。partitionBy（partitioner：Partitioner）如果原有RDD的分区器和现有分区器（partitioner）一致，则不重分区，如果不一致，则相当于根据分区器生成一个新的ShuffledRDD。



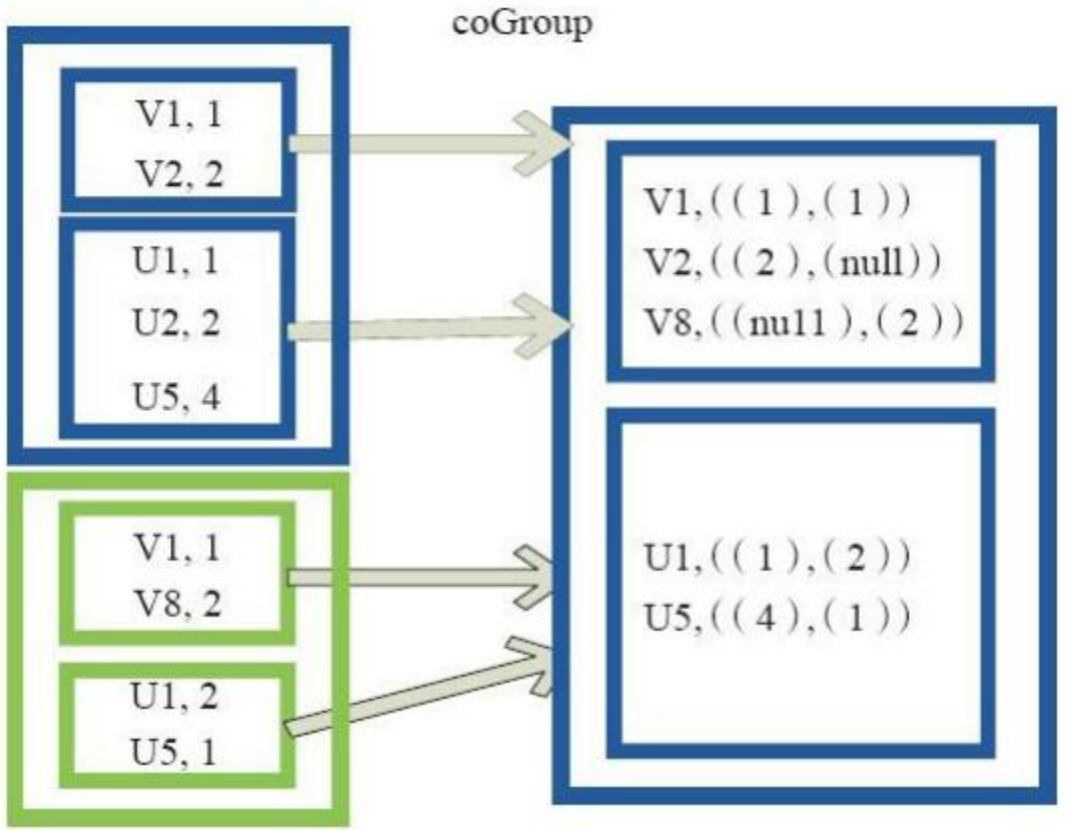
Cogroup 算子

- Cogroup 函数将两个RDD进行协同划分，cogroup

函数的定义如下：

```
def cogroup[W](other: RDD[(K, W)], partitioner: Partitioner
    : RDD[(K, (Iterable[V], Iterable[W]))] = self.withScope
```

对在两个RDD中的Key-Value类型的元素，每个RDD相同Key的元素分别聚合为一个集合，并且返回两个RDD中对应Key的元素集合的迭代器。(K,(Iterable[V]，Iterable[W]))其中，Key和Value，Value是两个RDD下相同Key的两个数据集合的迭代器所构成的元组。图中的大方框代表RDD，大方框内的小方框代表RDD中的分区。 将RDD1中的数据（U1，1）、 （U1，2）和RDD2中的数据（U1，2）合并为（U1，（（1，2），（2））。



join 算子

- join 对两个需要连接的 RDD 进行 cogroup函数操作，将相同 key 的数据能够放到一个分区，在 cogroup 操作之后形成的新 RDD 对每个key 下的元素进行笛卡尔积的操作，返回的结果再展平，对应 key 下的所有元组形成一个集合。最后返回 RDD[(K， (V， W))]。

