JerryLead/SparkInternals

Spark Internals

Spark Version: 1.0.2 Doc Version: 1.0.2.0

Authors

Weibo Id	Name
@JerryLead	Lijie Xu

Introduction

This series discusses the design and implementation of Apache Spark, with focuses on its design principles, execution mechanisms, code architecture and performance optimization. In addition, there's some comparisons with Hadoop MapReduce in terms of design and implementation. I'm reluctant to call this document a "code walkthrough" because the goal is not to analyze each piece of code in the project, but to understand the whole system in a systematic way, through the process of a Spark job, from its creation to completion.

There're many ways to discuss a computer system, here I've chosen a **problem-driven** approach. Firstly one concret problem is introduced, then it gets analyzed step by step. We'll start from a typical Spark example job to discuss all the system modules and supports it needs for its creation and execution, to give a big picture. Then we'll selectively go into the design and implementation details of some system modules. I believe that this approach is better than diving into each module right from the beginning.

The target audience of this series are geeks who want to have a deeper understanding of Apache Spark as well as other distributed computing frameworks.

I'll try my best to keep this documentation up to date with Spark since it's a fast evolving project with an active community. The documentation's main version is in sync with Spark's version. The additional number at the end represents the documentation's update version.

For more academic oriented discussion, please check Matei's PHD paper and other related papers. You can also have a look my blog (in Chinese) blog.

I haven't been writing such complete documentation for a while. Last time it was about three years ago when studying Andrew Ng's ML course. I was really motivated at that time! This time I've spent 20+ days

on this document, from the summer break till now. Most of the time is spent on debugging, drawing diagrams and thinking how to put my ideas in the right way. I hope you find this series helpful.

Contents

We start from the creation of a Spark job, then discuss its execution, finally we dive into some related system modules and features.

1. Overview Overview of Apache Spark
2. Job logical plan Logical plan of a job (data dependency graph)
3. Shuffle details Shuffle process
4. Architecture Coordination of system modules in job execution
5. Cache and Checkpoint Cache and Checkpoint
6. Broadcast Broadcast feature
7. Job Scheduling TODO
8. Fault-tolerance TODO

The documentation is written in markdown. The pdf version is also available here.

If you're under Max OS X, I recommand MacDown with a github theme for reading.

Examples

I've created some examples to debug the system during the writing, they are avaible under SparkLearning/src/internals.

Acknowledgement

I appreciate the help from the following in providing solutions and ideas for some detailed issues:

- @Andrew-Xia Participated in the discussion of BlockManager's implemetation's impact on broadcast(rdd).
- @CrazyJVM Participated in the discussion of BlockManager's implementation.
- @王联辉 Participated in the discussion of BlockManager's implementation.

Thanks to the following for complementing the document:

Weibo Id	Chapter	Content	Revision status
			There's not yet a conclusion on

@OopsOutOfMemory	Overview	Relation between workers and executors and Summary on Spark Executor Driver's Resouce Management (in Chinese)	this subject since its implementation is still changing, a link to the blog is added

Thanks to the following for finding errors:

Weibo Id	Chapter	Error/Issue	Revision status
@Joshuawangzj	Overview	When multiple applications are running, multiple Backend process will be created	Corrected, but need to be confirmed. No idea on how to control the number of Backend processes
@ cs cm	Overview	Latest groupByKey() has removed the mapValues() operation, there's no MapValuesRDD generated	Fixed groupByKey() related diagrams and text
@染染生起	JobLogicalPlan	N:N relation in FullDepedency N:N is a NarrowDependency	Modified the description of NarrowDependency into 3 different cases with detaild explaination, clearer than the 2 cases explaination before
@zzl0	Fisrt four chapters	Lots of typos，such as "groupByKey has generated the 3 following RDDs", should be 2. Check pull request。	All fixed
@左手牵右手TEL	Cache and Broadcast chapter	Lots of typos	All fixed
@cloud-fan	JobLogicalPlan	Some arrows in the Cogroup() diagram should be colored red	All fixed
@CrazyJvm	Shuffle details	Starting from Spark 1.1, the default value for spark.shuffle.file.buffer.kb is	All fixed

		32k, not 100k	

Special thanks to @明风Andy for his great support.

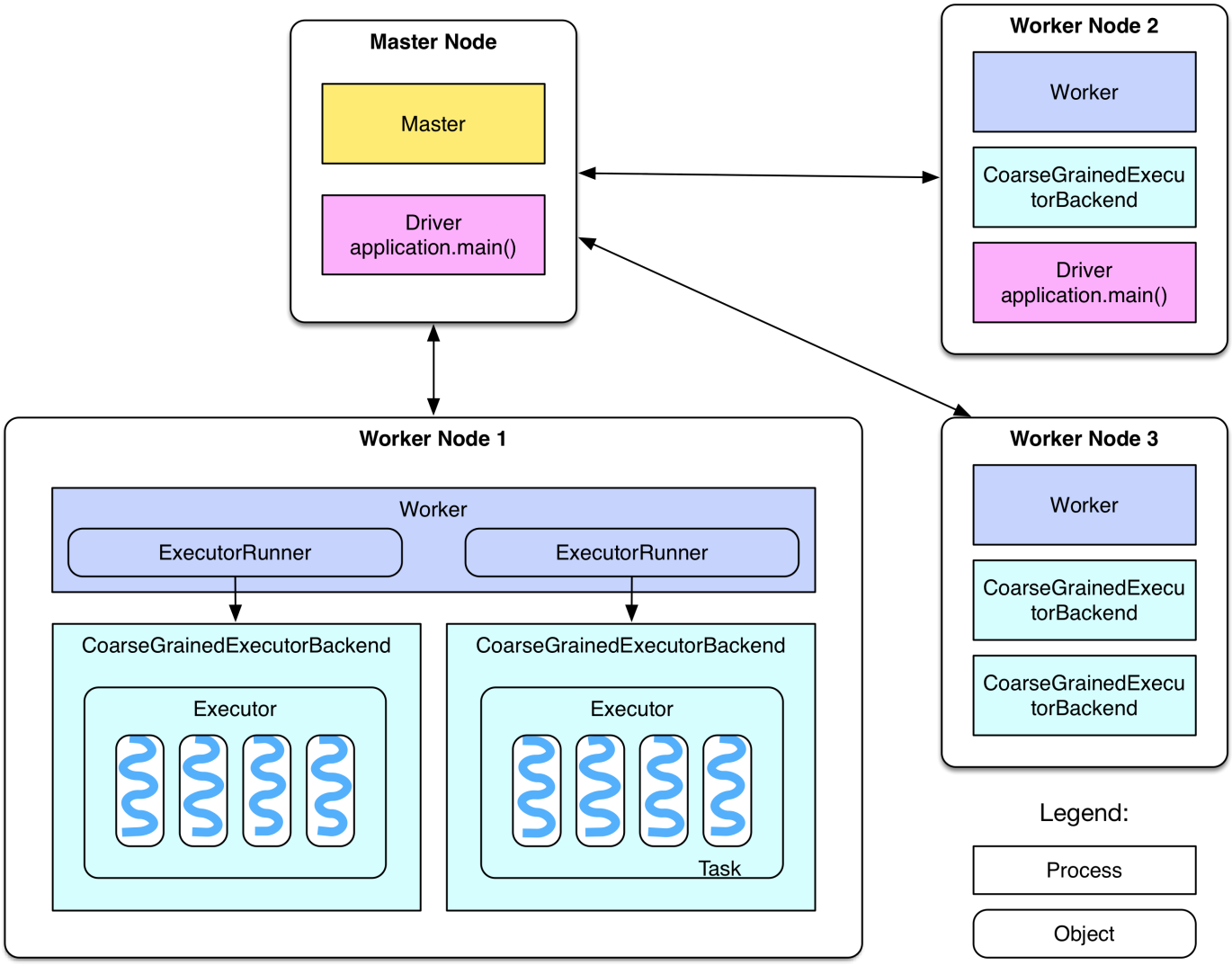
Special thanks to the rockers (including researchers, developers and users) who participate in the design, implementation and discussion of big data systems.

JerryLead/SparkInternals

Overview

Firstly we'll have a look at Spark's deployment. The question here is: **after a successful deployment, what are the services launched by each node of the cluster?**

Deployment Diagram



We can see from the diagram:

- There's Master node and Worker node in the cluster, they are equivalent to Hadoop's Master and Slave node
- Master node has a Master daemon process, managing all worker nodes
- Worker node has a Worker daemon process, responsible for communicating with the master node and

for managing local executors

- In the official document, the Driver is explained as "The process running the main() function of the application and creating the SparkContext". The application is the user's program (driver program), such as WordCount.scala. If the driver program is running on the Master node, for example if we run the this on the Master node

  ```
  ./bin/run-example SparkPi 10
  ```

  Then the SparkPi program will be the Driver on the Master node. In case of a YARN cluster, the Driver may be scheduled to a Worker node (for example Worker node 2 in the diagram). If the driver program is run on a local PC, such as running from within Eclipse with

  ```
  val sc = new SparkContext("spark://master:7077", "AppName")
  ```

  Then the driver program will be on the local machine. However this is not a recommended way of running Spark since the local machine may not be in the same network with the Worker nodes, which will slow down the communication between the driver and the executors

- There may have one or multiple ExecutorBackend processes in each Worker node, each one possesses an Executor instance. Each Executor object maintains a thread pool. Each task runs on a thread.

- Each application has one driver and multiple executors. All tasks within the same executor belongs to the same application
- In Standalone deployment mode, ExecutorBackend is instantiated as CoarseGrainedExecutorBackend

  > In my cluster there's only one CoarseGrainedExecutorBackend process on each worker and I didn't manage to configure multiple instances of it (my guess is that there'll be multiple CoarseGrainedExecutorBackend process when when multiple applications are running, need to be confirmed). Check this blog (in Chinese) Summary on Spark Executor Driver Resource Scheduling by @OopsOutOfMemory if you want to know more about the relation between Worker and Executor.

- Worker controls the CoarseGrainedExecutorBackend by using a ExecutorRunner

After the deployment diagram, we'll examine an example job to see how a Spark job is created and executed.

Example Spark Job

The example here is the GroupByTest application under the examples package in Spark. We assume that the application is run on the Master node, with the following command:

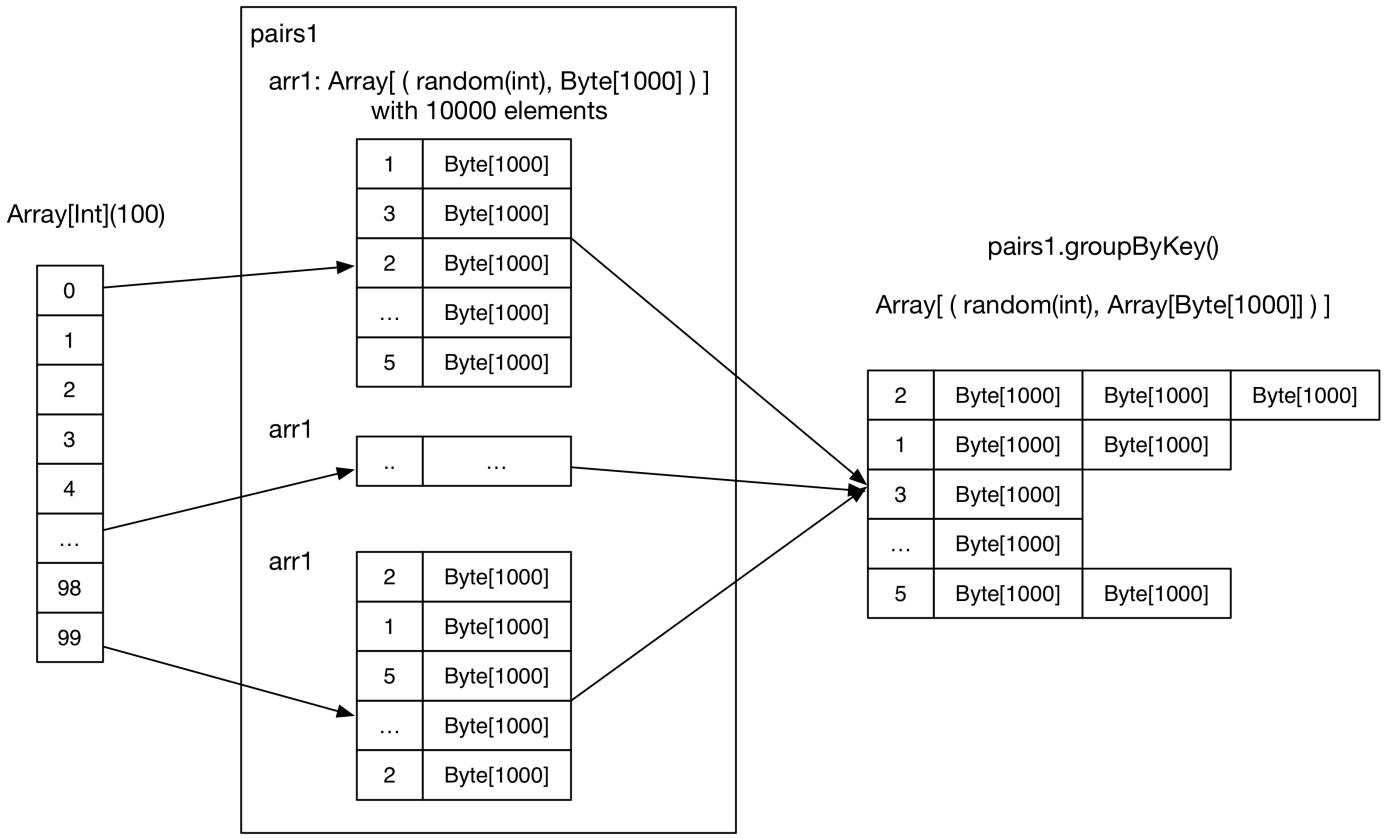
```
/* Usage: GroupByTest [numMappers] [numKVPairs] [valSize] [numReducers] */

bin/run-example GroupByTest 100 10000 1000 36
```

The code of this application is the following:

```
package org.apache.spark.examples

import java.util.Random

import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._

/**
  * Usage: GroupByTest [numMappers] [numKVPairs] [valSize] [numReducers]
  */
object GroupByTest {
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("GroupBy Test")
    var numMappers = 100
    var numKVPairs = 10000
    var valSize = 1000
    var numReducers = 36

    val sc = new SparkContext(sparkConf)

    val pairs1 = sc.parallelize(0 until numMappers, numMappers).flatMap { p
=>
      val ranGen = new Random
      var arr1 = new Array[(Int, Array[Byte])](numKVPairs)
      for (i <- 0 until numKVPairs) {
        val byteArr = new Array[Byte](valSize)
        ranGen.nextBytes(byteArr)
        arr1(i) = (ranGen.nextInt(Int.MaxValue), byteArr)
      }
      arr1
    }.cache
    // Enforce that everything has been calculated and in cache
    pairs1.count

    println(pairs1.groupByKey(numReducers).count)
```

```
    sc.stop()
  }
}
```

After reading the code, we should have an idea about how the data get transformed:



This is not a complicated application, let's estimate the data size and the result:

1. Initialize SparkConf
2. Initialize numMappers=100, numKVPairs=10,000, valSize=1000, numReducers= 36
3. Initialize SparkContext. This is an important step, the SparkContext contains objectes and actors that are needed for the creation of a driver
4. For each mapper, a `arr1: Array[(Int, Byte[])]` is created, with a length of numKVPairs. Each byte array's size is valSize, a randomly generated integer. We may estimate `Size(arr1) = numKVPairs * (4 + valSize) = 10MB`, and we have `Size(pairs1) = numMappers * Size(arr1) = 1000MB`
5. Each mapper is instructed to cache its `arr1` array into the memory
6. Then an action, count(), is applied to compute the size of `arr1` for all mappers, the result is `numMappers * numKVPairs = 1,000,000`. This action triggers the caching of `arr1`s
7. groupByKey operation is executed on `pairs1` which is already cached. The reducer number (or partition number) is numReducers. Theoretically, if hash(key) is well distributed, each reducer receives

`numMappers * numKVPairs / numReducer = 27,777` pairs of (Int, Array[Byte]), with a size of `Size(pairs1) / numReducer = 27MB`

8. Reducer aggregates the records with the same Int key, the result is `(Int, List(Byte[], Byte[], ..., Byte[]))`
9. Finally a count action sums up the record number in each reducer, the final result is actually the number of distinct integers in `paris1`

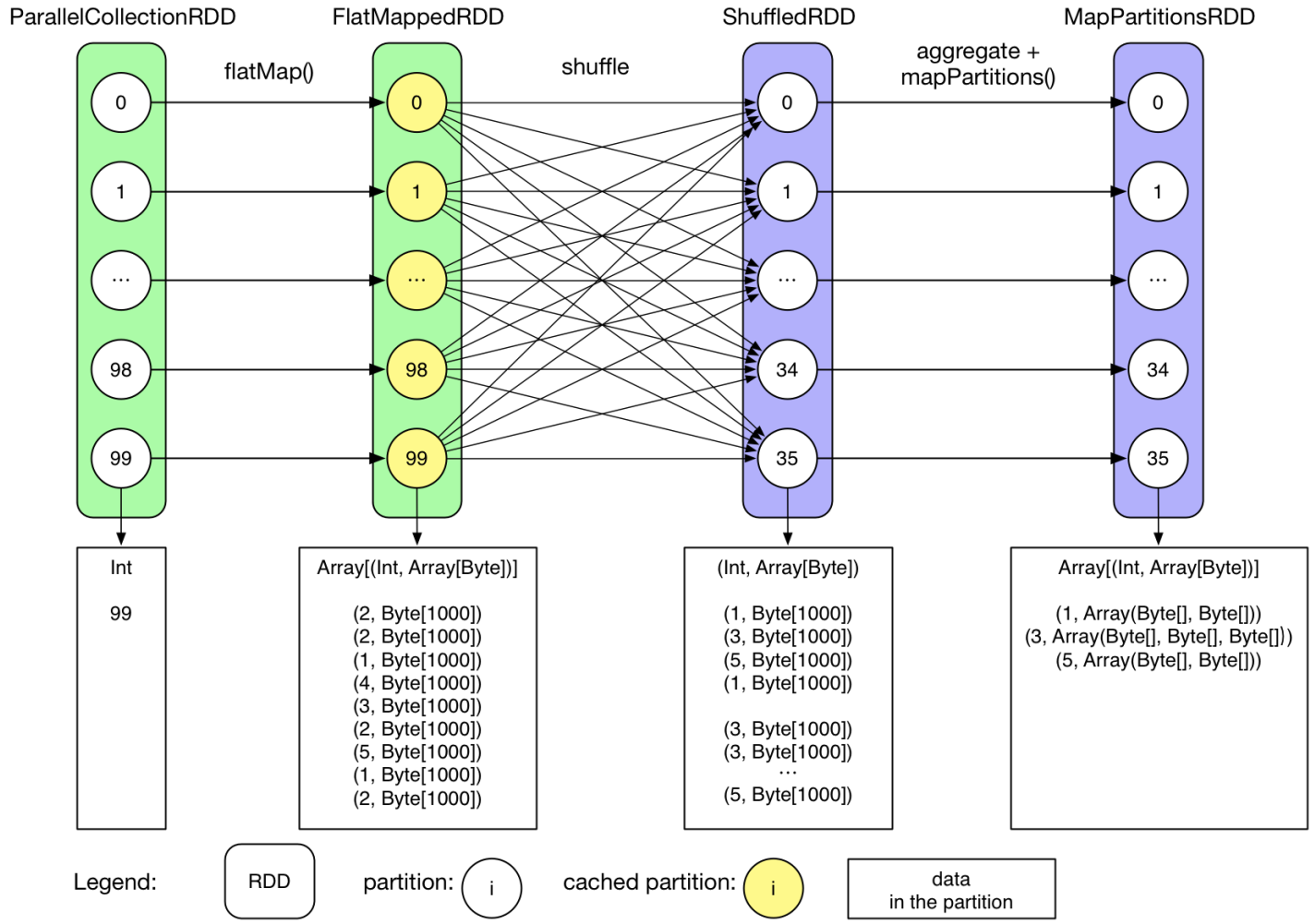
Logical Plan

The actual execution process of an application is more complicated than the above diagram. Generally speaking, a logical plan (or data dependency graph) will be created, then a physical plan (in the form of a DAG) will be generated. After that, concrete tasks will be generated and executed. Let's check the logical plan of this application:

A call of function `RDD.toDebugString` will return the logical plan:

```
  MapPartitionsRDD[3] at groupByKey at GroupByTest.scala:51 (36 partitions)
    ShuffledRDD[2] at groupByKey at GroupByTest.scala:51 (36 partitions)
      FlatMappedRDD[1] at flatMap at GroupByTest.scala:38 (100 partitions)
        ParallelCollectionRDD[0] at parallelize at GroupByTest.scala:38 (100
partitions)
```

We can also draw a diagram:



> Notice that the `data in the partition` blocks shows the final result of the partitions, this does not necessarily mean that all these data resides in the memory in the same time

So we could conclude:

- User initiated an array from 0 to 99: `0 until numMappers`
- parallelize() generate the initial ParallelCollectionRDD, each partititon contains an integer i
- FlatMappedRDD is generated by calling a transformation method (flatMap) on the ParallelCollectionRDD. Each partition of the FlatMappedRDD contains an `Array[(Int, Array[Byte])]`
- When the first count() executes, this action executes on each partition, the results are sent to the driver and are summed up in the driver
- Since the FlatMappedRDD is cached in memory, so its partitions are colored differently
- groupByKey() generates the following 2 RDDs (ShuffledRDD and MapPartitionsRDD), we'll see in later chapters why it is the case
- Usually we see a ShuffleRDD if the job needs a shuffle. Its relation with the former RDD is similar to the relation between the output of mappers and the input of reducers in Hadoop
- MapPartitionRDD contains groupByKey()'s result
- Each value in MapPartitionRDD (`Array[Byte]`) is converted to `Iterable`

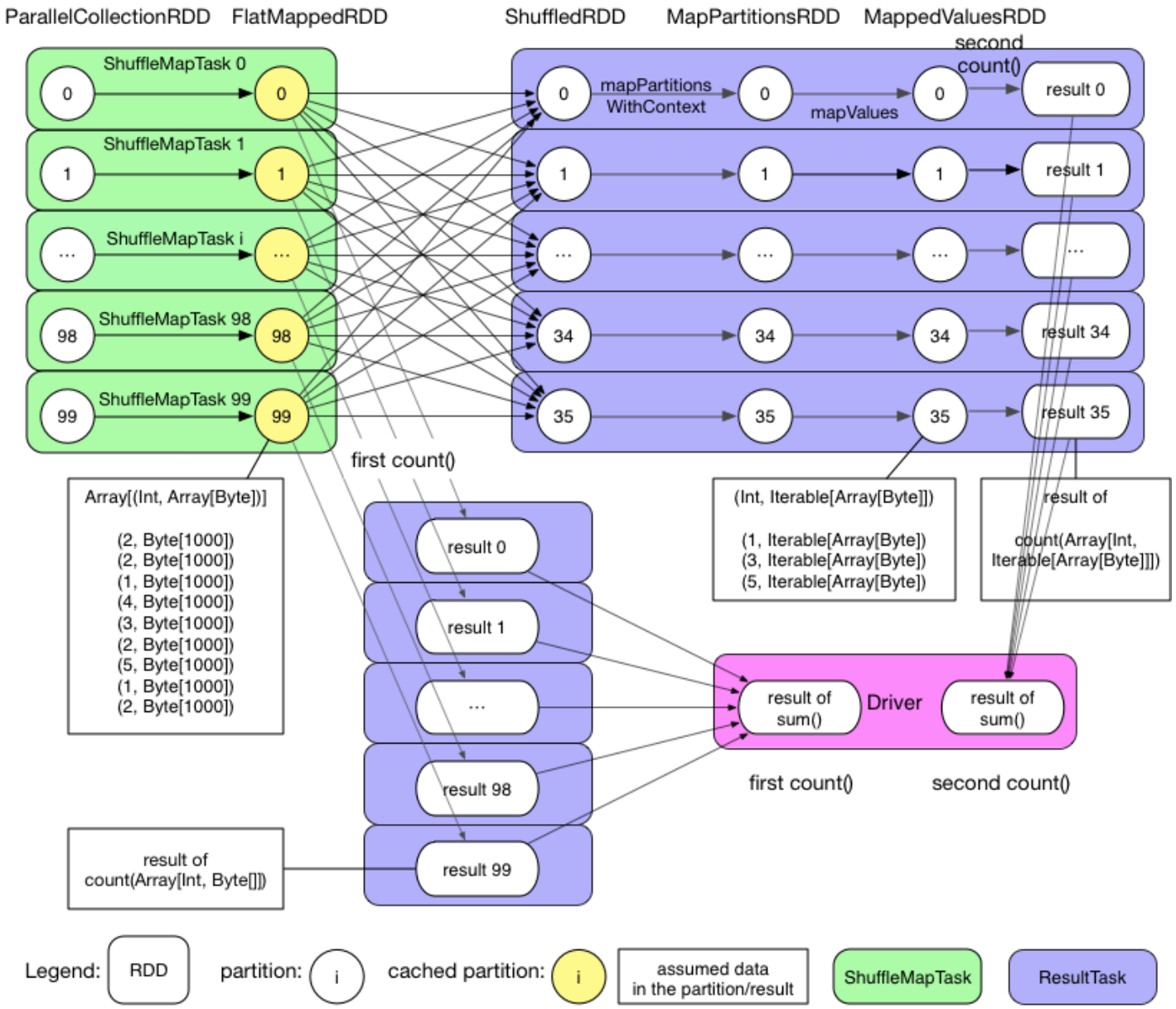
- The last count() action is executed in the same way as we explained above

We can see that the logical plan describes the data flow of the application: the transformations that are applied to the data, the intermediate RDDs and the dependency between these RDDs.

Physical Plan

As we've found out, the logical plan is about the dependency of data, not the actual execution of tasks. This is a main difference compared to Hadoop. In Hadoop, user handles directly the physical tasks: mapper tasks for applying operations on partitions and reducers tasks for aggregation. This is because in Hadoop, the data flow is pre-defined and fixed, user just fills in the map() and reduce() function. While in Spark, the data flow is very flexible and could be complicated, so it's difficult to simply combine together the concept of data dependency and the physical tasks. For this reason, Spark separates the data flow and the actual task execution process, and has algorithms to transform a logical plan into a physical plan. We'll discuss this transformation in later chapter.

For the example job, let's draw its physical DAG:



We can see that the GroupByTest application generates 2 jobs, the first job is triggered by the first action (that is `pairs1.count()`). Let's check this first job:

- The job contains only 1 stage (now we only need to know that there's a concept called stage, we'll see it in detail in later chapters)
- Stage 0 has 100 ResultTask
- Each task computes flatMap, generating FlatMappedRDD, then executes the action (`count()`), count the record number in each partition. For example in partition 99 there's only 9 records.
- Since `pairs1` is instructed to be cached, the tasks will cache the partitions of FlatMappedRDD inside the executor's memory space.
- After the tasks' execution, drive collects the results of tasks and sum them up
- Job 0 completes

The second job is triggered by `pairs1.groupByKey(numReducers).count`:

- There's 2 stages in this job
- Stage 0 contains 100 ShuffleMapTask, each task reads a part of `paris1` from the cache, partitions it, and write the partition results on local disk. For example, the task will place the a record `(1, Array(...))` in the bucket of key 1 and store it on the disk. This step is similar to the partitioning of mappers in Hadoop
- Stage 1 contains 36 ResultTask, each task fetches and shuffles the data that it need to process. It does aggregation in the same time as fetching data and the mapPartitions() operation, then count() is applied to get the result. For example the ResultTask responsible for bucket 3 will fetch all data of bucket 3 from the workers and aggregates them locally
- After the tasks' execution, drive collects the results of tasks and sum them up
- Job 1 completes

We can see that the physical plan is not simple. An Spark application can contain multiple jobs, each job could have multiple stages, and each stage has multiple tasks. **Later we'll see how the jobs are defined as well as how the stages and tasks are created**

Discussion

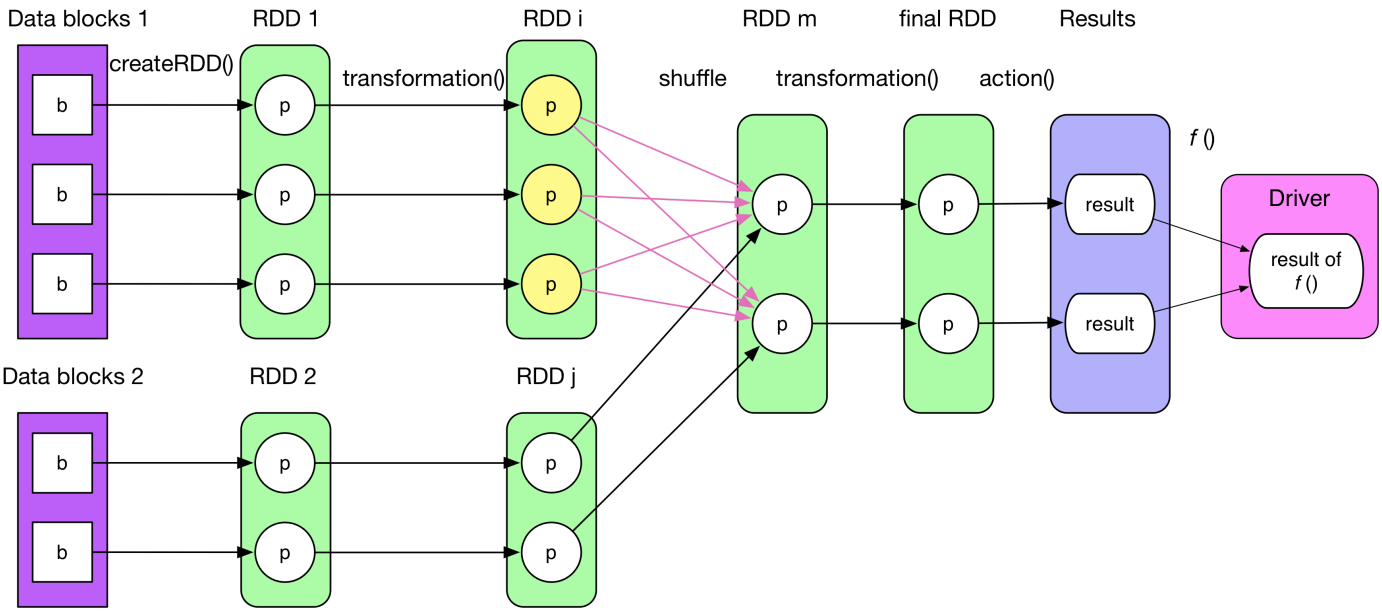
So now we have a basic knowledge about a Spark job's creation and execution. We also discussed the cache feature of Spark. In the following chapters we'll discuss in detail the key steps related to job creation and execution, including:

1. logical plan generation
2. physical plan generation
3. Job submission and scheduling
4. Task's creation, execution and result handling
5. How shuffle is done in Spark
6. Cache mechanism
7. Broadcast mechanism

JerryLead/SparkInternals

Job Logical Plan

An example of general logical plan



The picture above illustrates a general job logical plan which takes 4 steps to get the final result:

1. Construct the initial `RDD` from any source.(in-memory data, local file, HDFS, HBase, etc). (Note that `parallelize()` is equivalent to `createRDD()` mentioned in the previous chapter)

2. A series of *transformation operations* on `RDD`, denoted as `transformation()`, each `transformation()` produces one or more `RDD[T]`s, where `T` can be any type in Scala.

   > Need to mention that, for key-value pair `RDD[(K, V)]`, it will be handy if `K` is a basic type, like `Int`, `Double`, `String`, etc. It can not be collection type, like `Array`, `List`, etc, since it is hard to define `partition` function on collections

3. *Action operation*, denoted as `action()` is called on final `RDD`, then each partition produces a result

4. These results will be sent to the driver, then `f(List[Result])` will be computed as the final result to client side, for example, `count()` takes two steps, `action()` and `sum()`.

> RDD can be cached into memory or on hard disk, by calling `cache()`, `persist()` or `checkpoint()`. The number of partitions is usually set by user. Partition relationship between 2 RDDs can be not 1 to 1. In the picture above, we can see not only 1 to 1 relationship, but also many to many ones.

Logical Plan

When writing your spark code, you might also have a dependency diagram in you mind (like the one above). However, the reality is that some more `RDD`s will be produced.

In order to make this more clear, we will talk about :

- How to produce RDD ? What kind of RDD should be produced ?
- How to build dependency relationship between RDDs ?

1. How to produce RDD? What RDD should be produced?

A `transformation()` usually returns a new `RDD`, but some `transformation()`s which are more complicated and contain several sub-`transformation()` produce multiple `RDD`s. That's why the number of RDDs is, in fact, more than we thought.

Logical plan is essentially a *computing chain*. Every `RDD` has a `compute()` method which takes the input records of the previous `RDD` or data source, then performs `transformation()`, finally outputs computed records.

What `RDD` to be produced depends on the computing logic. Let's talk about some typical transformation() and the RDDs they produce.

We can learn about the meaning of each `transformation()` on Spark site. More details are listed in the following table, where `iterator(split)` means *for each record in partition*. There are some blanks in the table, because they are complex `transformation()` producing multiple RDDs, they will be illustrated soon after.

Transformation	Generated RDDs	Compute()
map(func)	MappedRDD	iterator(split).map(f)
filter(func)	FilteredRDD	iterator(split).filter(f)
flatMap(func)	FlatMappedRDD	iterator(split).flatMap(f)
mapPartitions(func)	MapPartitionsRDD	f(iterator(split))
mapPartitionsWithIndex(func)	MapPartitionsRDD	f(split.index, iterator(split))
sample(withReplacement, fraction, seed)	PartitionwiseSampledRDD	PoissonSampler.sample(iterator(split)) BernoulliSampler.sample(iterator(split))
pipe(command, [envVars])	PipedRDD	

union(otherDataset)		
intersection(otherDataset)		
distinct([numTasks]))		
groupByKey([numTasks])		
reduceByKey(func, [numTasks])		
sortByKey([ascending], [numTasks])		
join(otherDataset, [numTasks])		
cogroup(otherDataset, [numTasks])		
cartesian(otherDataset)		
coalesce(numPartitions)		
repartition(numPartitions)		

2. How to build RDD relationship?

We need to figure out the following things:

- RDD dependencies. `RDD x` depends on one parent RDD or several parent RDDs?
- How many partitions are there in `RDD x`?
- What's the relationship between the partitions of `RDD x` and those of its parent RDD(s)? One partition depends one or several partition of parent RDD?

The first question is trivial, such as `x = rdda.transformation(rddb)`, e.g., `val x = a.join(b)` means that `RDD x` depends both `RDD a` and `RDD b`

For the second question, as mentioned before, the number of partitions is defined by user, by default, it takes `max(numPartitions[parent RDD 1], ..., numPartitions[parent RDD n])`

The third one is a little bit complex, we need to consider the meaning of a `transformation()`. Different `transformation()`s have different dependency. For example, `map()` is 1:1, while `groupByKey()` produces a `ShuffledRDD` in which each partition depends on all partitions in its parent RDD. Besides this, some `transformation()` can be more complex.

In spark, there are 2 kinds of dependencies which are defined in terms of parent RDD's partition:

- NarrowDependency (OneToOneDependency, RangeDependency)

  > each partition of the child RDD depends on a small number of partitions of the parent RDD, i.e. a child partition depends the **entire** parent partition. (**full dependency**)

- ShuffleDependency (or wide dependency, mentioned in Matei's paper)

  > multiple child partitions depends on a parent partition, i.e. each child partition depends **a part of** the parent partition. (**partial dependency**)

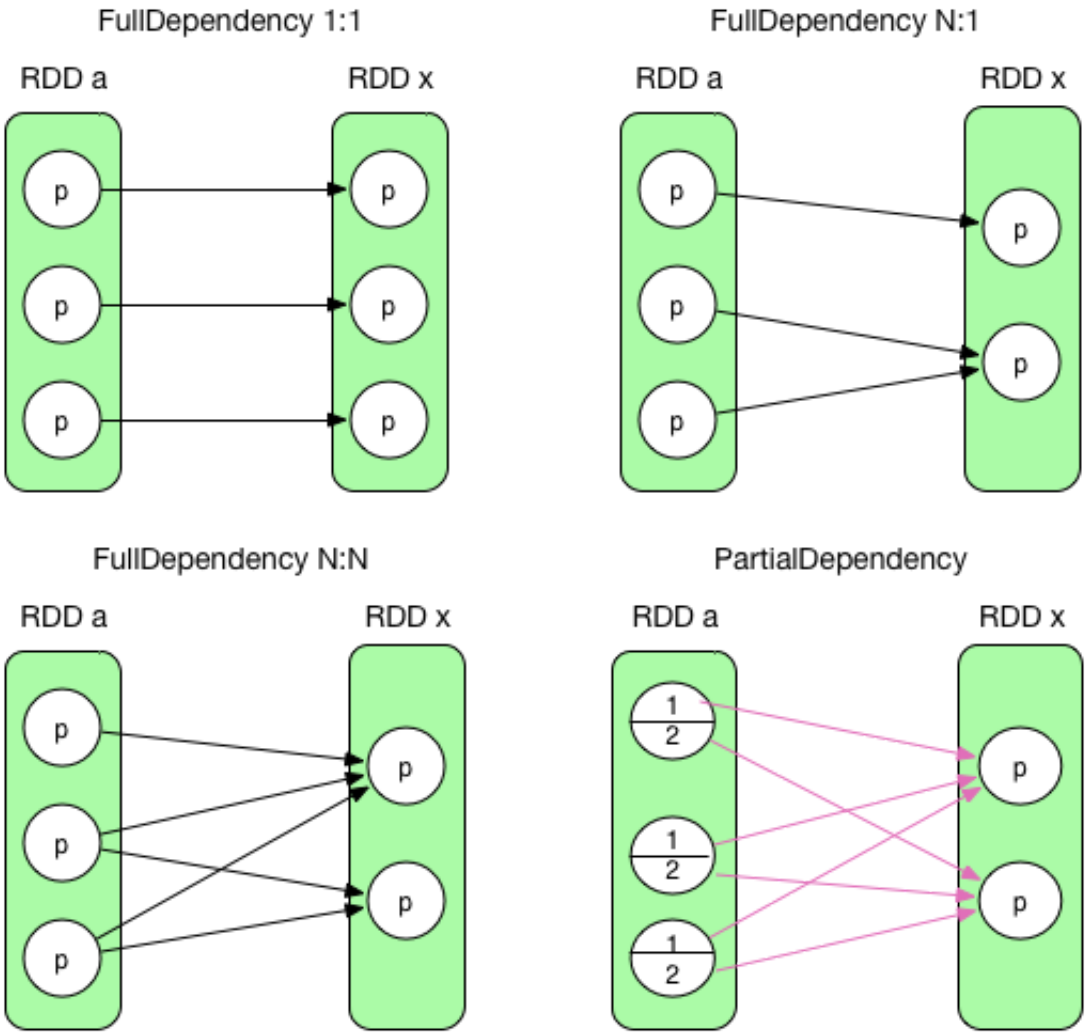
For example, `map` leads to a narrow dependency, while `join` leads to to wide dependencies (unless the two parents are hash-partitioned).

On the other hand, each child partition can depend on one partition in parent RDD, or some partitions in parent RDD.

Note that:

- For `NarrowDependency`, whether a child partition needs one or multiple parent partition depends on `getParents(partition i)` function in child RDD. (More details later)
- ShuffleDependency is like shuffle dependency in MapReduce（mapper partitions its output, then each reducer will fetch all the needed output partitions via http.fetch)

The two dependencies are illustrated in the following picture.



According to the definition, the two on the first row are `NarrowDependency` and the last one is `ShuffleDependency`.

Need to mention that the left one on the second row is a very rare case between two RDD. It is a `NarrowDependency` (N:N) whose logical plan is like ShuffleDependency, but it is a full dependency. It can be created by some tricks. We will not talk about this, because, more strictly, `NarrowDependency` essentially means **each partition of the parent RDD is used by at most one partition of the child RDD**. Some typical RDD dependencies will be talked about soon.

To conclude, partition dependencies are listed as below

- NarrowDependency (black arrow)
  - RangeDependency -> only for UnionRDD
  - OneToOneDependency (1:1) -> e.g. map, filter
  - NarrowDependency (N:1) -> e.g. join co-partitioned
  - NarrowDependency (N:N) -> rare case
- ShuffleDependency (red arrow)

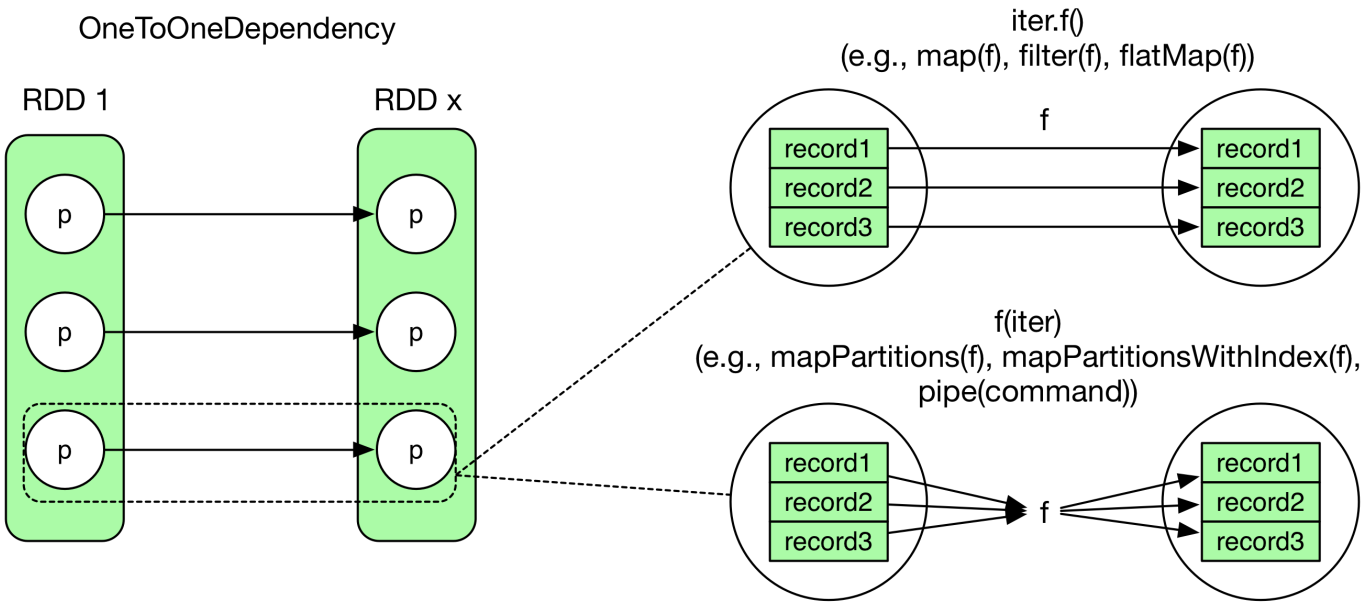
Note that, in the rest of this chapter, `NarrowDependency` will be represented by black arrow and `ShuffleDependency` are red ones.