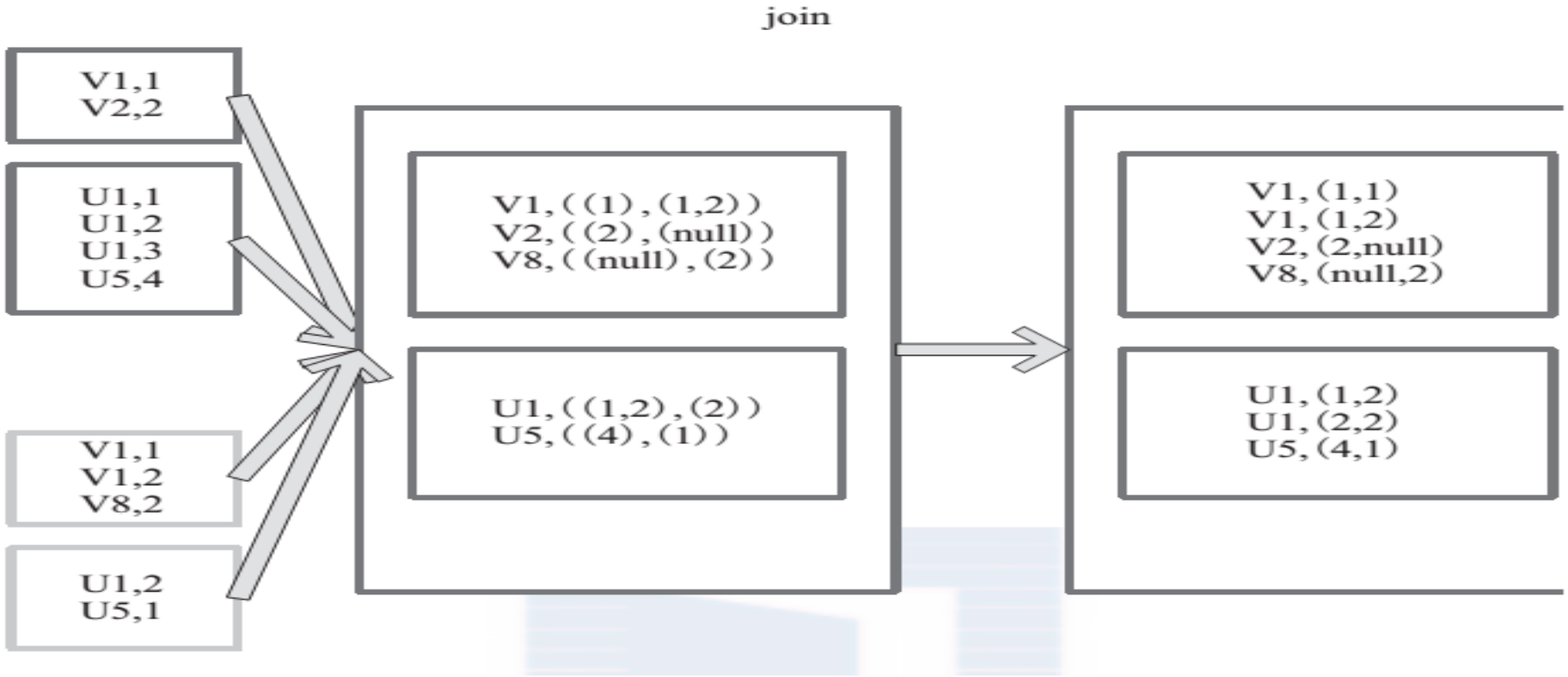
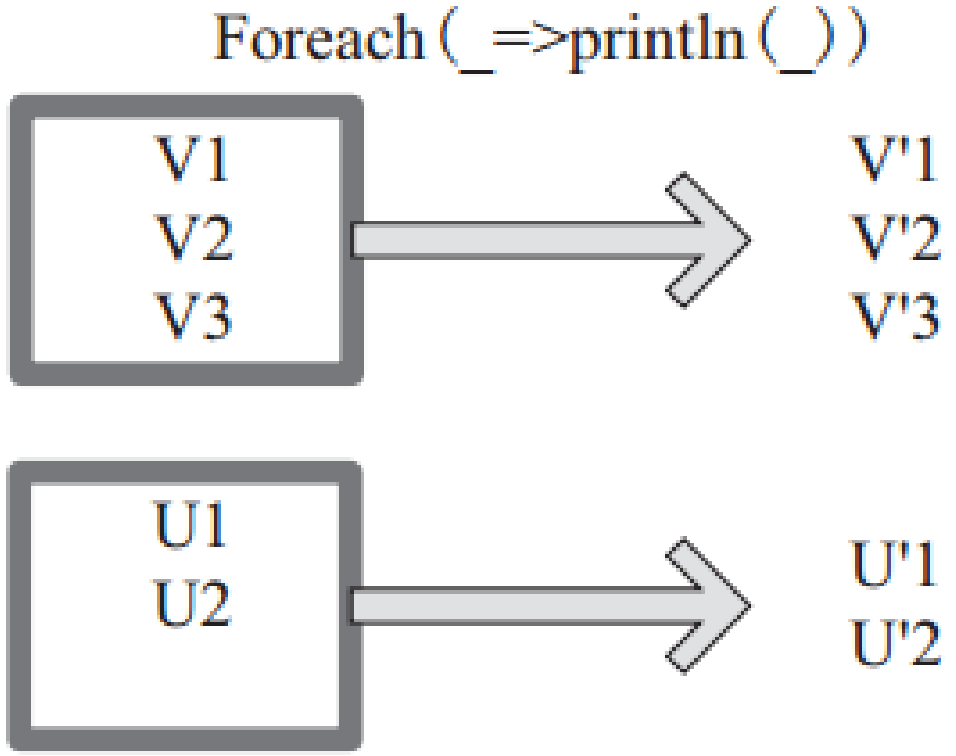