`NarrowDependency` and `ShuffleDependency` are needed for physical plan which will be talked about in the next chapter.

How to compute records in RDD x

An `OneToOneDependency` case is shown in the picture below. Although it is a 1 to 1 relationship between two partitions, it doesn't mean that records are read and computed one by one.

The difference between the two patterns on the right side is similar to the following code snippets.



code1 of iter.f()

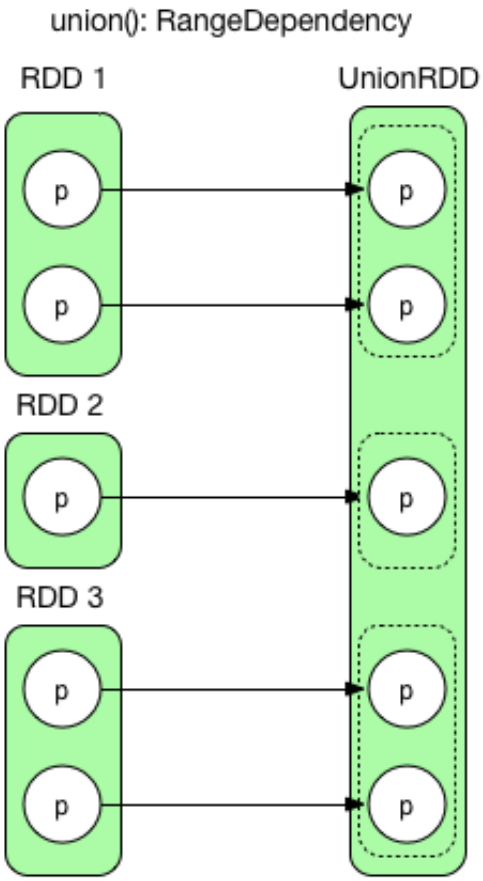
```
int[] array = {1, 2, 3, 4, 5}
for(int i = 0; i < array.length; i++)
    f(array[i])
```

code2 of f(iter)

```
int[] array = {1, 2, 3, 4, 5}
f(array)
```

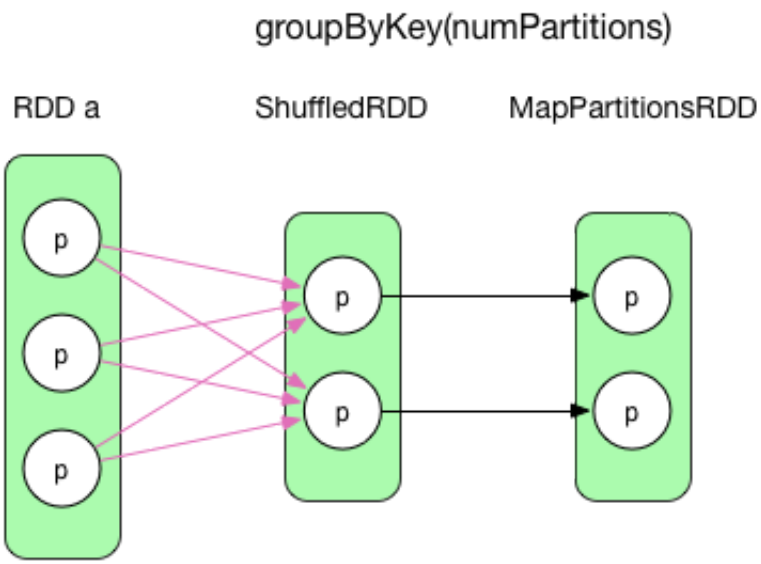
3. Illustration of typical dependencies and their computation

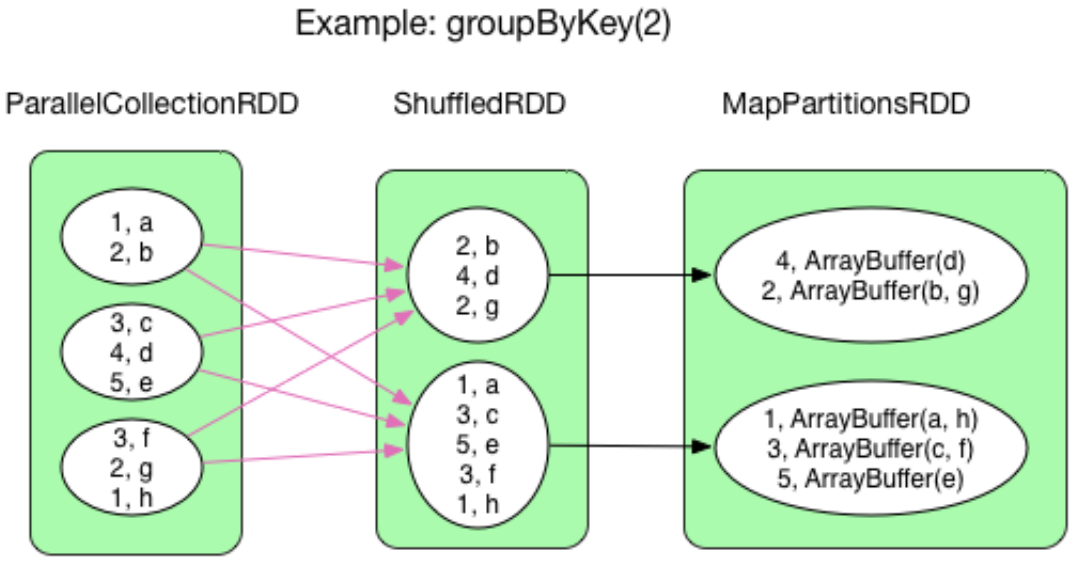
1) union(otherRDD)



`union()` simply combines two RDDs together. It never changes data of a partition. `RangeDependency`(1:1) retains the borders of original RDDs in order to make it easy to revisit the partitions from RDD produced by `union()`

2) groupByKey(numPartitions) [changed in 1.3]



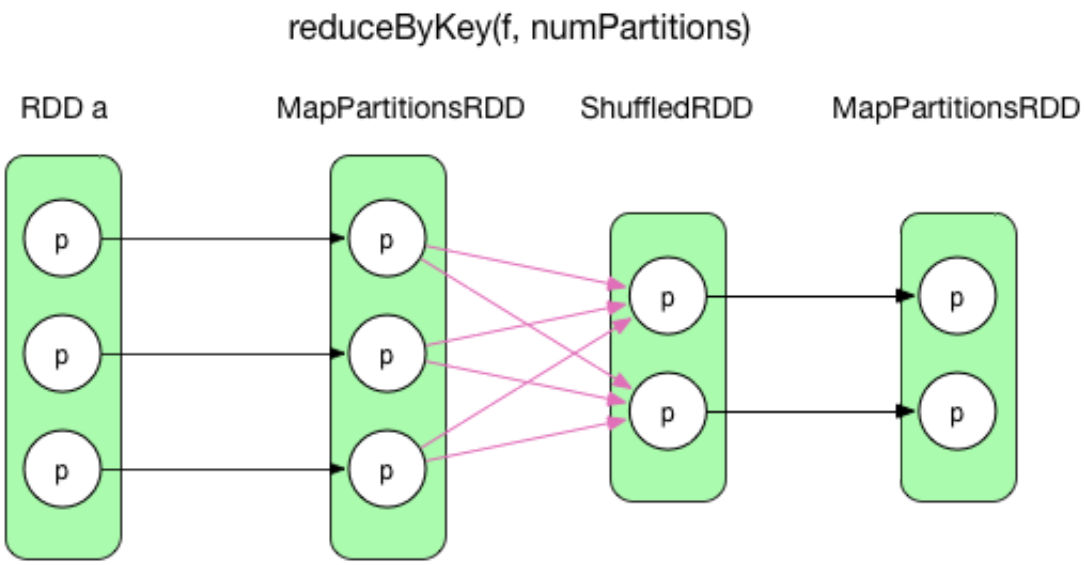


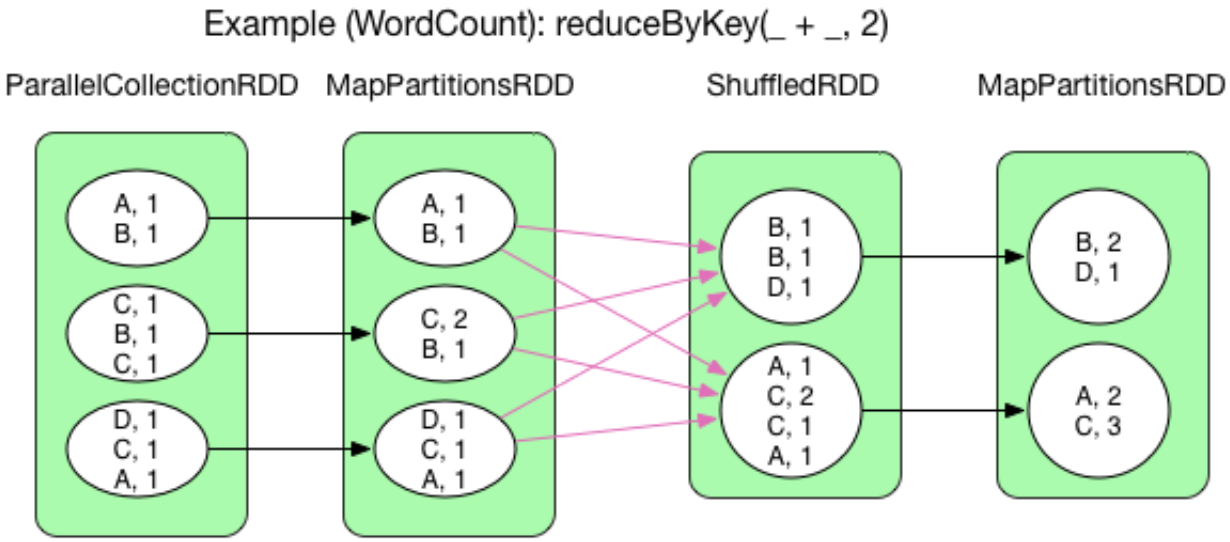
We have talked about `groupByKey`'s dependency before, now we make it more clear.

`groupByKey()` combines records with the same key by shuffle. The `compute()` function in `ShuffledRDD` fetches necessary data for its partitions, then take `mapPartition()` operation (like `OneToOneDependency`), `MapPartitionsRDD` will be produced by `aggregate()`. Finally, `ArrayBuffer` type in the value is casted to `Iterable`

> `groupByKey()` has no map side combine, because map side combine does not reduce the amount of data shuffled and requires all map side data be inserted into a hash table, leading to more objects in the old gen.
>
> `ArrayBuffer` is essentially `a CompactBuffer` which is an append-only buffer similar to ArrayBuffer, but more memory-efficient for small buffers.

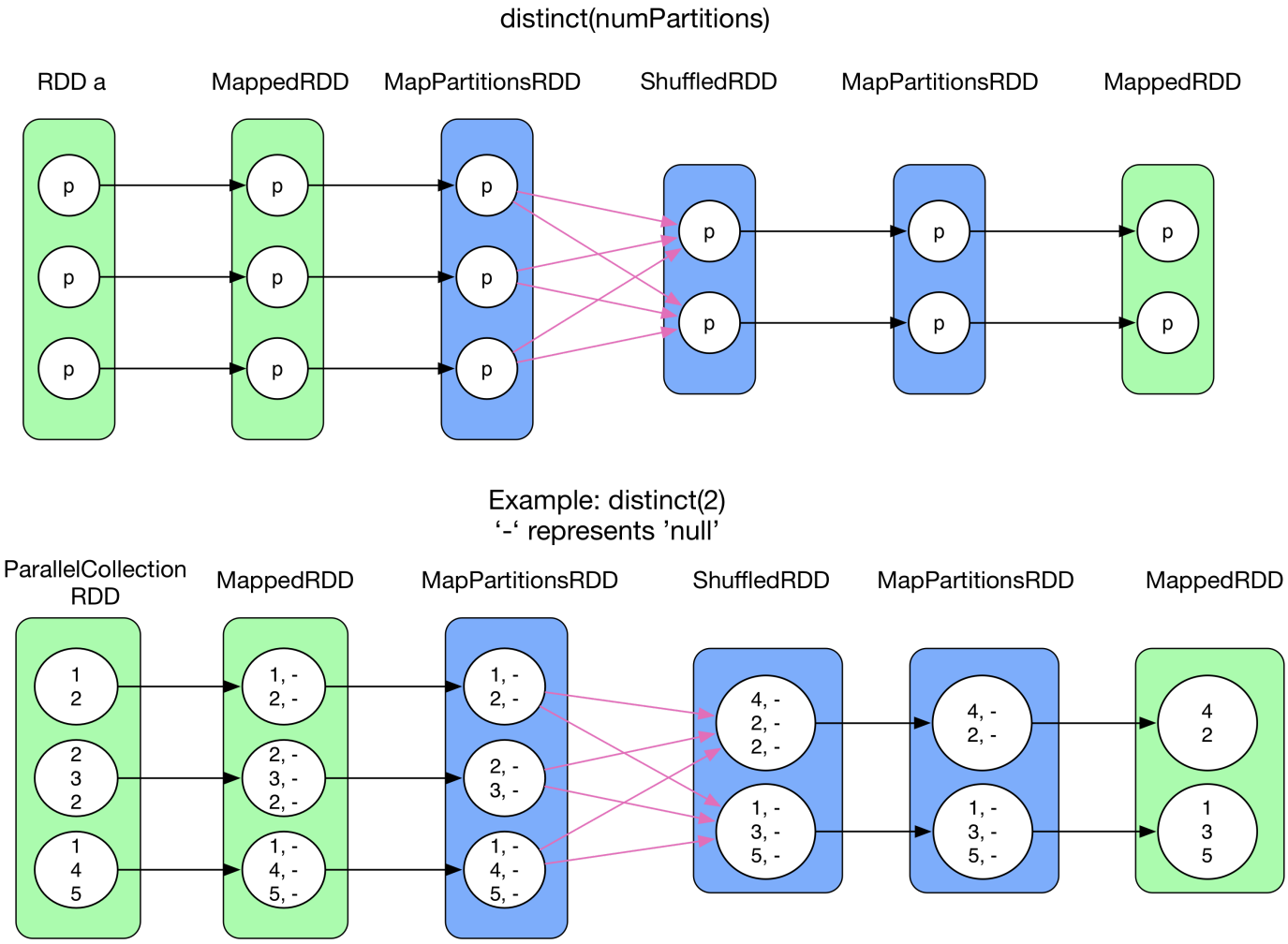
2) reduceyByKey(func, numPartitions) [changed in 1.3]





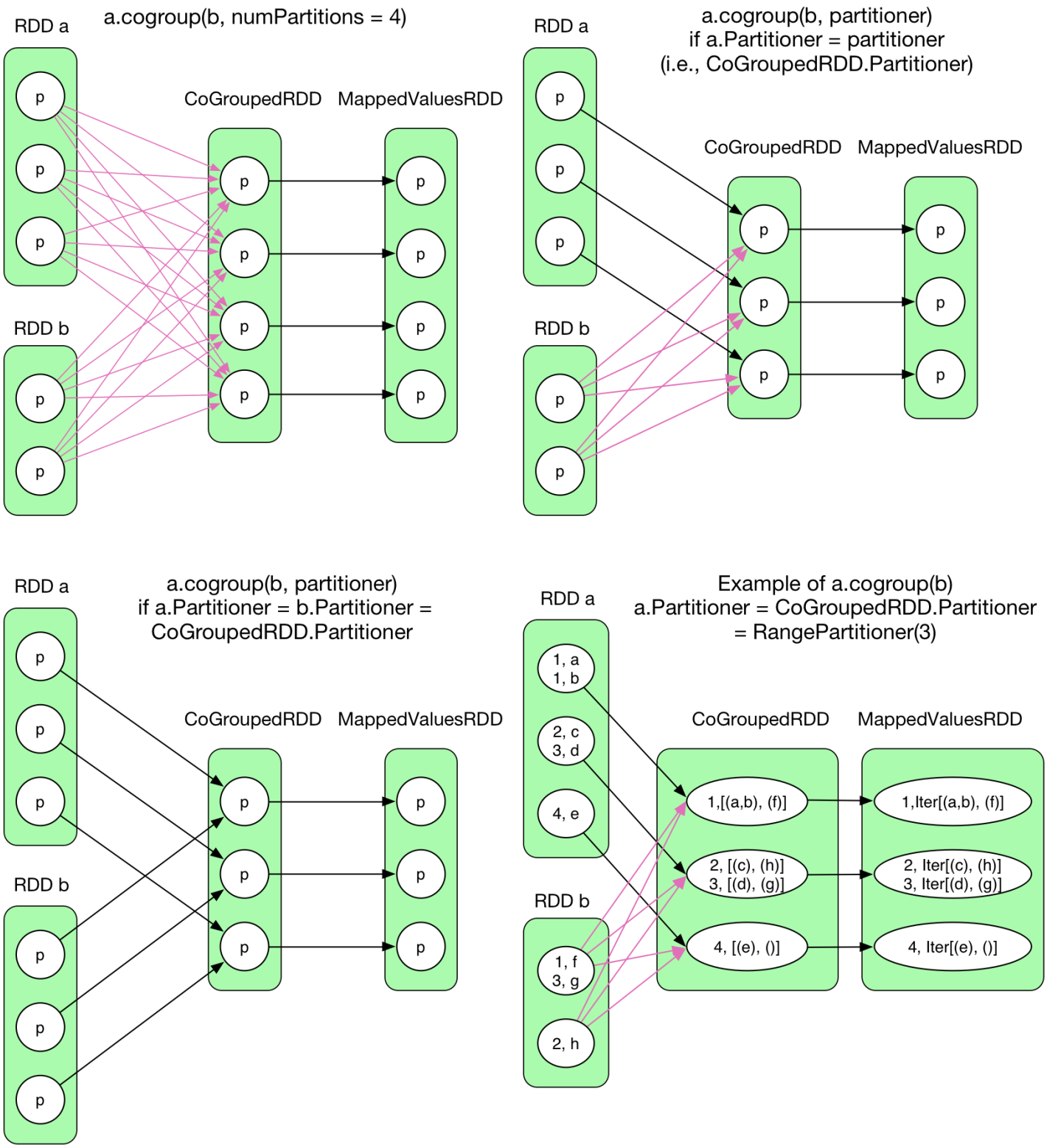
`reduceByKey()` is similar to `MapReduce`. The data flow is equivalent. `redcuceByKey` enables map side combine by default, which is carried out by `mapPartitions` before shuffle and results in `MapPartitionsRDD`. After shuffle, `aggregate + mapPartitions` is applied to `ShuffledRDD`. Again, we get a `MapPartitionsRDD`

3) distinct(numPartitions)



`distinct()` aims to deduplicate RDD records. Since duplicated records can be found in different partitions, shuffle is needed to deduplicate records by using `aggregate()`. However, shuffle need `RDD[(K, V)]`. If the original records have only keys, e.g. `RDD[Int]`, then it should be completed as `<K, null>` by `map()` (`MappedRDD`). After that, `reduceByKey()` is used to do some shuffle (mapSideCombine->reduce->MapPartitionsRDD). Finally, only key is taken from by `map()`(`MappedRDD`). `ReduceByKey()` RDDs are colored in blue

4) cogroup(otherRDD, numPartitions)



Different from `groupByKey()`, `cogroup()` aggregates 2 or more RDDs. **What's the relationship between GoGroupedRDD and (RDD a, RDD b)? ShuffleDependency or OneToOneDependency**?

- number of partition

  The # of partition in `CoGroupedRDD` is defined by user, it has nothing to do with `RDD a` and `RDD b`. However, if #partition of `CoGroupedRDD` is different from the one of `RDD a/b`, then it is not an `OneToOneDependency`.

- type of partitioner

  The `partitioner` defined by user (`HashPartitioner` by default) for `cogroup()` decides where to put the its results. Even if `RDD a/b` and `CoGroupedRDD` have the same # of partition, while their partitioner are different, it can not be `OneToOneDependency`. Let's take the examples in the picture above, `RDD a` is `RangePartitioner`, `RDD b` is `HashPartitioner`, and `CoGroupedRDD` is `RangePartitioner` with the same # partition as `RDD a`. Obviously, the records in each partition of `RDD a` can be directly sent to the corresponding partitions in `CoGroupedRDD`, but those in `RDD b` need to be divided in order to be shuffled into the right partitions of `CoGroupedRDD`.

To conclude, `OneToOneDependency` occurs iff the partitioner type and #partitions of 2 RDDs and `CoGroupedRDD` are the same, otherwise, the dependency must be a `ShuffleDependency`. More details can be found in `CoGroupedRDD.getDependencies()`'s source code

How does spark deal with the fact that `CoGroupedRDD`'s partition depends on multiple parent partitions?
Firstly, `CoGroupedRDD` put all needed `RDD` into `rdds: Array[RDD]`

Then,

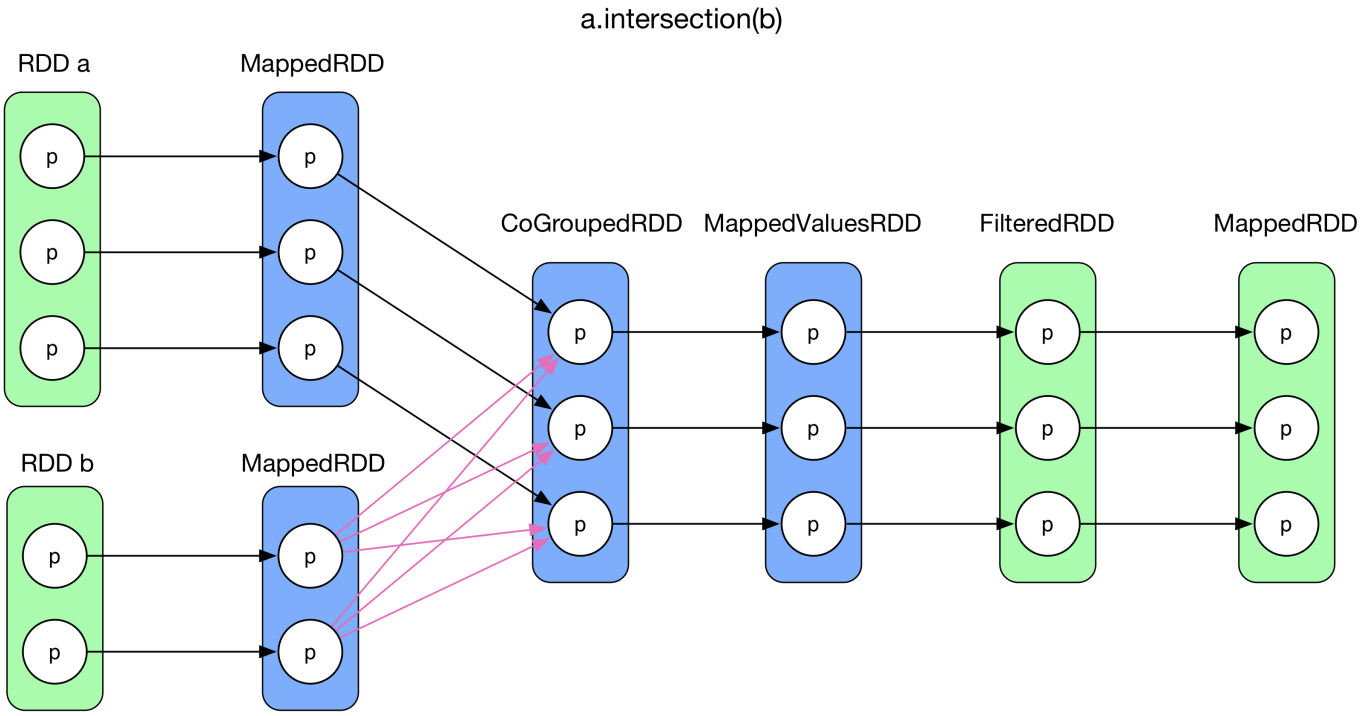
```
Foreach rdd = rdds(i):
    if CoGroupedRDD and rdds(i) are OneToOneDependency
        Dependecy[i] = new OneToOneDependency(rdd)
    else
        Dependecy[i] = new ShuffleDependency(rdd)
```

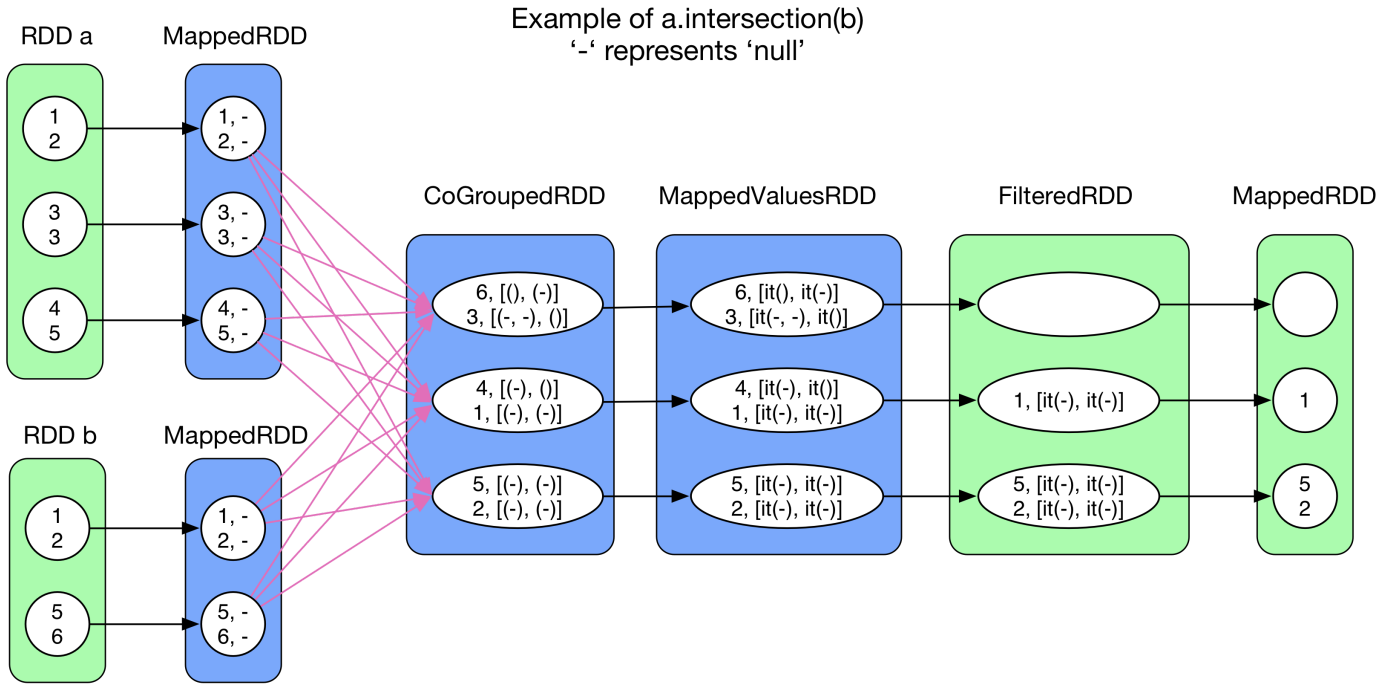
Finally, it returns `deps: Array[Dependency]` which is an array of `Dependency` corresponding to each parent RDD.

`Dependency.getParents(partition id)` returns `partitions: List[Int]` which are the necessary parent partitions of the specified partition (`partition id`) with respect to the given `Dependency`

`getPartitions()` tells how many partitions are in `RDD` and how each partition is serialized.

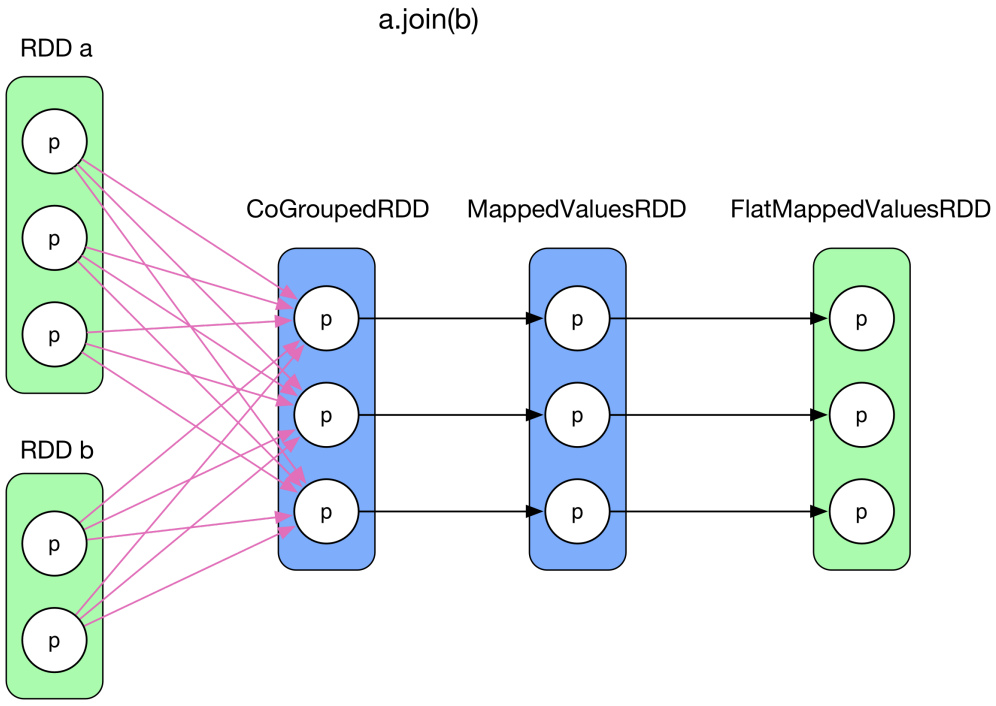
5) intersection(otherRDD)

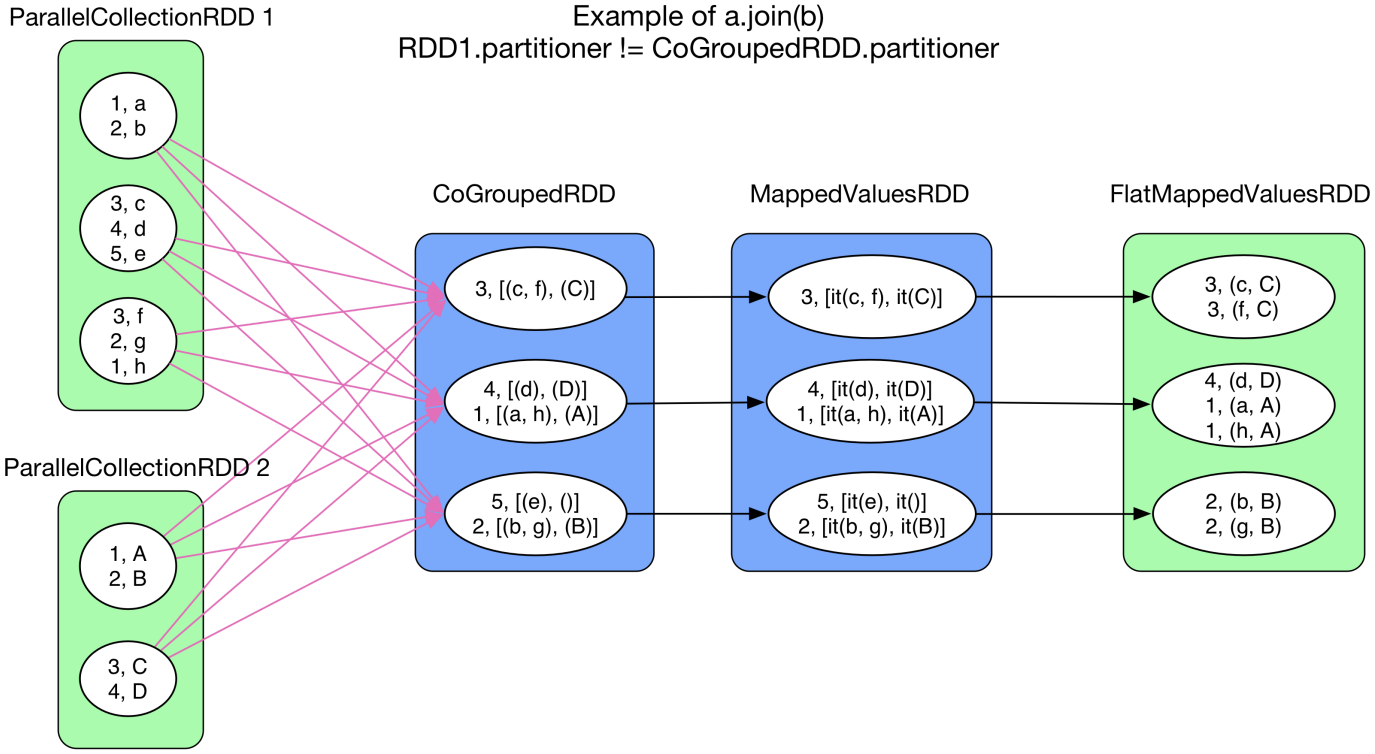


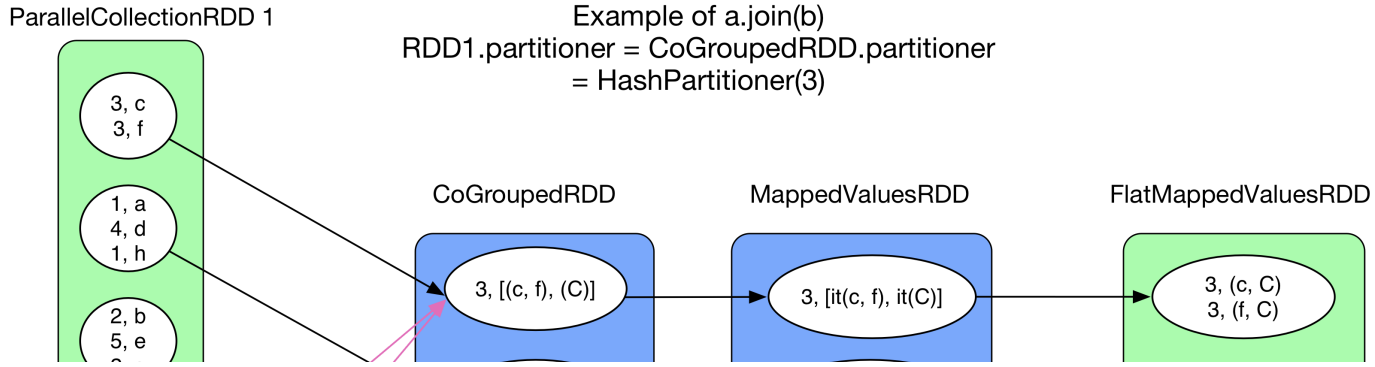


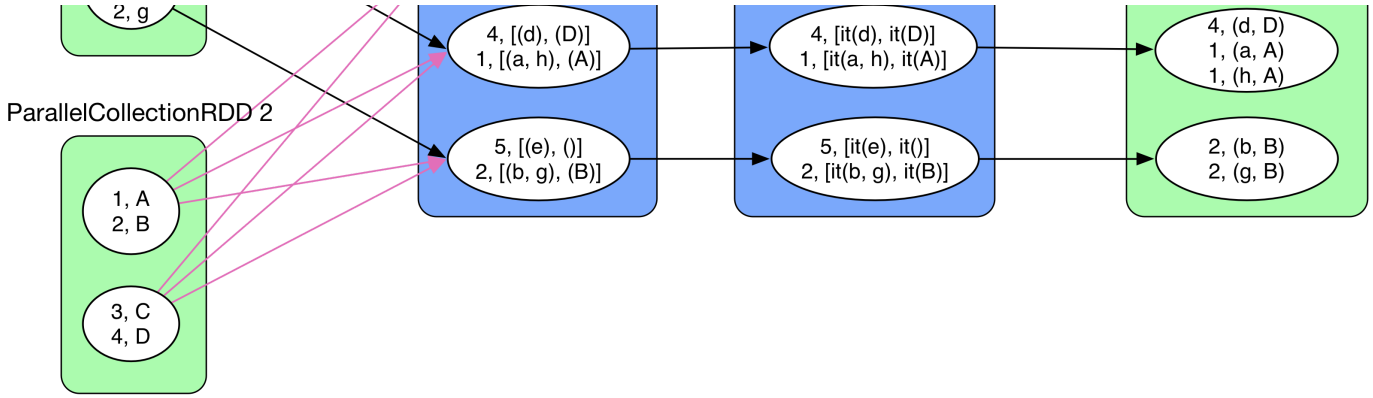
`intersection()` aims to extract all the common elements from `RDD a and b`. `RDD[T]` is mapped into `RDD[(T, null)]`, where `T` can not be any collections, then `a.cogroup(b)` (colored in blue). `filter()` only keeps records where neither of `[iter(groupA()), iter(groupB())]` is empty (`FilteredRDD`). Finally, only `keys()` are kept (`MappedRDD`)

6) **join(otherRDD, numPartitions)**


a.join(b)
RDD a
RDD b
CoGroupedRDD
MappedValuesRDD
FlatMappedValuesRDD
p


ParallelCollectionRDD 1
Example of a.join(b)
RDD1.partitioner != CoGroupedRDD.partitioner
1, a
2, b
3, c
4, d
5, e
3, f
2, g
1, h
ParallelCollectionRDD 2
1, A
2, B
3, C
4, D
CoGroupedRDD
MappedValuesRDD
FlatMappedValuesRDD
3, [(c, f), (C)]
4, [(d), (D)]
1, [(a, h), (A)]
5, [(e), ()]
2, [(b, g), (B)]
3, [it(c, f), it(C)]
4, [it(d), it(D)]
1, [it(a, h), it(A)]
5, [it(e), it()]
2, [it(b, g), it(B)]
3, (c, C)
3, (f, C)
4, (d, D)
1, (a, A)
1, (h, A)
2, (b, B)
2, (g, B)

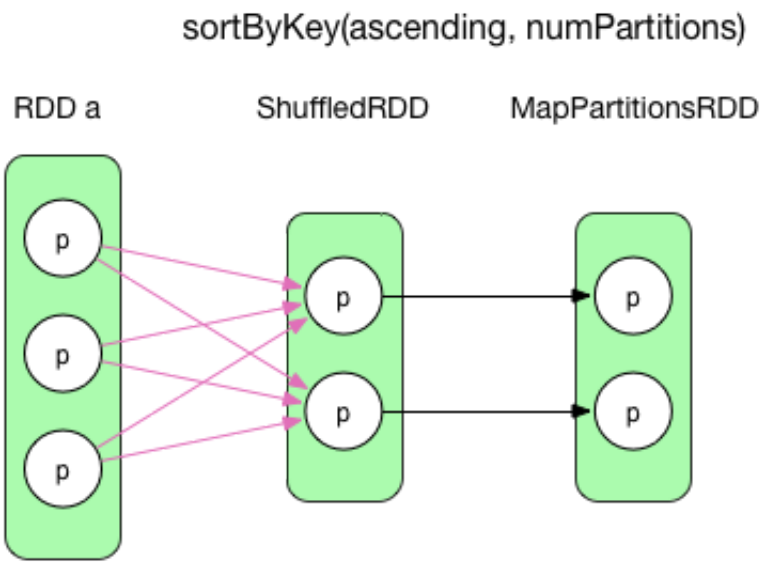

ParallelCollectionRDD 1
Example of a.join(b)
RDD1.partitioner = CoGroupedRDD.partitioner
= HashPartitioner(3)
3, c
3, f
1, a
4, d
1, h
2, b
5, e
CoGroupedRDD
MappedValuesRDD
FlatMappedValuesRDD
3, [(c, f), (C)]
3, [it(c, f), it(C)]
3, (c, C)
3, (f, C)

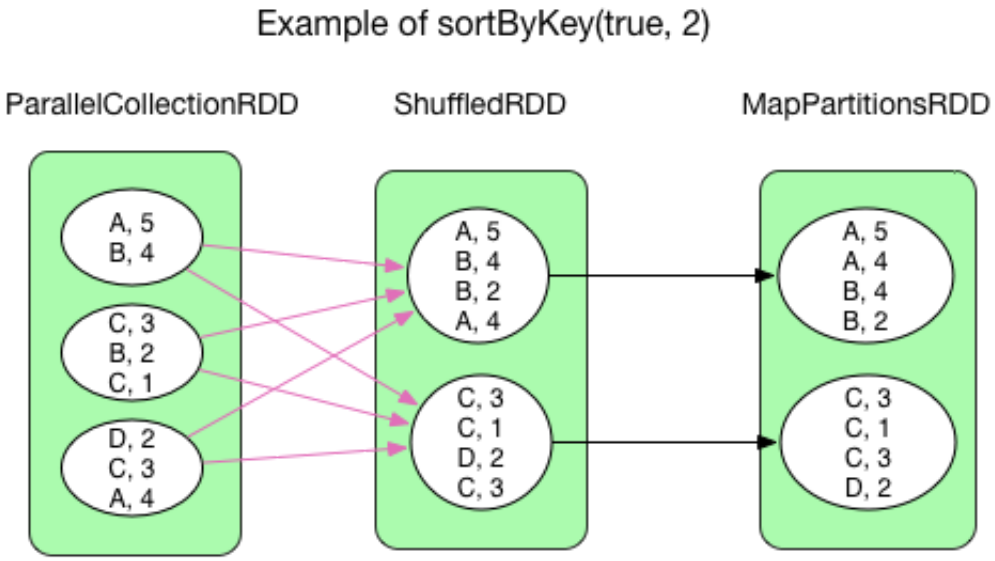


`join()` takes two `RDD[(K, V)]`, like `join` in SQL. Similar to `intersection()`, it does `cogroup()` first and results in a `MappedValuesRDD` whose type is `RDD[(K, (Iterable[V1], Iterable[V2]))]`, then compute the Cartesian product between the two `Iterable`, finally `flatMap()` is called.