图 20　join 算子对 RDD 转换

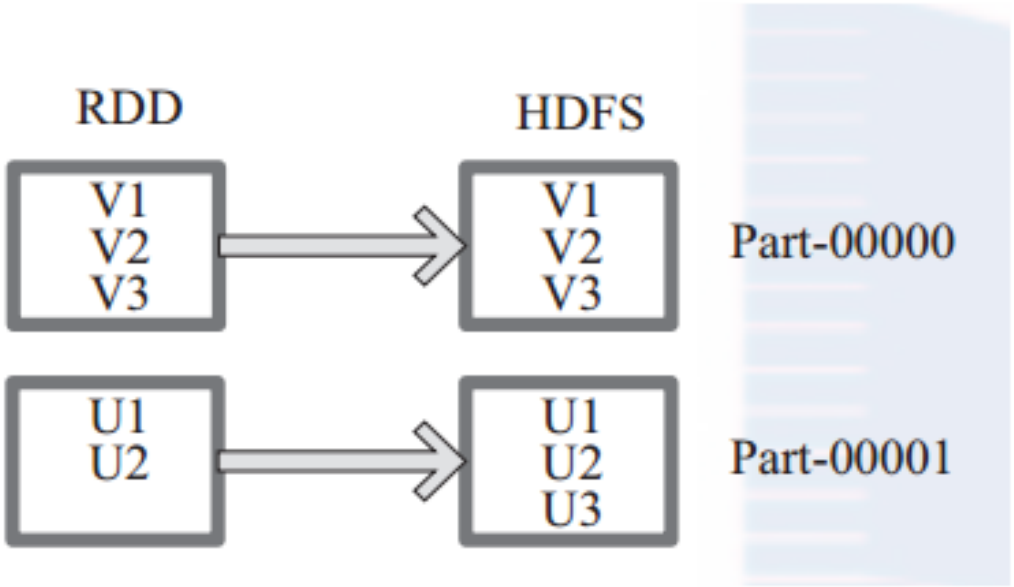
Action ——Foreach 算子

- foreach 对 RDD 中的每个元素都应用 f 函数操作，不返回 RDD 和 Array， 而是返回Unit。下图表示 foreach 算子通过用户自定义函数对每个数据项进行操作。本例中自定义函数为 println()，控制台打印所有数据项。