Here are two examples, in the first one, `RDD 1` and `RDD 2` use `RangePartitioner`, `CoGroupedRDD` takes `HashPartitioner` which is different from `RDD 1/2`, so it's a `ShuffleDependency`. In the second one, `RDD 1` is initially partitioned on key by `HashPartitioner` and gets 3 partition which is the same as the one `CoGroupedRDD` takes, so it's a `OneToOneDependency`. Furthermore, if `RDD 2` is also initially divided by `HashPartitioner(3)`, then there is no `ShuffleDependency`. This kind of `join` is called `hashjoin()`

7) sortByKey(ascending, numPartitions)

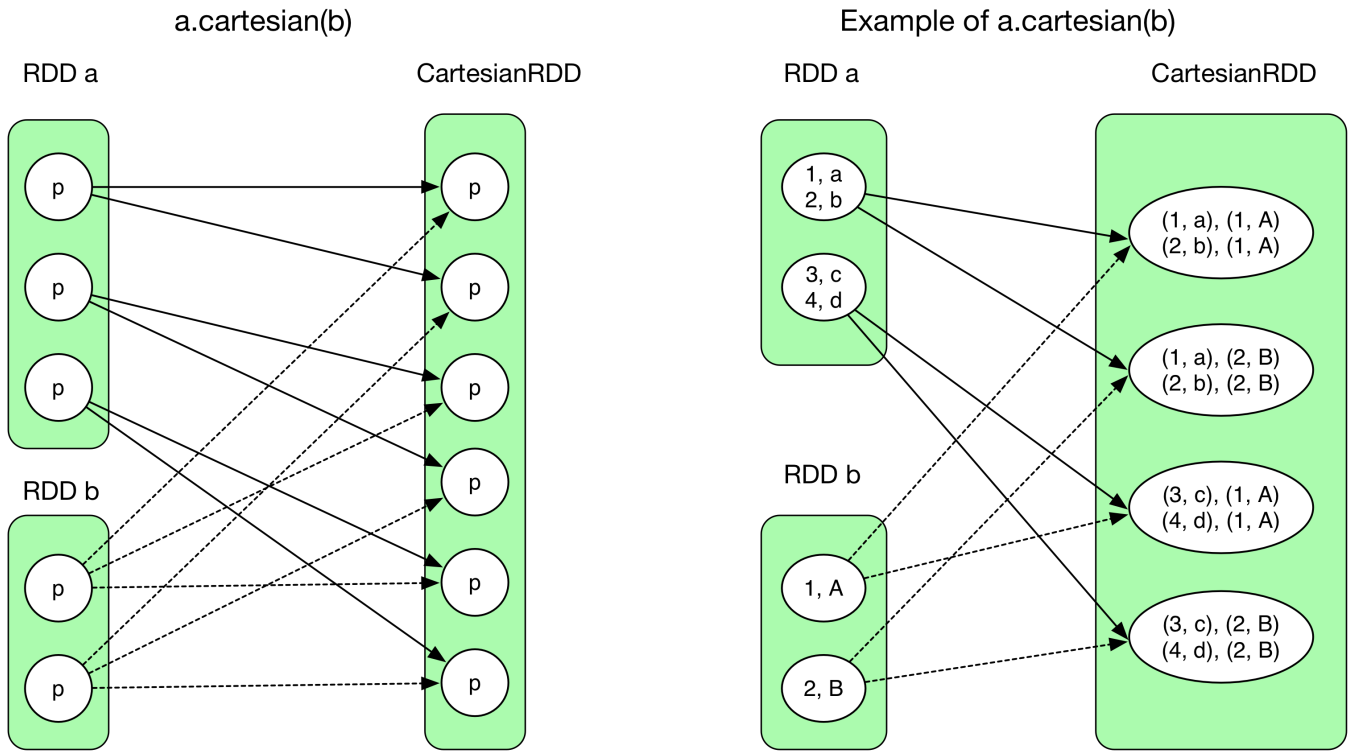




`sortByKey()` sorts records of `RDD[(K, V)]` by key. `ascending` is a self-explanatory boolean flag. It produces a `ShuffledRDD` which takes a `rangePartitioner`. The partitioner decides the border of each partition, e.g. the first partition takes records with keys from `char A` to `char B`, and the second takes those from `char C` to `char D`. Inside each partition, records are sorted by key. Finally, the records in `MapPartitionsRDD` are in order.

> `sortByKey()` use `Array` to store records of each partition, then sorts them.

8) cartesian(otherRDD)



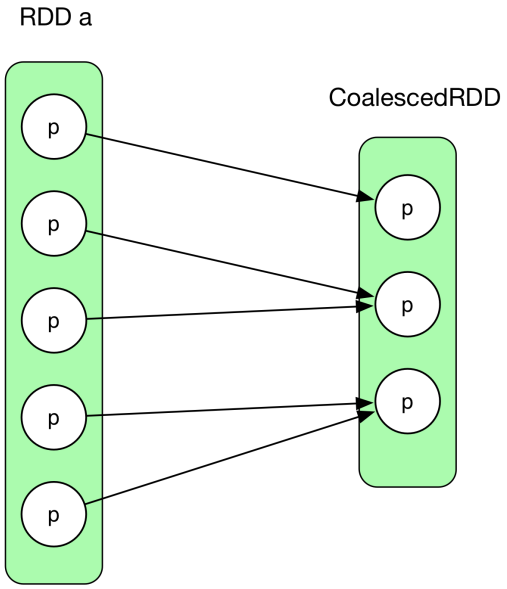
`Cartesian()` returns a Cartesian product of 2 `RDD`s. The resulting `RDD` has `#partition(RDD a) x #partition(RDD b)` partitions.

Need to pay attention to the dependency, each partition in `CartesianRDD` depends 2 **entire** parent RDDs. They are all `NarrowDependency`.

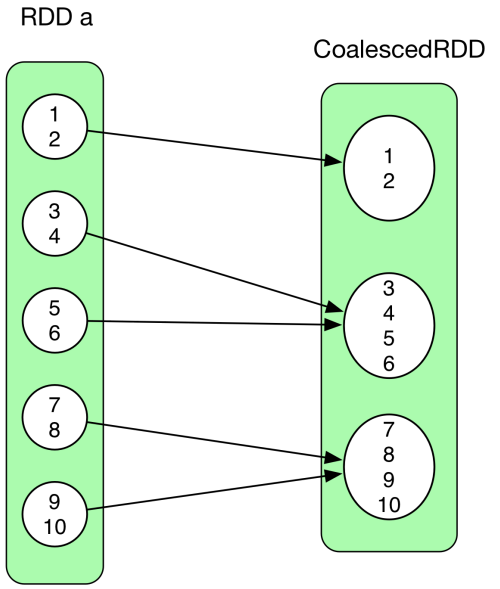
> `CartesianRDD.getDependencies()` returns `rdds: Array(RDD a, RDD b)`. The ith partition of `CartesianRDD` depends:
>
> - `a.partitions(i / #partitionA)`
> - `b.partitions(i % #partitionB)`

9) coalesce(numPartitions, shuffle = false)

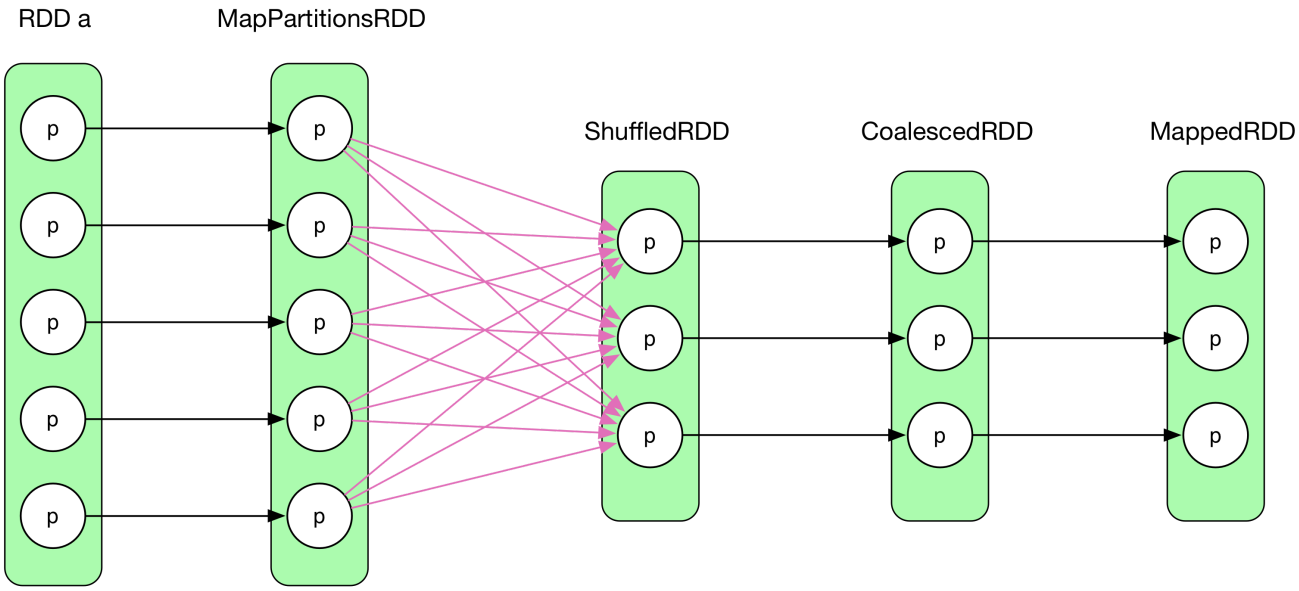
a.coalesce(numPartitions, shuffle = false)



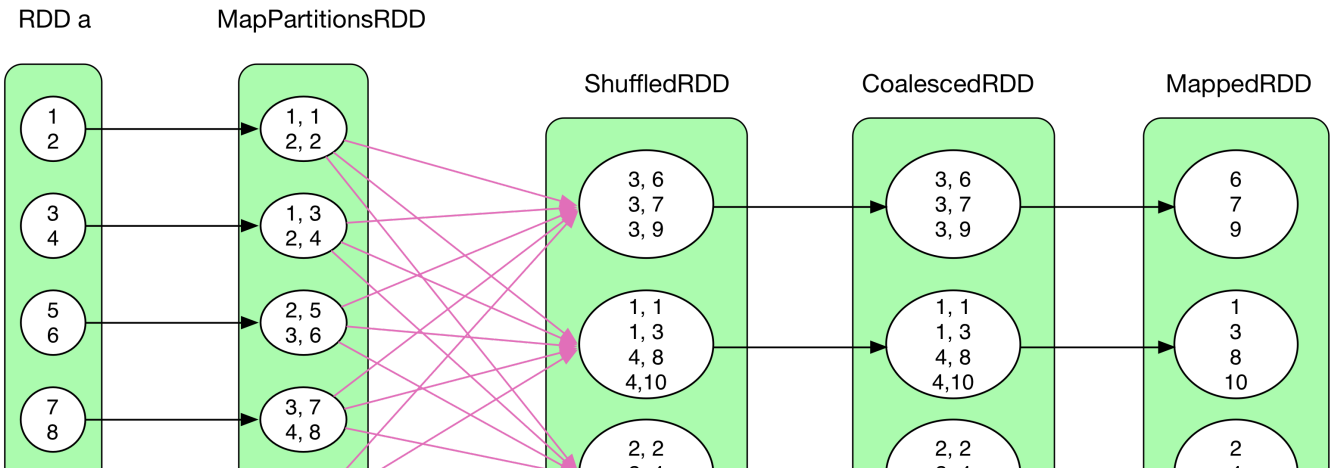
Example: a.coalesce(3, shuffle = false)

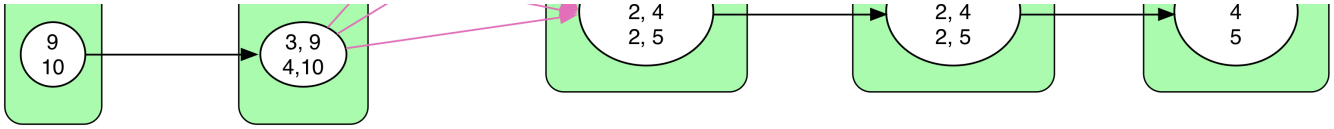


a.coalesce(numPartitions, shuffle = true)



Example: a.coalesce(3, shuffle = true)





`coalesce()` can reorganize partitions, e.g. decrease # of partitions from 5 to 3, or increase from 5 to 10. Need to notice that when `shuffle = false`, we can not increase partitions, because that will force a shuffle while we don't want shuffle, which is nonsense.

To understand `coalesce()`, we need to know **the relationship between `CoalescedRDD`'s partitions and its parent partitions**

- `coalesce(shuffle = false)` As shuffle is disabled, what we need to do is just to group certain parent partitions. In fact, there are many factors to take into consideration, e.g. # records in partition, locality ,balance, etc. Spark has a rather complicated algorithm to do with that. (we will not talk about that for the moment). For example, `a.coalesce(3, shuffle = false)` is essentially a `NarrowDependency` of N:1.

- `coalesce(shuffle = true)` When shuffle is enabled, `coalesce` simply divides all records of `RDD` in N parts, which can be done by the following trick (like round-robin algorithm):

  - for each partition, every record is assigned a key which is an increasing number.
  - hash(key) leads to a uniform records distribution on all different partitions.

  In the second example, every element in `RDD a` is combined with a increasing key (on the left side of the pair). The key of the first element in a partition is equal to `(new Random(index)).nextInt(numPartitions)`, where `index` is the index of the partition and `numPartitions` is the # of partitions in `CoalescedRDD`. The following keys increase by 1. After shuffle, the records in `ShffledRDD` are uniformly distributed. The relationship between `ShuffledRDD` and `CoalescedRDD` is defined a complicated algorithm. In the end, keys are removed (`MappedRDD`).

10) repartition(numPartitions)
equivalent to coalesce(numPartitions, shuffle = true)

Primitive transformation()

combineByKey()So far, we have seen a lot of logic plans. It's true that some of them are very similar. The reason lies in their implementation.
Knowing that the RDD on left side of `ShuffleDependency` should be `RDD[(K, V)]`, while, on the right side, all records with the same key are aggregated, then different operation will be applied on these aggregated records.

In fact, many `transformation()`, like `groupByKey()`, `reduceBykey()`, executes `aggregate()`

while doing logical computation. So **the similarity is that `aggregate()` and `compute()` are executed in the same time.** Spark uses `combineByKey()` to implement `aggregate() + compute()` operation.

Here is the definition of `combineByKey()`

```
def combineByKey[C](createCombiner: V => C,
     mergeValue: (C, V) => C,
     mergeCombiners: (C, C) => C,
     partitioner: Partitioner,
     mapSideCombine: Boolean = true,
     serializer: Serializer = null): RDD[(K, C)]
```

There are three important parameters to talk about:

- `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.

Details:

- When some (K, V) pair records are being pushed to `combineByKey()`, `createCombiner` takes the first record to initialize a combiner of type `C` (e.g. C = V).
- From then on, `mergeValue` takes every incoming record, `mergeValue(combiner, record.value)`, to update the combiner. Let's take `sum` as an example, `combiner = combiner + recoder.value`. In the end, all concerned records are merged into the combiner
- If there is another set of records with the same key as the pairs above. `combineByKey()` will produce another `combiner'`. In the last step, the final result is equal to `mergeCombiners(combiner, combiner')`.

Discussion

So far, we have discussed how to produce job's logical plan as well as the complex dependency and computation behind spark

`tranformation()` decides what kind of RDDs will be produced. Some `transformation()` are reused by other operations (e.g. `cogroup`)

The dependency of a `RDD` depends on how `transformation()` produces corresponding RDD. e.g. `CoGroupdRDD` depends on all `RDD`s used for `cogroup()`

The relationship of `RDD` partitions are `NarrowDependency` and `ShuffleDependency`. The former is **full dependency** and the latter is **partial dependency**. `NarrowDependency` can be represented in many

cases, a dependency is a `NarrowDependency` iff #partition and partitioner type are the same.

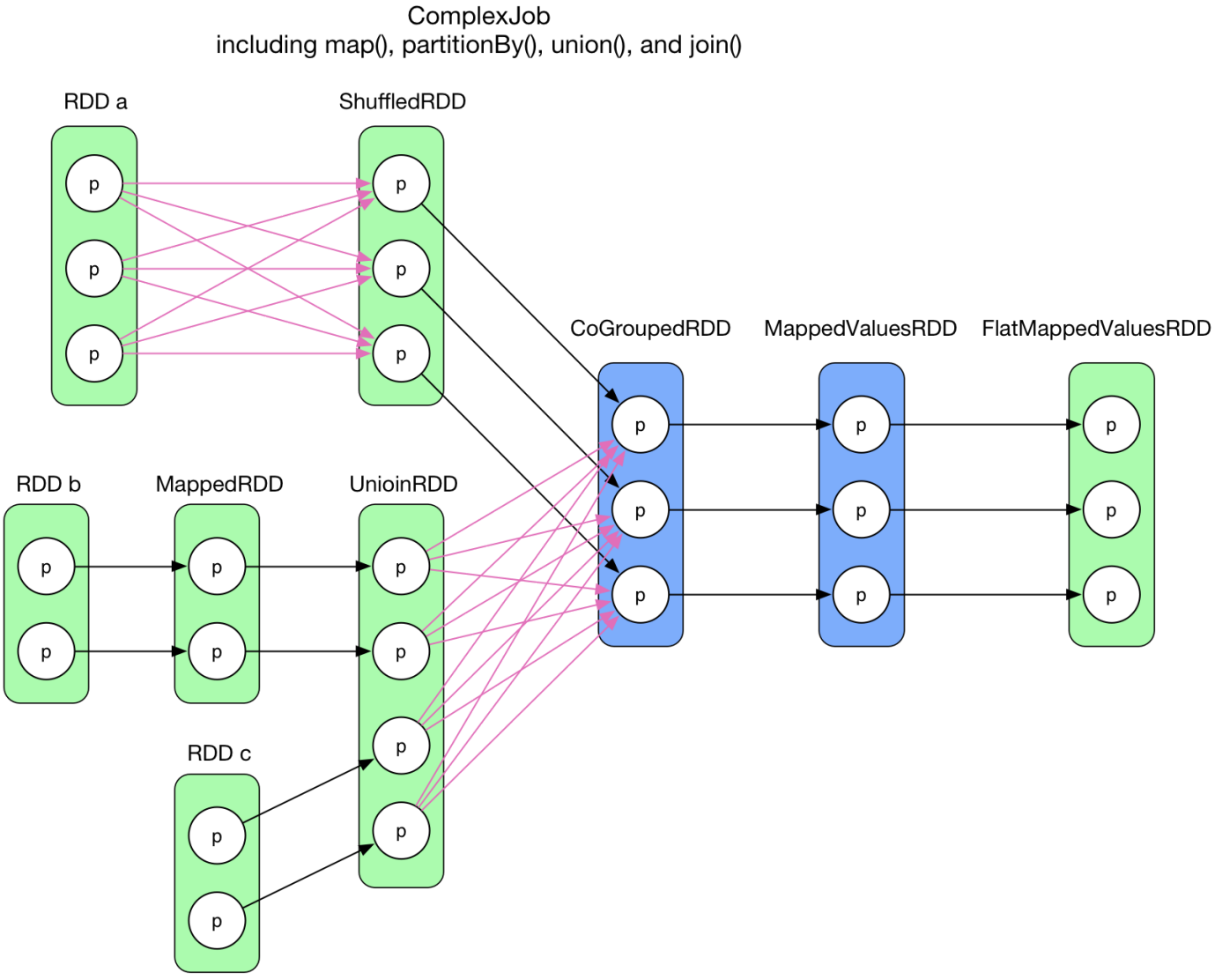
In terms of data flow, `MapReduce` is equivalent to `map() + reduceByKey()`. Technically, the `reduce` of `MapReduce` would be more powerful than `reduceByKey()`. These details will be talked about in chapter **Shuffle details**.