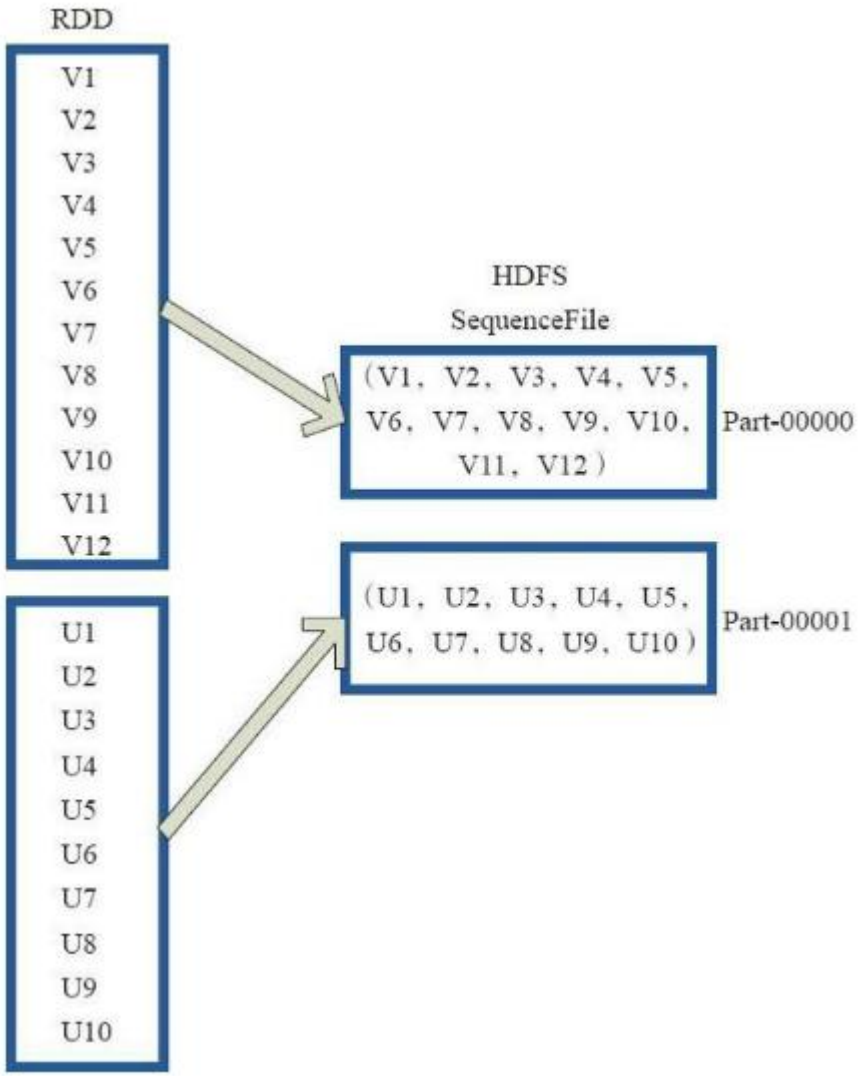
Action——SaveAsTextFile 算子

- 函数将数据输出，存储到 HDFS 的指定目录。



Action——SaveAsObjectFile 算子

- SaveAsObjectFile将分区中的每10个元素组成一个Array，然后将这个Array序列化，写入HDFS为SequenceFile的格式。