JerryLead/SparkInternals

Physical Plan

We have briefly introduced the DAG-like physical plan, which contains stages and tasks. In this chapter, we'll look at **how the physical plan (so the stages and tasks) is generated given a logical plan of a Spark application.**

A Complex Logical Plan

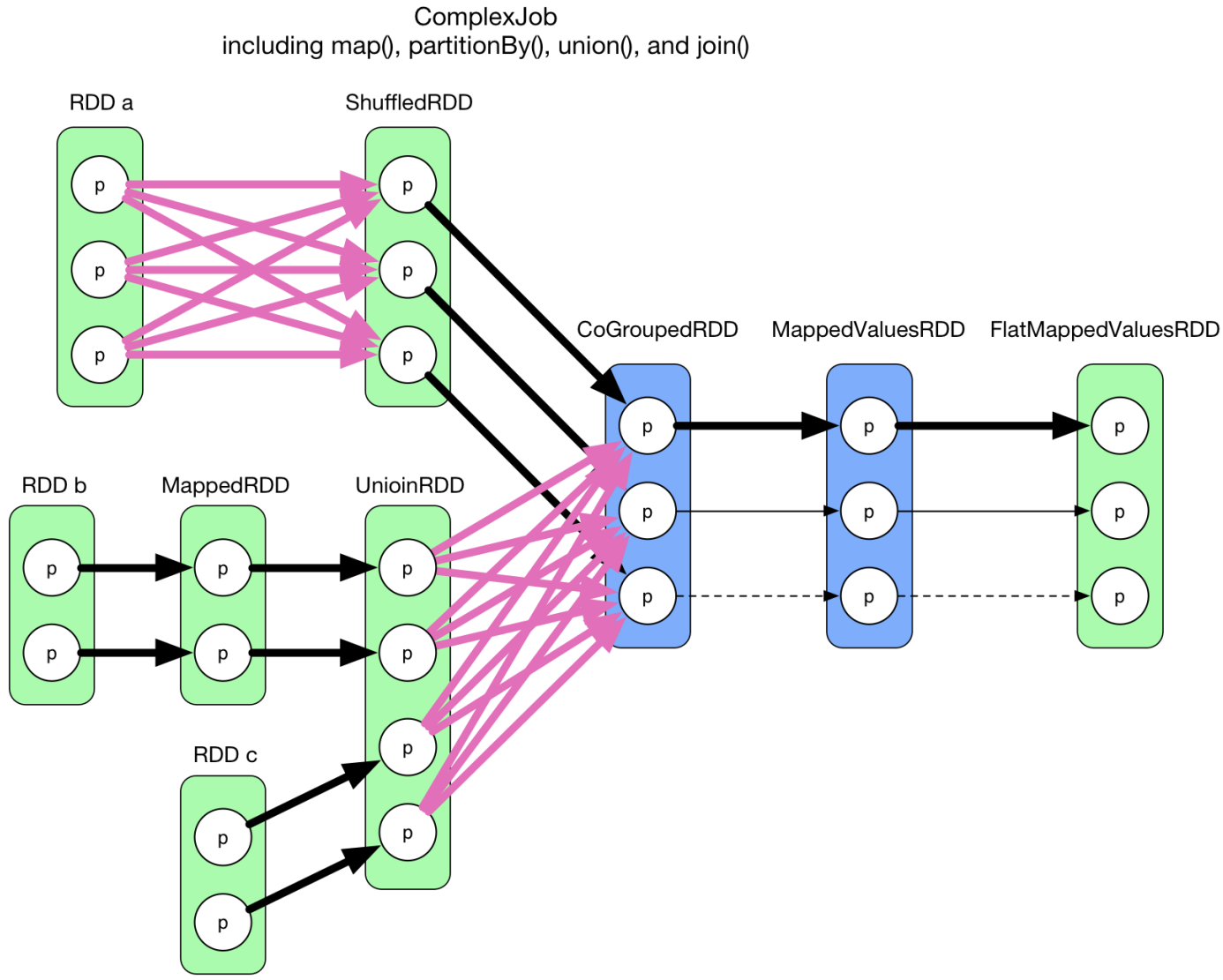


The code of this application is attached at the end of this chapter.

How to properly define stages and determine the tasks with such a complex data dependency graph?

An intuitive idea is to associate one RDD and its preceding RDD to form a stage, in this way each arrow in the above graph will become a task. For the case of 2 RDDs aggregates into one, we may create a stage with these 3 RDDs. This strategy could be a working solution, but will not be efficient. It has a subtle, but severe problem: **lots of intermediate data needs to be stored**. For a physical task, its result will be stored either on local disk, or in the memory, or both. If a task is generated for each arrow in the data dependency graph, the system needs to store data of all the RDDs. It will cost a lot.

If we examine the logical plan more closely, we may find out that in each RDD, the partitions are independent from each other. That is to say, inside each RDD, the data within a partition will not interfere others. With this observation, an aggressive idea is to consider the whole diagram as a single stage and create one physical task for each partition of the final RDD (`FlatMappedValuesRDD`). The following diagram illustrates this idea:



All thick arrows in above diagram belong to task1 whose result is the first partition of the final RDD of the job. Note that in order to compute the first partition of the `CoGroupedRDD`, we need to evaluate all partitions of its preceding RDDs since it's a `ShuffleDependency`. So in the computation of task1, we can also take the chance to compute the `CoGroupedRDD`'s second and third partition, for task2 and task3

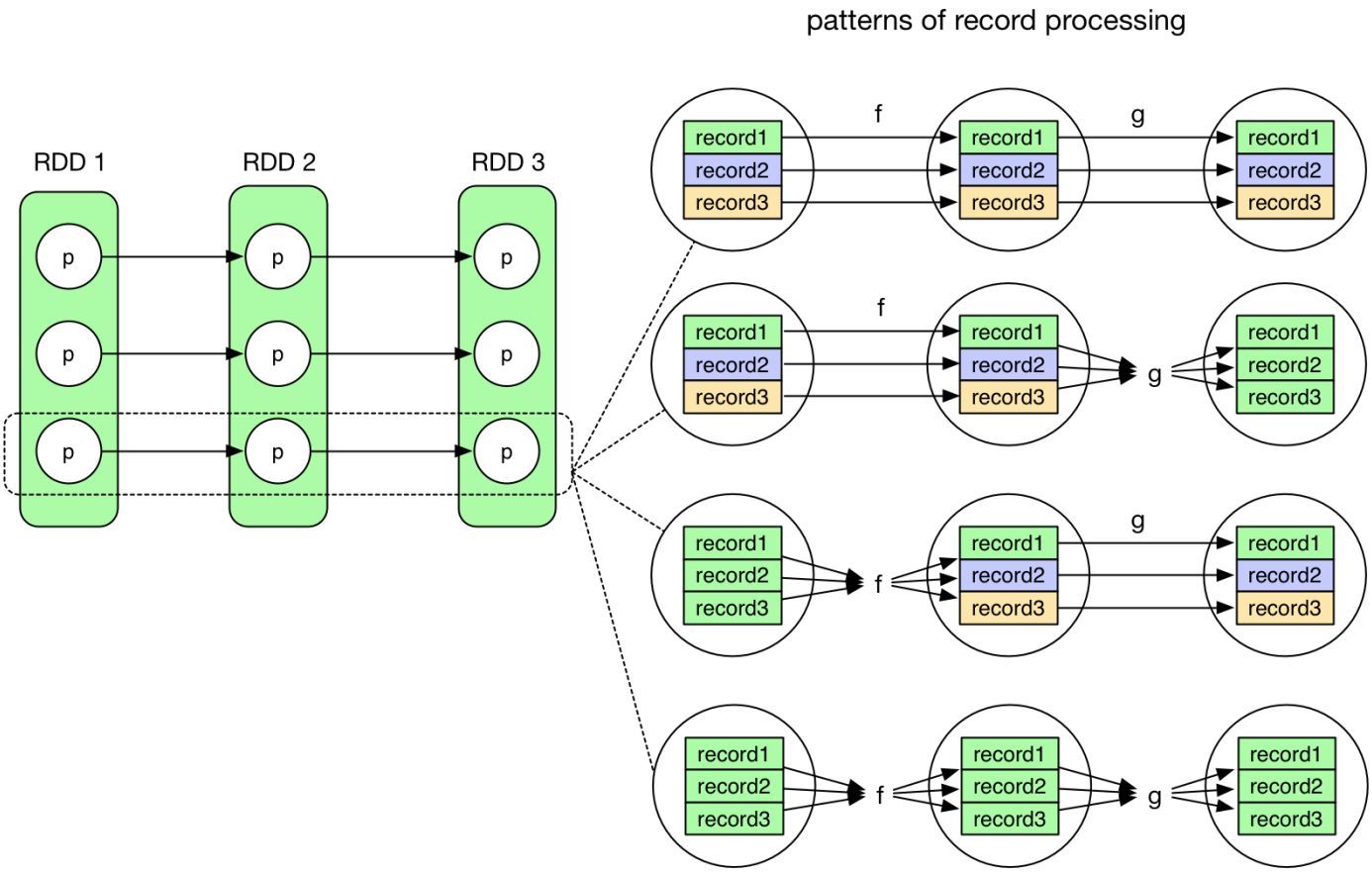
respecively. As a result, the task2 and task3 are simpler. They are represented by thin arrows and dashed arrows in the diagram.

However, there's 2 problems with this idea:

- The first task is big. Because of the `ShuffleDependency`, we have to evaluate all the partitions of the preceding RDDs.
- Need to design clever algorithms to determine which are the partitions to cache.

But there's also one good point in this idea, that is to **pipeline the data: the data is computed when they are actually needed in a flow fashion**. For example in the first task, we check backwards from the final RDD (`FlatMappedValuesRDD`) to see which are the RDDs and partitions that are actually needed to be evaluated. And between the RDDs with `NarrowDependency` relation, no intermediate data needs to be stored.

It will be clearer to understand the pipelining if we consider a record-level point of view. The following diagram illustrates different evaluation patterns for RDDs with `NarrowDependency`.



The first pattern (pipeline pattern) is equivalent to:

```
for (record <- records) {
  f(g(record))
```

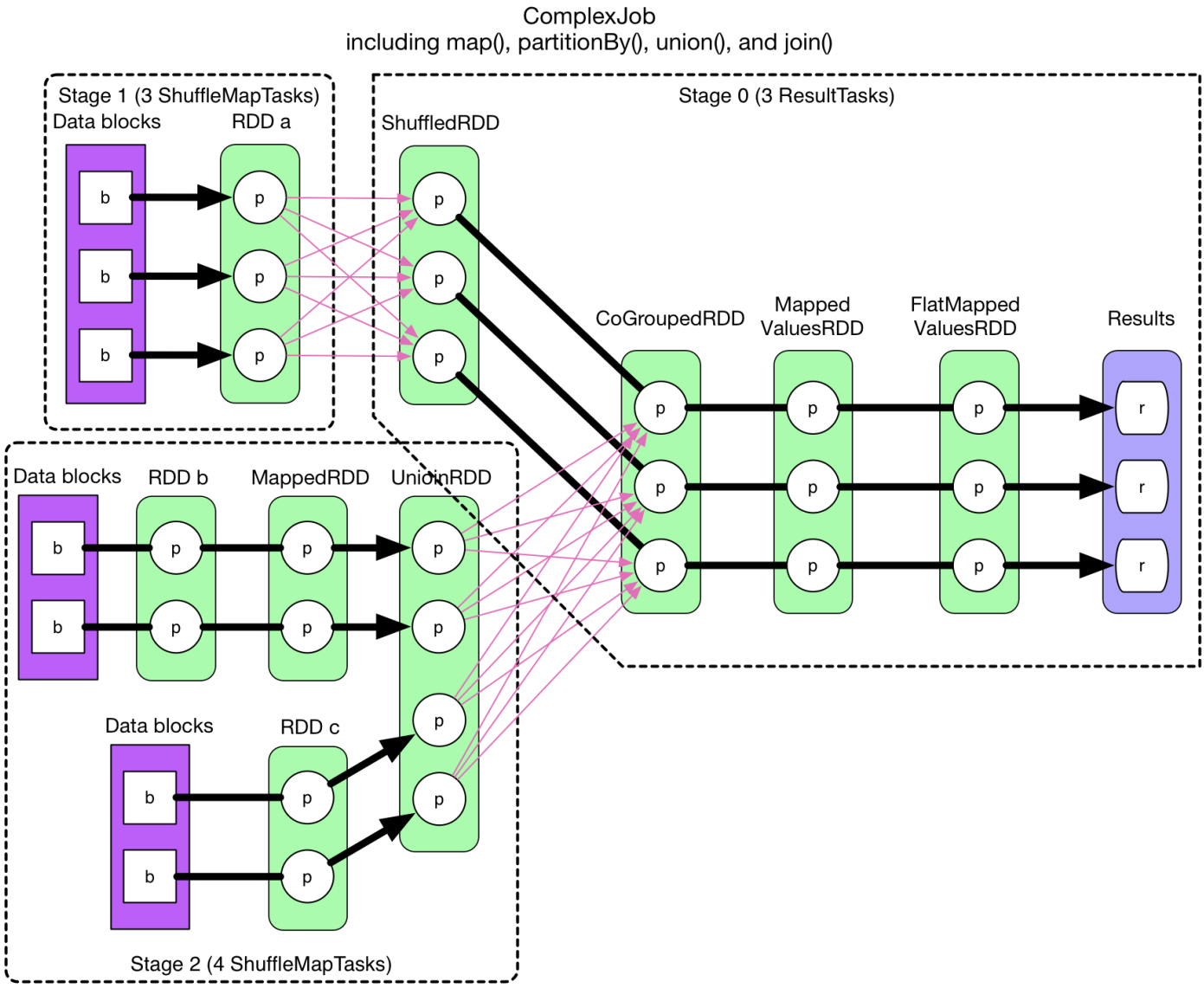
```
}
```

Consider `records` as a stream, we can see that no intermediate result need to be stored.Once the computation `f(g(record))` is done, the result will be stored and the record line could be garbarge collected. But for other patterns, for example the third pattern, this is not the case:

```
for (record <- records) {
  val fResult = f(record)
  store(fResult)  // need to store the intermediate result here
}

for (record <- fResult) {
  g(record)
  ...
}
```

It's clear that `f`'s result need to be stored somewhere.

Let's go back to our problem with stages and tasks. The main issue of our aggressive idea is that we can't actually pipelines then data flow if there's a `ShuffleDependency`. Then how about to cut the data flow at each `ShuffleDependency`? That leaves us with chains of RDDs connected by `NarrowDependency` and we know that `NarrowDependency` can be pipelined. So we can just divide the logical plan into stages like this:

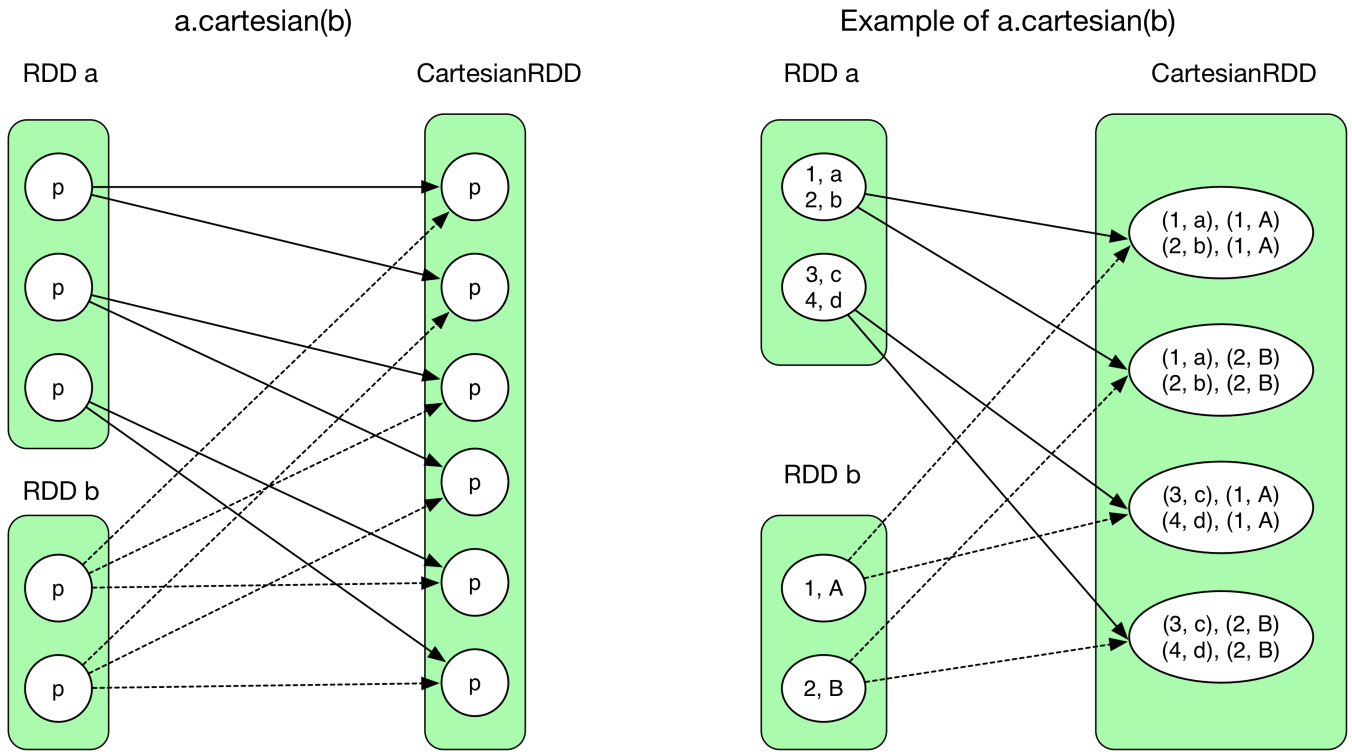


The strategy for creating stages is to: **check backwards from the final RDD, add each `NarrowDependency` into the current stage and break out for a new stage when there's a `ShuffleDependency`. In each stage, the task number is determined by the partition number of the last RDD in the stage.**

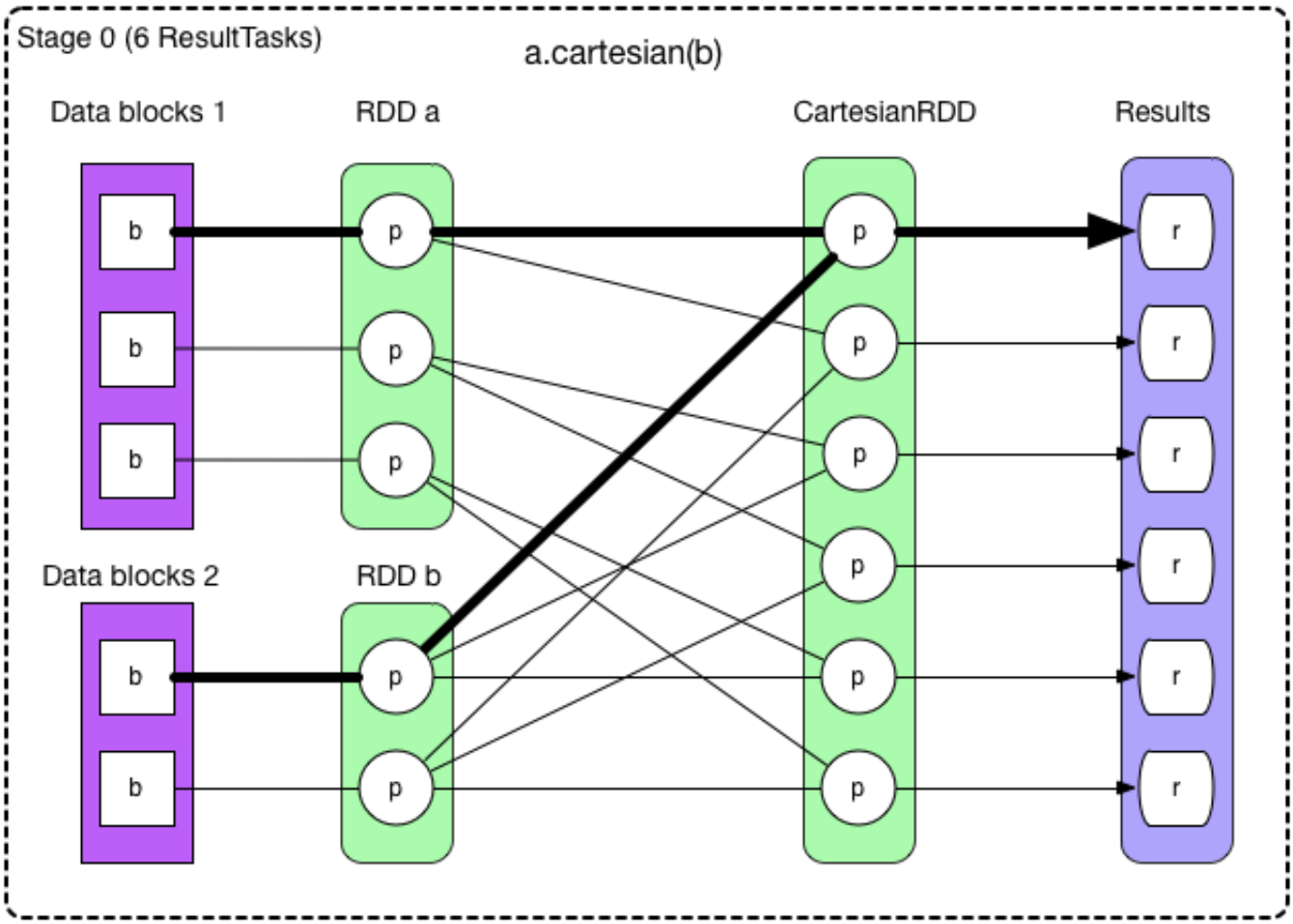
In above diagram, all thick arrows represent tasks. Since the stages are determined backwards, the last stage's id is 0, stage 1 and stage 2 are both parent to stage 0. **If a stage gives the final result, then its tasks are of type `ResultTask`, otherwise they are `ShuffleMapTask`.** `ShuffleMapTask` gets its name because its result needs to be shuffled before goes into the next stage, it's similar to the mappers in Hadoop MapReduce. `ResultTask` can be seen as reducer in Hadoop (if it gets shuffled data from its parent stage), or it could be mappers (if the stage has no parent).

One problem remains: `NarrowDependency` chain can be pipelined, but in our example application, we've showed only `OneToOneDependency` and `RangeDependency`, how about other `NarrowDependency`?