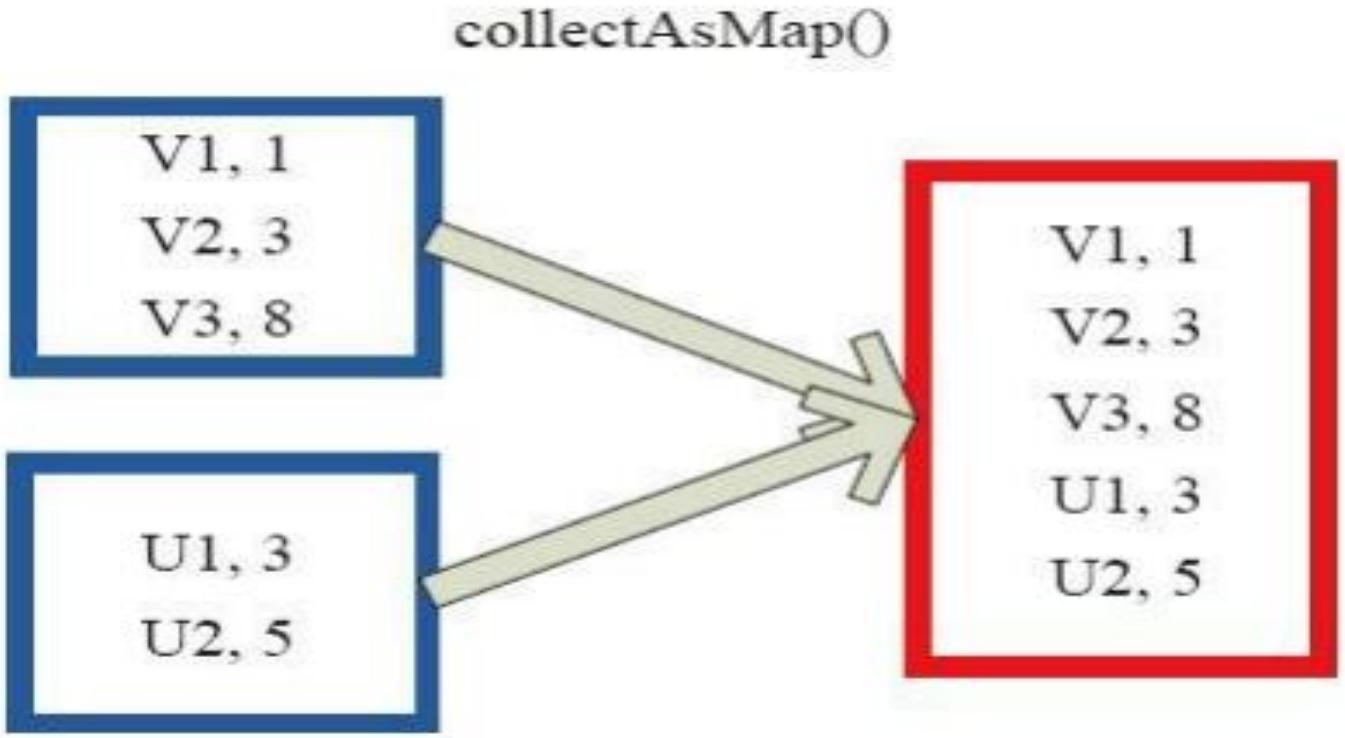


Action——Collect 算子

- Collect 相当于 toArray， toArray 已经过时不推荐使用， collect 将分布式的 RDD 返回为一个单机的 scala Array 数组。在这个数组上运用 scala 的函数式操作。
- Collect 算子在使用的时候需要注意Driver 端内存是否能够承载这么大量的数据。

Action——CollectAsMap 算子

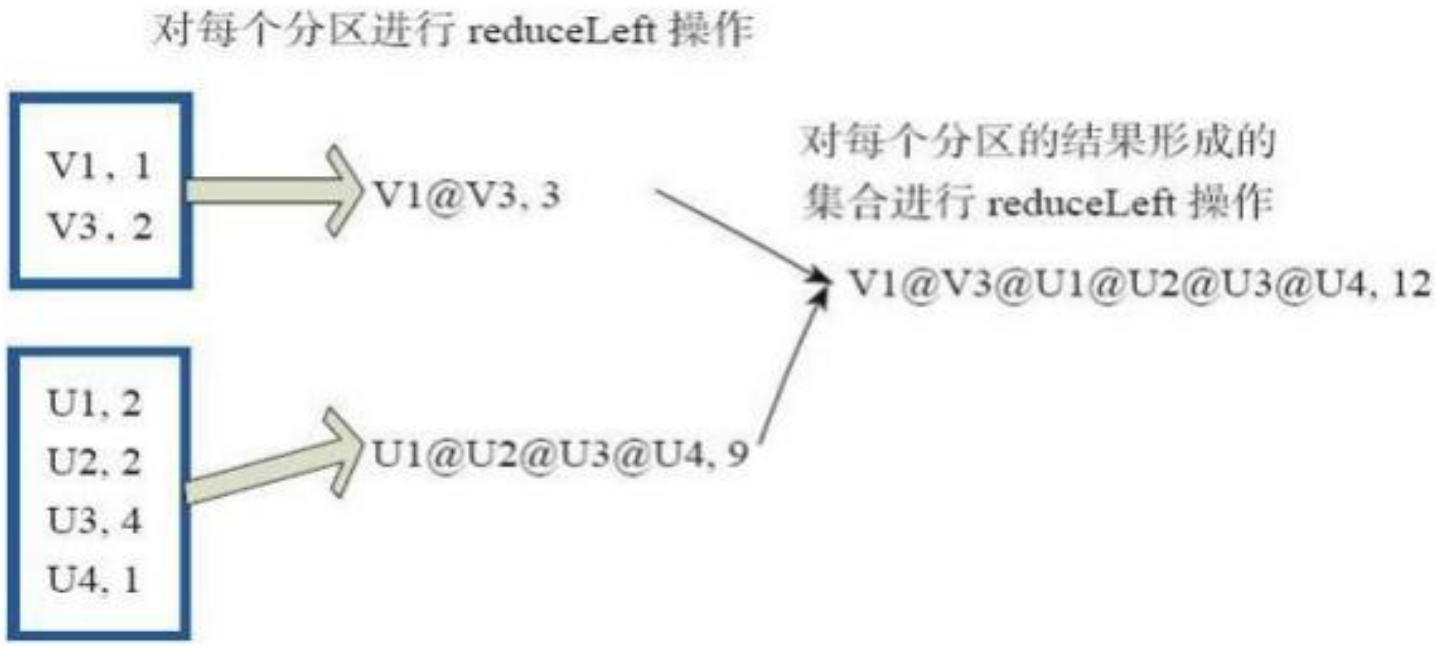
- CollectAsMap对（K，V）型的RDD数据返回一个单机HashMap。 对于重复K的RDD元素，后面的元素覆盖前面的元素。