Let's check back the cartesian operation in the last chapter with complex `NarrowDependency` inside:



With stages:



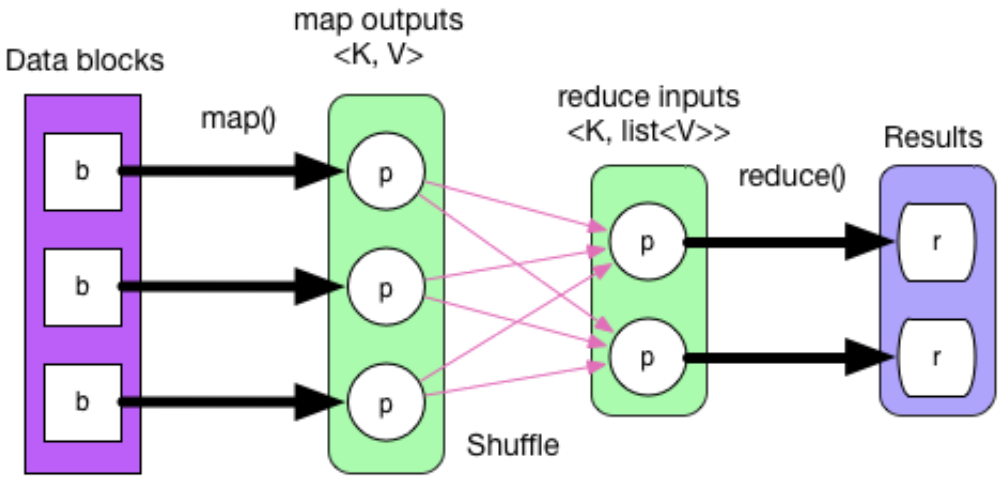
The thick arrows represents the first `ResultTask`. Since the stage gives directly the final result, in above

diagram we have 6 `ResultTask`. Different with `OneToOneDependency`, each `ResultTask` in this job needs to evaluate 3 RDDs and read 2 data blocks, all executed in one single task. **We can see that regardless of the actual type of `NarrowDependency`, be it 1:1 or N:N, `NarrowDependency` chain can always be pipelined. The number of task needed is the same as the partition number in the final RDD.**

Execution of the Physical Plan

We have stages and tasks, next problem: **how the tasks are executed for the final result?**

Let's go back to the physical plan of our example application of the chapter. Recall that in Hadoop MapReduce, the tasks are executed in order, `map()` generates map outputs, which in term gets partitioned and written to local disk. Then `shuffle-sort-aggregate` process is applied to generate reduce inputs. Finally `reduce()` executes for the final result. This process is illustrated in the following diagram:



This execution process can not be used directly on Spark's physical plan since Hadoop MapReduce's physical plan is simple and fixed, and without pipelining.

The main idea of pipelining is that **the data is computed when they are actually needed in a flow fashion.** We start from the final result (this is obviously where the computation is needed) and check backwards the RDD chain to find what are the RDDs and partitions that need to be evaluated for computing the final result. In most cases, we trace back to some partitions in the leftmost RDD and they are the first to be evaluated.

For a stage without parent, its leftmost RDD can be evaluate directly (it has no dependency), and each record evaluated can be streamed into the subsequent computations (pipelining). The computation chain is deduced backwards from the final step, but the actual execution streams the records forwards. One record goes through the whole computation chain before the computation of the next record starts.

For stages with parent, we need to execute its parent stages and fetch the data through shuffle. Once it's done, it becomes the same case as a stage without parent.

> In the code, each RDD's `getDependency()` method declares its data dependency. `compute()` method is in charge of receiving upstream records (from parent RDD or data source) and applying computation logic. We see often code like this in RDDs: `firstParent[T].iterator(split, context).map(f)`. `firstParent` is the first dependent RDD, `iterator()` shows that the records are consumed one by one, and `map(f)` applies the computation logic on each record. The `compute()` method returns an iterator for next computation.

Summary so far: **The whole computation chain is created by checking backwards the data depency from the last RDD. Each `ShuffleDependency` separates stages. In each stage, each RDD's `compute()` method calls `parentRDD.itererator()` to receive the upstream record stream.**

Notice that `compute()` method is reserved only for computation logic that generates output records from parent RDDs. The actual dependent RDDs are declared in `getDependency()` method. The actual dependent partitions are declared in `dependency.getParents()` method.

Let's check the `CartesianRDD` as an example:

```
 // RDD x is the cartesian product of RDD a and RDD b
 // RDD x = (RDD a).cartesian(RDD b)
 // Defines how many partitions RDD x should have, what are the types for
each partition
 override def getPartitions: Array[Partition] = {
    // create the cross product split
    val array = new Array[Partition](rdd1.partitions.size *
rdd2.partitions.size)
    for (s1 <- rdd1.partitions; s2 <- rdd2.partitions) {
      val idx = s1.index * numPartitionsInRdd2 + s2.index
      array(idx) = new CartesianPartition(idx, rdd1, rdd2, s1.index,
s2.index)
    }
    array
  }

  // Defines the computation logic for each partition of RDD x (the result
RDD)
  override def compute(split: Partition, context: TaskContext) = {
    val currSplit = split.asInstanceOf[CartesianPartition]
    // s1 shows that a partition in RDD x depends on one partition in RDD a
    // s2 shows that a partition in RDD x depends on one partition in RDD b
```

```
    for (x <- rdd1.iterator(currSplit.s1, context);
         y <- rdd2.iterator(currSplit.s2, context)) yield (x, y)
  }

  // Defines which are the dependent partitions and RDDs for partition i in
RDD x
  //
  // RDD x depends on RDD a and RDD b, both in `NarrowDependency`
  // For the first dependency, partition i in RDD x depends on the partition
with
  //     index `i / numPartitionsInRDD2` in RDD a
  // For the second dependency, partition i in RDD x depends on the partition
with
  //     index `i % numPartitionsInRDD2` in RDD b
  override def getDependencies: Seq[Dependency[_]] = List(
    new NarrowDependency(rdd1) {
      def getParents(id: Int): Seq[Int] = List(id / numPartitionsInRdd2)
    },
    new NarrowDependency(rdd2) {
      def getParents(id: Int): Seq[Int] = List(id % numPartitionsInRdd2)
    }
  )
```

Job Creation

Now we've introduced the logical plan and physical plan, then **how and when a job is created? What exactly is a job?**

The following table shows the typical action(). The second column is `processPartition()` method, it defines how to process the records in each partition for the result. The third column is `resultHandler()` method, it defines how to process the partial results from each partition to form the final result.

Action	finalRDD(records) => result	compute(results)
reduce(func)	(record1, record2) => result, (result, record i) => result	(result1, result 2) => result, (result, result i) => result
collect()	Array[records] => result	Array[result]
count()	count(records) => result	sum(result)
foreach(f)	f(records) => result	Array[result]

take(n)	record (i<=n) => result	Array[result]
first()	record 1 => result	Array[result]
takeSample()	selected records => result	Array[result]
takeOrdered(n, [ordering])	TopN(records) => result	TopN(results)
saveAsHadoopFile(path)	records => write(records)	null
countByKey()	(K, V) => Map(K, count(K))	(Map, Map) => Map(K, count(K))

Each time there's an `action()` in user's driver program, a job will be created. For example `foreach()` action will call `sc.runJob(this, (iter: Iterator[T]) => iter.foreach(f))` to submit a job to the `DAGScheduler`. If there's other `action()`s in the driver program, there will be other jobs submitted. So we'll have as many jobs as the `action()` operations in a driver program. This is why in Spark a driver program is called an application rather than a job.

The last stage of a job generates the job's result. For example in the `GroupByTest` in the first chapter, there's 2 jobs with 2 sets of results. When a job is submitted, the `DAGScheduler` applies the strategy to figure out the stages, and submit firstly the **stages without parents** for execution. In this process, the number and type of tasks are also determined. A stage is executed after its parent stages' execution.

Details in Job Submission

Let's briefly analyze the code for job creation and submission. We'll come back to this part in the Architecture chapter.

1. `rdd.action()` calls `DAGScheduler.runJob(rdd, processPartition, resultHandler)` to create a job.
2. `runJob()` gets the partition number and type of the final RDD by calling `rdd.getPartitions()`. Then it allocates `Array[Result](partitions.size)` for holding the results based on the partition number.
3. Finally `runJob(rdd, cleanedFunc, partitions, allowLocal, resultHandler)` in `DAGScheduler` is called to submit the job. `cleanedFunc` is the closure-cleaned version of `processPartition`. In this way this function can be serialized and sent to the different worker nodes.
4. `DAGScheduler`'s `runJob()` calls `submitJob(rdd, func, partitions, allowLocal, resultHandler)` to submit a job.
5. `submitJob()` gets a `jobId`, then wrap the function once again and send a `JobSubmitted` message to `DAGSchedulerEventProcessActor`. Upon receiving this message, the actor calls `dagScheduler.handleJobSubmitted()` to handle the submitted job. This is an example of event-

driven programming model.

6. `handleJobSubmitted()` firstly calls `finalStage = newStage()` to create stages, then it `submitStage(finalStage)`. If `finalStage` has parents, the parent stages will be submitted first. In this case, `finalStage` is actually submitted by `submitWaitingStages()`.

How `newStage()` divide an RDD chain in stages?

- This method calls `getParentStages()` of the final RDD when instantiating a new stage (`new Stage(...)`)
- `getParentStages()` starts from the final RDD, check backwards the logical plan. It adds the RDD into the current stage if it's a `NarrowDependency`. When it meets a `ShuffleDependency` between RDDs, it takes in the right-side RDD (the RDD after the shuffle) and then concludes the current stage. Then the same logic is applied on the left hand side RDD of the shuffle to form another stage.
- Once a `ShuffleMapStage` is created, its last RDD will be registered `MapOutputTrackerMaster.registerShuffle(shuffleDep.shuffleId, rdd.partitions.size)`. This is important since the shuffle process needs to know the data output location from `MapOuputTrackerMaster`.

Now let's see how `submitStage(stage)` submits stages and tasks:

1. `getMissingParentStages(stage)` is called to determine the `missingParentStages` of the current stage. If the parent stages are all executed, `missingParentStages` will be empty.
2. If `missingParentStages` is not empty, then recursively submit these missing stages, and the current stage is inserted into `waitingStages`. Once the parent stages are done, stages inside `waitingStages` will be run.
3. if `missingParentStages` is empty, then we know the stage can be executed right now. Then `submitMissingTasks(stage, jobId)` is called to generate and submit the actual tasks. If the stage is a `ShuffleMapStage`, then we'll allocate as many `ShuffleMapTask` as the partition number in the final RDD. In the case of `ResultStage`, `ResultTask` instances are allocated instead. The tasks in a stage form a `TaskSet`. Finally `taskScheduler.submitTasks(taskSet)` is called to submit the whole task set.
4. The type of `taskScheduler` is `TaskSchedulerImpl`. In `submitTasks()`, each `taskSet` gets wrapped in a `manager` variable of type `TaskSetManager`, then we pass it to `schedulableBuilder.addTaskSetManager(manager)`. `schedulableBuilder` could be `FIFOSchedulableBuilder` or `FairSchedulableBuilder`, depending on the configuration. The last step of `submitTasks()` is to inform `backend.reviveOffers()` to run the task. The type of backend is `SchedulerBackend`. If the application is run on a cluster, its type will be `SparkDeploySchedulerBackend`.
5. `SparkDeploySchedulerBackend` is a subclass of `CoarseGrainedSchedulerBackend`, `backend.reviveOffers()` actually sends `ReviveOffers` message to `DriverActor`. `SparkDeploySchedulerBackend` launches a `DriverActor` when it starts. Once `DriverActor`

receives the `ReviveOffers` message, it will call `launchTasks(scheduler.resourceOffers(Seq(new WorkerOffer(executorId, executorHost(executorId), freeCores(executorId)))))` to launch the tasks. `scheduler.resourceOffers()` obtains the sorted `TaskSetManager` from the FIFO or Fair scheduler and gethers other information about the tasks from `TaskSchedulerImpl.resourceOffer()`. These information are stored in a `TaskDescription`. In this step the data locality information is also considered.

6. `launchTasks()` in the `DriverActor` serialize each task. If the serialized size does not exceed the `akkaFrameSize` limit of Akka, then the task is finally sent to the executor for execution: `executorActor(task.executorId) ! LaunchTask(new SerializableBuffer(serializedTask))`.

Discussion

Up till now, we've discussed:

- how the driver program triggers jobs
- how to generate a physical plan from a logical plan
- what is pipelining in Spark and how it is implemented
- the actual code of job creation and submission

However, there's subjects that we've left for details:

- the shuffle process
- task execution and its execution location

In the next chapter we'll discuss the shuffle process in Spark.

In my personal opinion, the transformation from logical plan to physical plan is really a masterpiece. The abstractions, such as dependencies, stages and tasks are all well defined and the logic of the algorithms are very clear.

Source Code of the Example Job

```
package internals

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.HashPartitioner
```

```
object complexJob {
  def main(args: Array[String]) {

    val sc = new SparkContext("local", "ComplexJob test")

    val data1 = Array[(Int, Char)](
      (1, 'a'), (2, 'b'),
      (3, 'c'), (4, 'd'),
      (5, 'e'), (3, 'f'),
      (2, 'g'), (1, 'h'))
    val rangePairs1 = sc.parallelize(data1, 3)

    val hashPairs1 = rangePairs1.partitionBy(new HashPartitioner(3))


    val data2 = Array[(Int, String)]((1, "A"), (2, "B"),
      (3, "C"), (4, "D"))

    val pairs2 = sc.parallelize(data2, 2)
    val rangePairs2 = pairs2.map(x => (x._1, x._2.charAt(0)))


    val data3 = Array[(Int, Char)]((1, 'X'), (2, 'Y'))
    val rangePairs3 = sc.parallelize(data3, 2)


    val rangePairs = rangePairs2.union(rangePairs3)


    val result = hashPairs1.join(rangePairs)

    result.foreachWith(i => i)((x, i) => println("[result " + i + "] " + x))

    println(result.toDebugString)
  }
}
```

JerryLead/SparkInternals

Shuffle Process

Previously we've discussed Spark's physical plan and its execution details. But one thing is left untouched: **how data gets through a `ShuffleDependency` to the next stage?**

Shuffle Comparison between Hadoop and Spark

There're some differences and also similarities between the shuffle process in Hadoop and in Spark:

From a high-level point of view, they are similar. They both partition the mapper's (or `ShuffleMapTask` in Spark) output and send each partition to its corresponding reducer (in Spark, it could be a `ShuffleMapTask` in the next stage, or a `ResultTask`). The reducer buffers the data in memory, shuffles and aggregates the data, and applies the `reduce()` logic once the data is aggregated.

From a low-level point of view, there're quite a few differences. The shuffle in Hadoop is sort-based since the records must be sorted before `combine()` and `reduce()`. The sort can be done by an external sort algorithm thus allowing `combine()` or `reduce()` to tackle very large datasets. Currently in Spark the default shuffle process is hash-based. Usually it uses a `HashMap` to aggregate the shuffle data and no sort is applied. If the data needs to be sorted, user has to call `sortByKey()` explicitly. In Spark 1.1, we can set the configuration `spark.shuffle.manager` to `sort` to enable sort-based shuffle. In Spark 1.2, the default shuffle process will be sort-based.

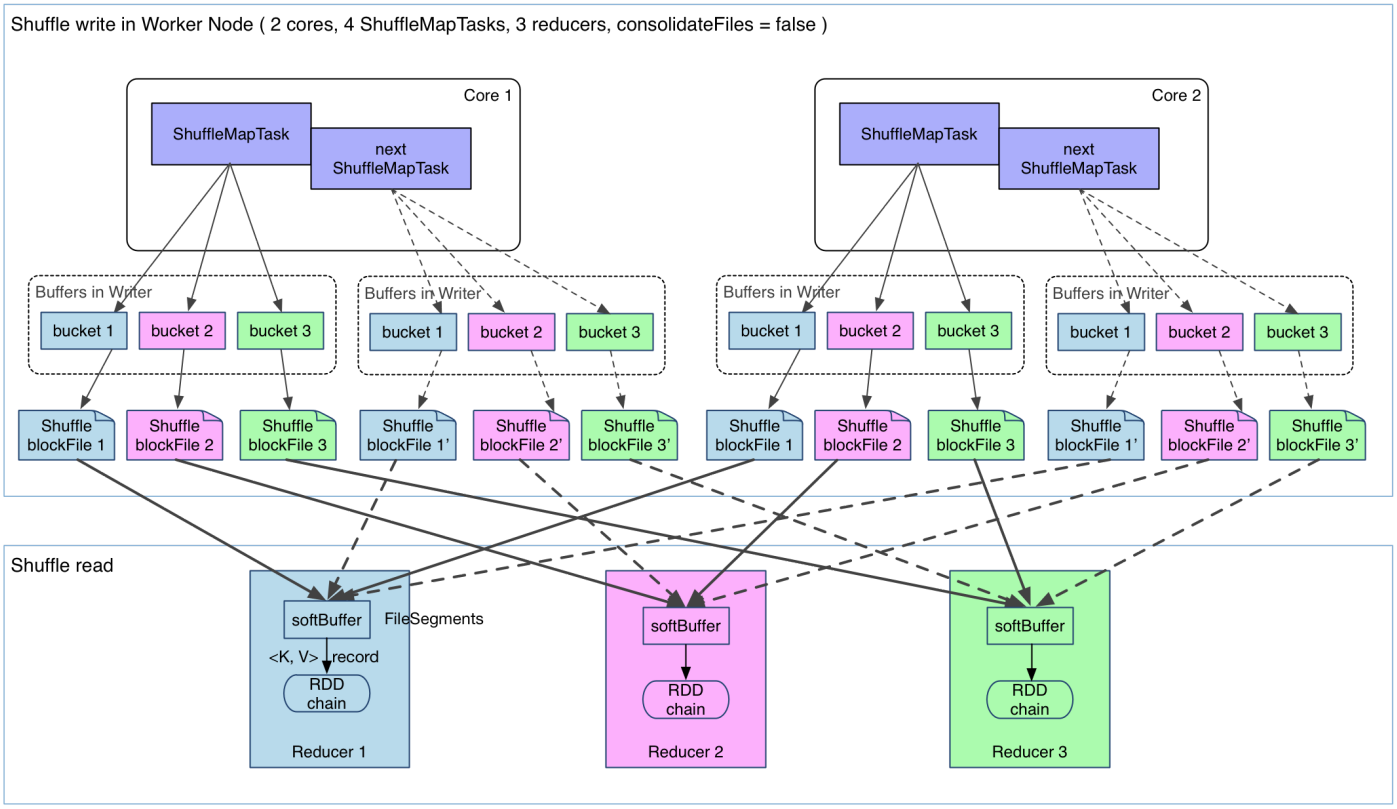
Implementation-wise, there're also differences. As we know, there are obvious steps in a Hadoop workflow: `map()`, `spill`, `merge`, `shuffle`, `sort` and `reduce()`. Each step has a predefined responsibility and it fits the procedural programming model well. However in Spark, there's no such fixed steps, instead we have stages and a series of transformations. So operations like `spill`, `merge` and `aggregate` need to be somehow included in the transformations.

If we name the mapper side process of partitioning and persisting data "shuffle write", and the reducer side reading and aggregating data "shuffle read". Then the problem becomes: **How to integrate shuffle write and shuffle read logic in Spark's logical or physical plan? How to implement shuffle write and shuffle read efficiently?**

Shuffle Write

Shuffle write is a relatively simple task if a sorted output is not required. It partitions and persists the data. The persistance of data here has two advantages: reducing heap pressure and enhancing fault-tolerance.

Its implementation is simple: add the shuffle write logic at the end of `ShuffleMapStage` (in which there's a `ShuffleMapTask`). Each output record of the final RDD in this stage is partitioned and persisted, as shown in the following diagram:



In the diagram there're 4 `ShuffleMapTask`s to execute in the same worker node with 2 cores. The task result (records of the final RDD in the stage) is written on the local disk (data persistence). Each task has `R` buffers, `R` equals the number of reducers (the number if tasks in the next stage). The buffers are called buckets in Spark. By default the size of each bucket is 32KB (100KB before Spark 1.1) and is configurable by `spark.shuffle.file.buffer.kb` .

> In fact bucket is a general concept in Spark that represents the location of the partitioned output of a `ShuffleMapTask`. Here for simplicity a bucket is referred to an in-memory buffer.