Action——Reduce 算子

- Reduce函数相当于对RDD中的元素进行 reduceLeft 函数的操作。

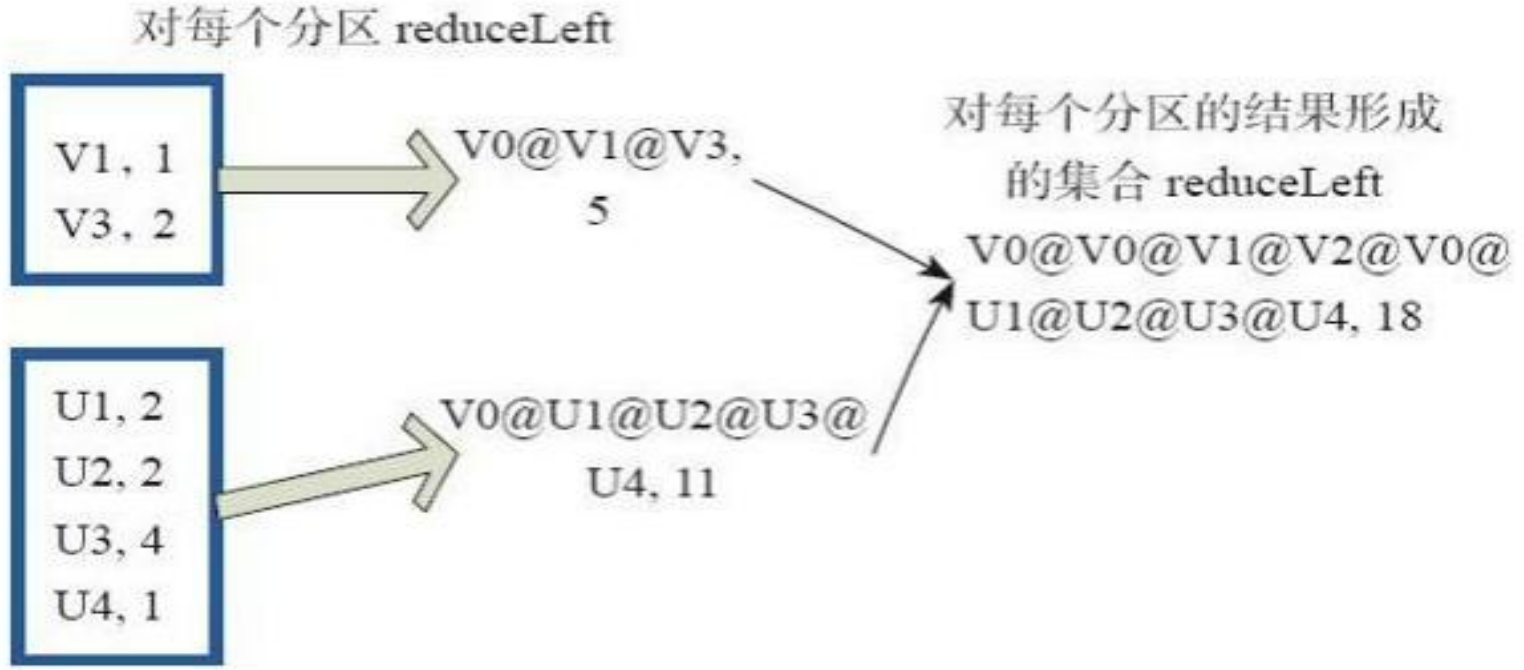
reduceLeft先对两个元素<K，V>进行reduce函数操作，然后将结果和迭代器取出的下一个元素<k，V>进行reduce函数操作，直到迭代器遍历完所有元素，得到最后结果。在RDD中，先对每个分区中的所有元素<K，V>的集合分别进行reduceLeft。 每个分区形成的结果相当于一个元素<K，V>，再对这个结果集合进行reduceleft操作。



Action——Fold 算子

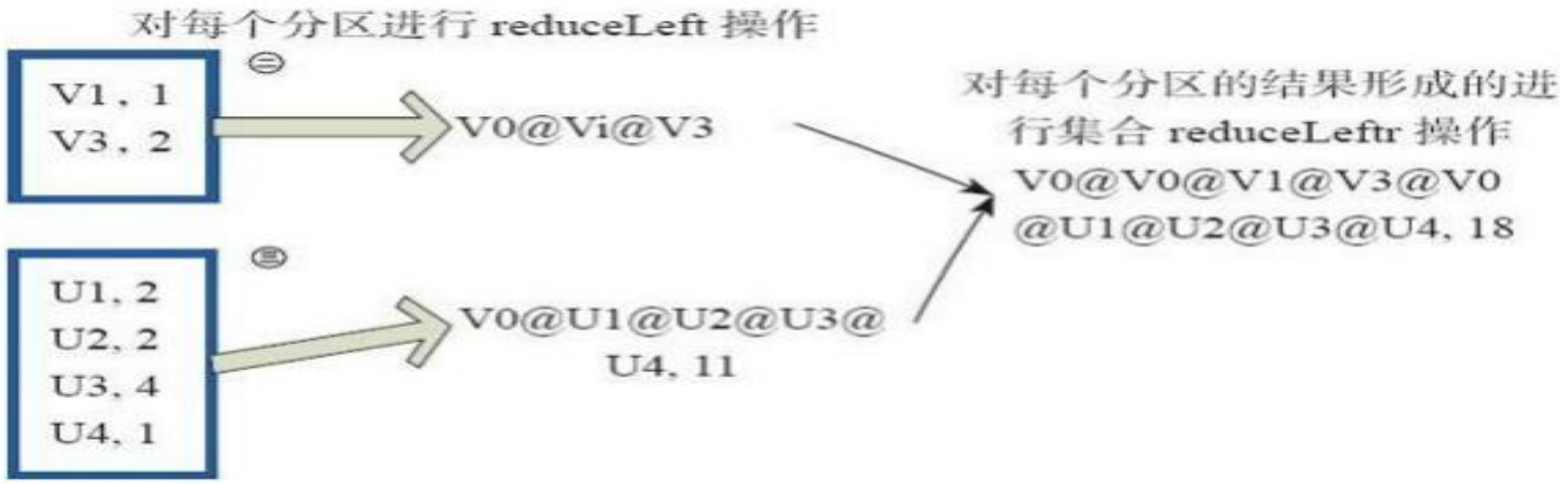
- Fold和reduce的原理相同，但是与reduce不同，相当于每个reduce时，迭代器取的第一个元素是zeroValue。

图中通过下面的用户自定义函数进行fold运算，图中的一个方框代表一个RDD分区。 读者可以参照reduce 函 数 理 解 。 Fold (("V0@" ， 2))((A ， B)=>(A._1+"@"+B._1 ， A._2+B._2))



Action——Aggregate 算子

- Aggregate先对每个分区的所有元素进行aggregate操作，再对分区的结果进行fold操作。aggreagate与fold和reduce的不同之处在于，aggregate相当于采用归并的方式进行数据聚集，这种聚集是并行化的。 而在fold和reduce函数的运算过程中，每个分区中需要进行串行处理，每个分区串行计算完结果，结果再按之前的方式进行聚集，并返回最终聚集结果。



综合案例

- 数据集

  http://grouplens.org/datasets/movielens/

  MovieLens 1M Dataset

- 相关数据文件
- **users.dat**
- **UserID::Gender::Age::Occupation::Zip-code**
- **movies.dat**
- **MovieID::Title::Genres**
- **ratings.dat**
- **UserID::MovieID::Rating::Timestamp**
- 通过Spark 计算看过“Lord of the Rings, The (1978)”用户年龄和性别分布

课后习题

- 1. 年龄段在“18-24”的男性年轻人，最喜欢看哪10部
- 2.得分最高的10部电影；看过电影最多的前10个人；女性看多最多的10部电影；男性看过最多的10部电影
- 3.利用数据集SogouQ2012.mini.tar.gz 将数据按照访问次数进行排序，求访问量前10的网站


联系我们
如何联系我们...

数据 引领变革

缔造新型 价值

联系我们

地址： 沈阳市和平区三好街84-8号易购大厦319A

电话： 024-88507865

邮箱： horizon@syhc.com.cn

公司网站： http://www.syhc.com.cn

THANKS