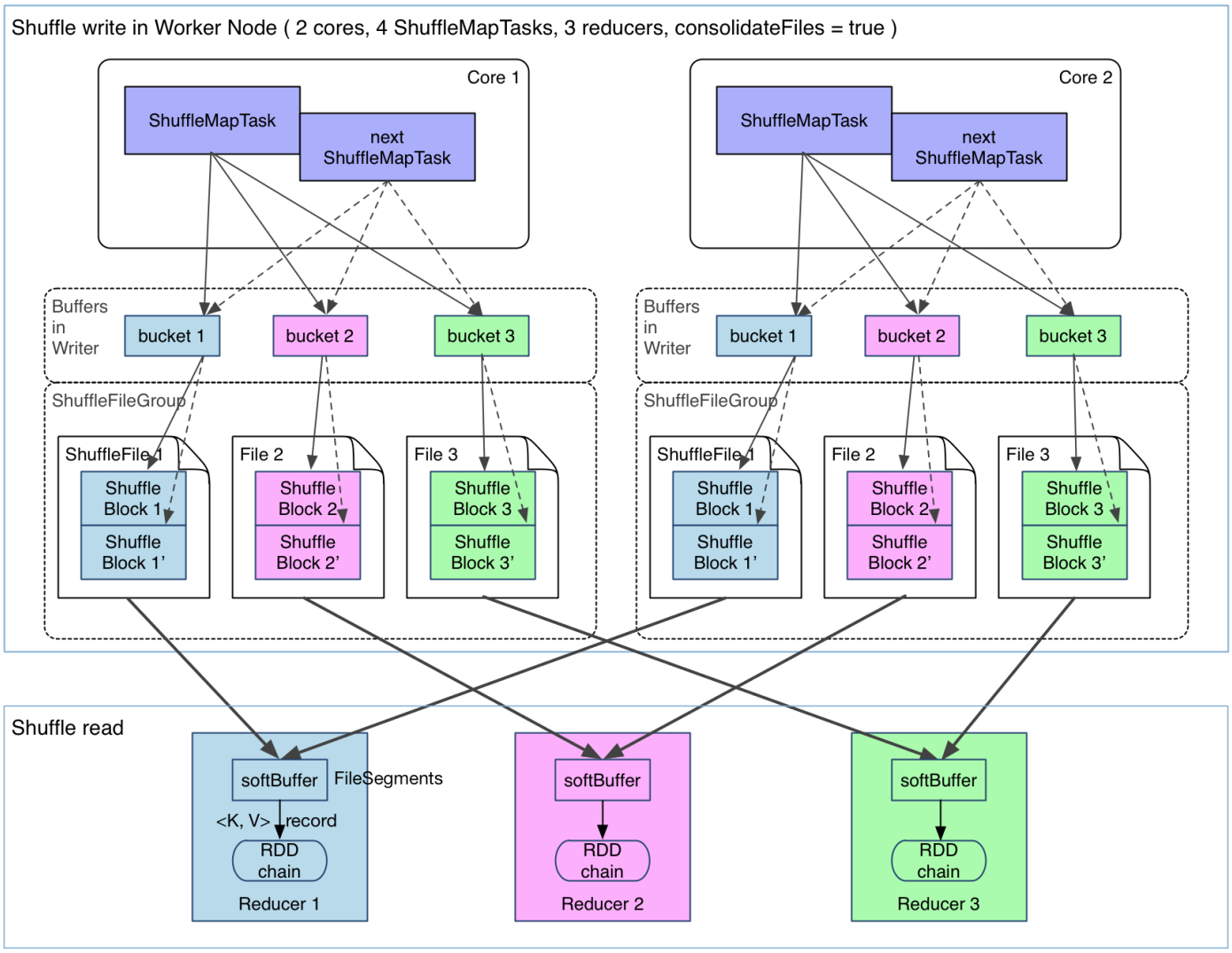
`ShuffleMapTask` employs the pipelining techinque to compute the result records of the final RDD. Each record is sent to the bucket of its corresponding partition, which is determined by `partitioner.partition(record.getKey())`. The content of these buckets is written continuously to local disk files called `ShuffleBlockFile`, or `FileSegment` for short. Reducers will fetch their `FileSegment` in shuffle read phase.

An implementation like this is very simple, but has some issues:

1. **We may produce too many `FileSegment`.** Each `ShuffleMapTask` produces `R`(number of reducers) `FileSegment`, so `M` `ShuffleMapTask` will produce `M * R` files. For big datasets we could have big `M` and `R`, as a result there may be lots of intermediate data files.

2. **Buffers could take a lot of space.** On a worker node, we could have `R * M` buckets for each core available to Spark. Spark will reuse the buffer space after a `ShuffleMapTask` but there could still be `R * cores` buckets in memory. On a node with 8 cores processing a 1000-reducer job, buckets will take up 256MB (`R * cores * 32KB`).

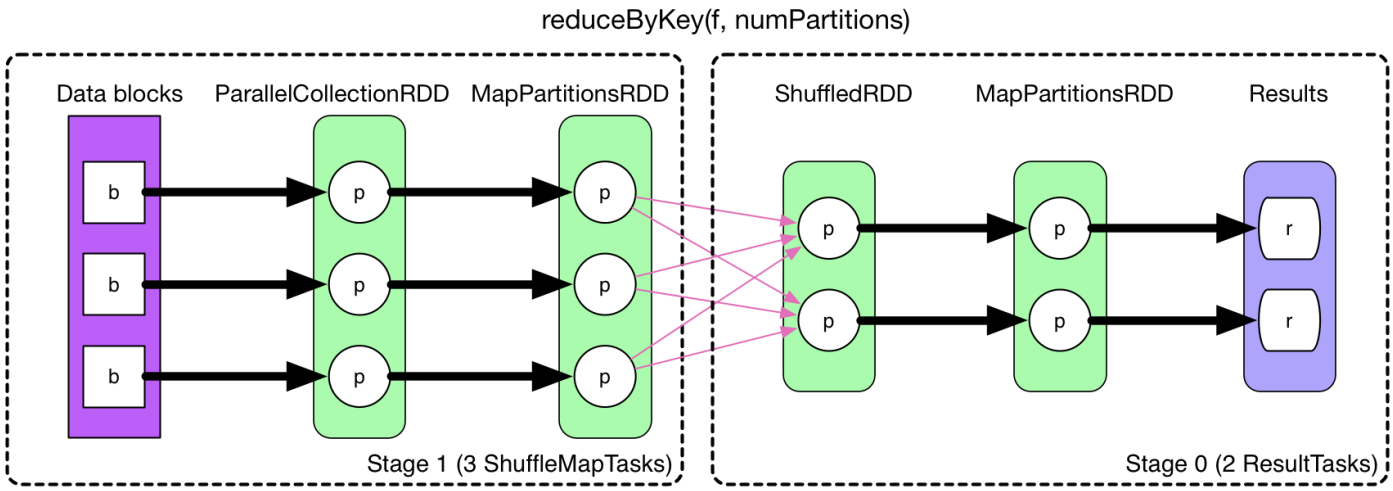
Currently, there's no good solution to the second problem. We need to write buffers anyway and if they're too small there will be impact on IO speed. For the first problem, we have a file consolidation solution already implemented in Spark. Let's check it out:

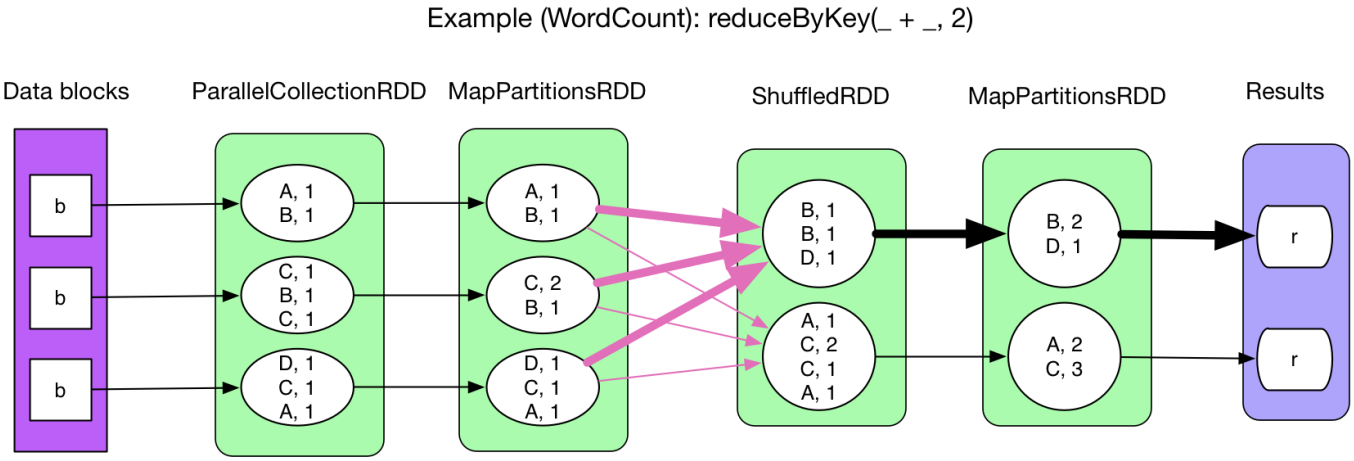


It's clear that from the above diagram, consecutive `ShuffleMapTask`s running on the same core share a shuffle file. Each task appends its output data, `ShuffleBlock` i', after the output data of the previous task, `ShuffleBlock` i. A `ShuffleBlock` is called a `FileSegment`. In this way, reducers in the next stage can just fetch the whole file and we reduce the number of files needed in each worker node to `cores * R`. File consolidation feature can be activated by setting `spark.shuffle.consolidateFiles` to true.

Shuffle Read

Let's check a physical plan of `reduceBykey`, which contains `ShuffleDependency`:





Intuitively, we need to fetch the data of `MapPartitionRDD` to be able to evaluate `ShuffleRDD`. Then comes the problems:

- When to fetch? Fetch for each `ShuffleMapTask` or fetch only once after all `ShuffleMapTask`s are done?
- Fetch and process the records at the same time or fetch and then process?
- Where to store the fetched data?
- How do the tasks of the next stage know the location of the fetched data?

Solutions in Spark:

- **When to fetch?** Wait after all `ShuffleMapTask`s end and then fetch. We know that a stage will be executed only after its parent stages are executed, so it's intuitive that the fetch operation begins after all `ShuffleMapTask`s in the previous stage are done. The fetched `FileSegments` have to be buffered in memory, so we can't fetch too much before the buffer content is written to disk. Spark limits this buffer size by `spark.reducer.maxMbInFlight`, here we name it `softBuffer`. It has default size 48MB. A `softBuffer` usually contains multiple fetched `FileSegments`. But sometimes one single segment can fill up the buffer.

- **Fetch and process the records at the same time or fetch and then process?** Fetch and process the records at the same time. In MapReduce, the shuffle stage fetches the data and then applies `combine()` logic at the same time. However in MapReduce the reducer input data needs to be sorted, so the `reduce()` logic is applied after the shuffle-sort process. Since Spark does not require a sorted order for the reducer input data, we don't need to wait until all the data gets fetched to start processing. **Then how Spark implements this shuffle and processing?** In fact Spark utilizes data structures like HashMap to do the job. Each <Key, Value> pair from the shuffle process is inserted into a HashMap. If the `Key` is already present, then the pair is aggregated by `func(hashMap.get(Key), Value)`. In the above WordCount example, the `func` is `hashMap.get(Key) + Value`, and its result is updated in the HashMap. This `func` has a similar role to `reduce()` in Hadoop, but they differ in details. We illustrate the difference by the following code snippet:

```
// MapReduce
reduce(K key, Iterable<V> values) {
    result = process(key, values)
    return result
}

// Spark
reduce(K key, Iterable<V> values) {
    result = null
    for (V value : values)
        result  = func(result, value)
    return result
}
```

In Hadoop MapReduce, we can define any data structure we like in `process` function. It's just a function that takes an `Iterable` as parameter. We can also choose to cache the `values` for further processing. In Spark, a `foldLeft` like technique is used to apply the `func`. For example, in Hadoop, it's very easy to compute the average out of `values`: `sum(values) / values.length`. But it's not the case in the Spark model. We'll come back to this part later.

- **Where to store the fetched data?** The fetched `FileSegment`s get buffered in `softBuffer`. Then the data is processed, and written to a configurable location. If `spark.shuffle.spill` is false, then the write location is only memory. A special data structure, `AppendOnlyMap`, is used to hold these processed data in memory. Otherwise, the processed data will be written to memory and disk, using `ExternalAppendOnlyMap`. This data structure can spill the sorted key-value pairs on disk when there isn't enough memory available. **A key problem in using both memory and disk is how to find a balance of the two.** In Hadoop, by default 70% of the memory is reserved for shuffle data. Once 66% of this part of the memory is used, Hadoop starts the merge-combine-spill process. In Spark a similar

strategy is used. We'll talk about its details later in this chapter.

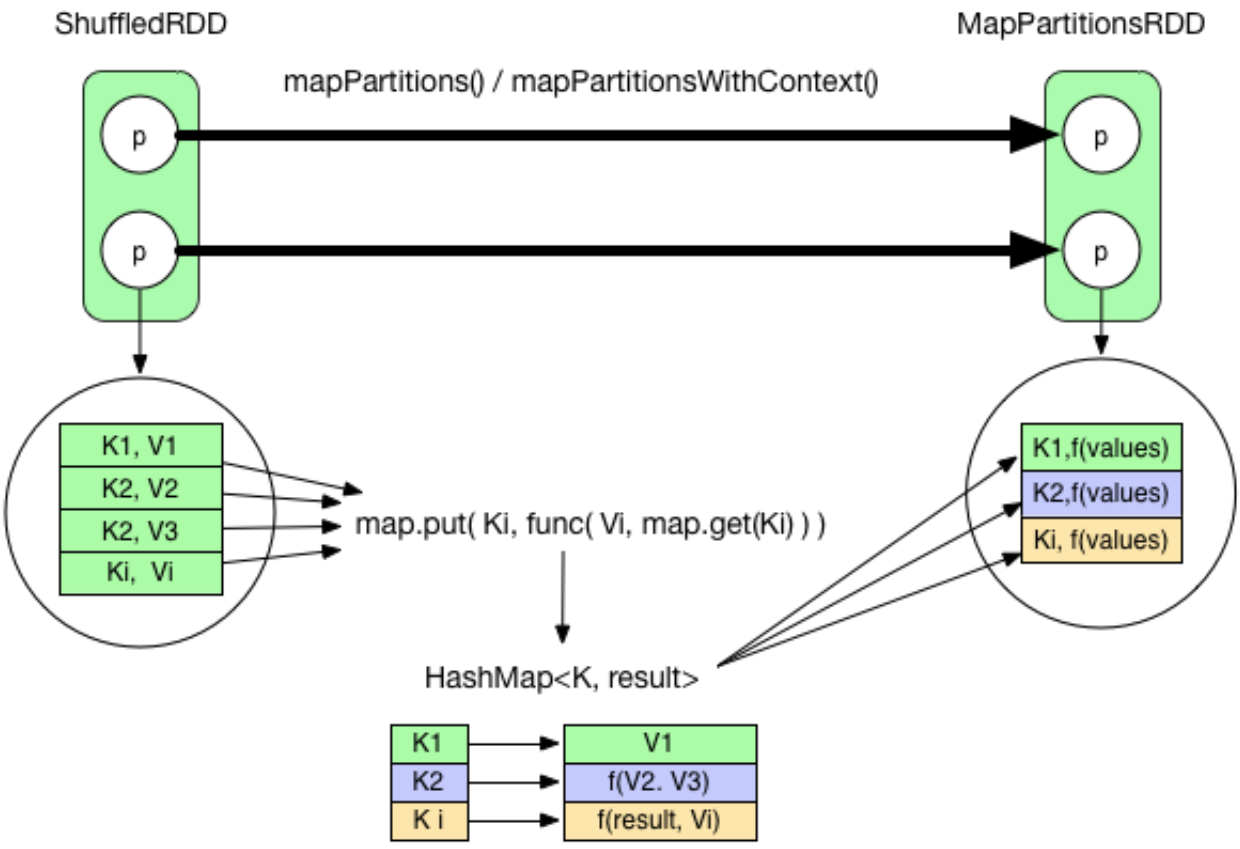
- **How do the tasks of the next stage know the location of the fetched data?** Recall that in the last chapter, there's an important step: `ShuffleMapStage`, which will register its final RDD by calling `MapOutputTrackerMaster.registerShuffle(shuffleId, rdd.partitions.size)`. So during the shuffle process, reducers get the data location by querying `MapOutputTrackerMaster` in the driver process. When a `ShuffleMapTask` finishes, it will report the location of its `FileSegment` to `MapOutputTrackerMaster`.

Now we have discussed the main ideas behind shuffle write and shuffle read as well as some implementation details. Let's dive into some interesting details.

Shuffle Read of Typical Transformations

`reduceByKey(func)`

We have briefly talked about the fetch and reduce process of `reduceByKey()`. Note that for an RDD, not all its data is present in the memory at a given time. The processing is always on a record basis. Processed record is rejected if possible. On a record level perspective, the `reduce()` logic can be shown as below:



We can see that the fetched records are aggregated using a HashMap, and once all the records are aggregated, we will have the result. The `func` needs to be commutative.

A `mapPartitionsWithContext` operation is used to transform the `ShuffledRDD` to a `MapPartitionsRDD`.

To reduce network trafic between nodes, we could use map side `combine()` in Hadoop. It's also feasible in Spark. All we need is to apply the `mapPartitionsWithContext` in the `ShuffleMapStage`. For example in `reduceByKey` , the transformation of `ParallelCollectionRDD` to `MapPartitionsRDD` is equivalent to a map side combine.

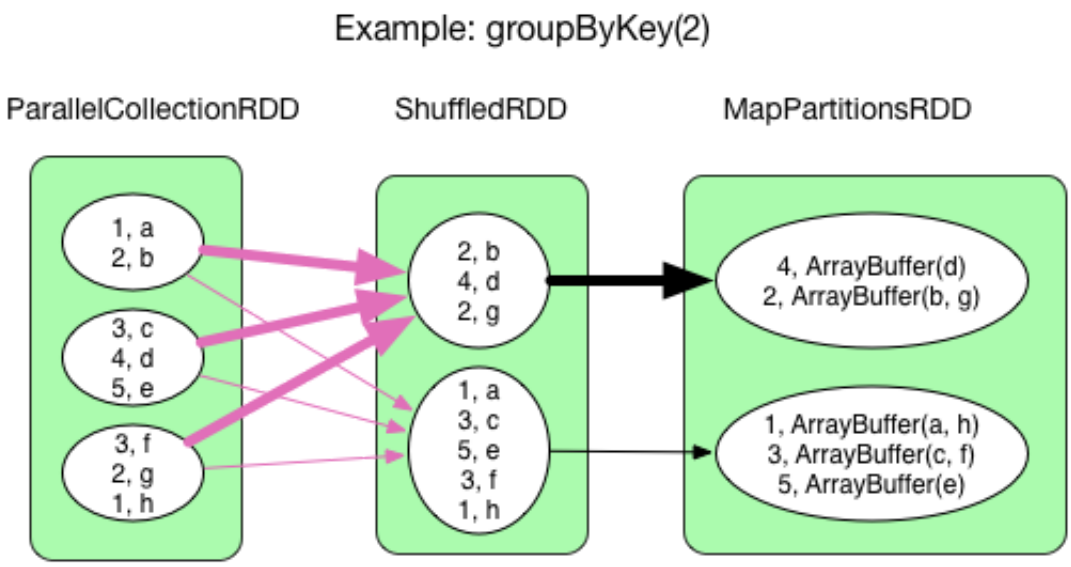
Comparison between map()->reduce() in Hadoop and `reduceByKey` in Spark

- map side: there's no difference on the map side. For `combine()` logic, Hadoop imposes a sort before `combine()`. Spark applies the `combine()` logic by using a hash map.
- reduce side: Shuffle process in Hadoop will fetch the data until a certain amount, then applies `combine()` logic, then merge sort the data to feed the `reduce()` function. In Spark fetch and reduce is done at the same time (in a hash map), so the reduce function need to be commutative.

Comparison in terms of memory usage

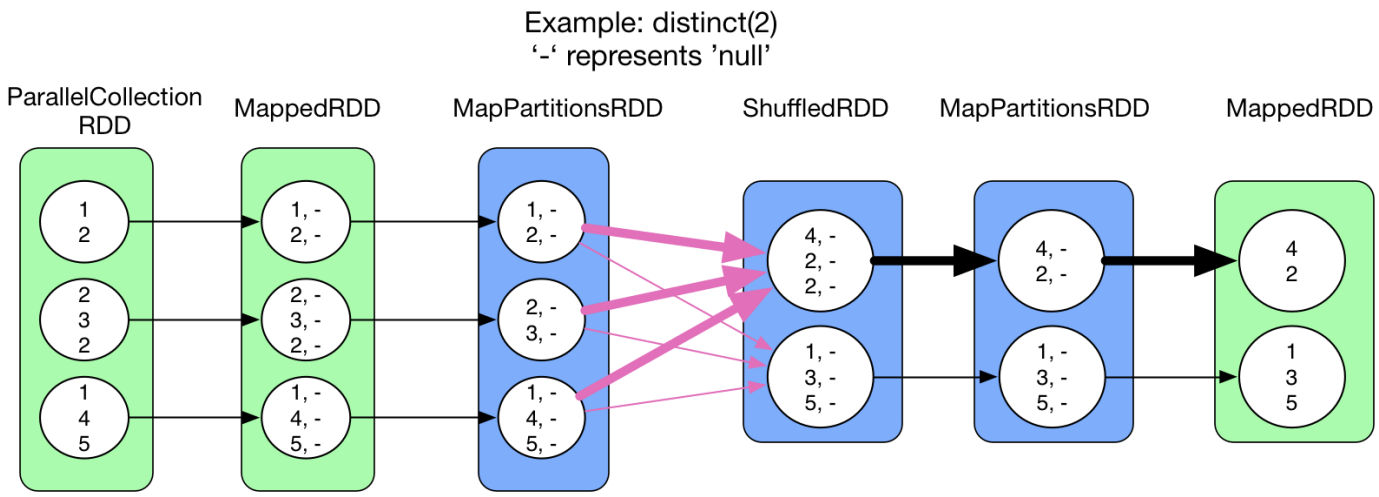
- map side: Hadoop needs a big, circular buffer to hold and sort the `map()` output data. But `combine()` does not need extra space. Spark needs a hash map to do `combine()`. And persisting records to local disk needs buffers (buckets).
- reduce side: Hadoop needs some memory space to store shuffled data. `combine()` and `reduce()` require no extra space since their input is sorted and can be grouped and then aggregated. In Spark, a `softBuffer` is needed for fetching. A hash map is used for storing the result of `combine()` and `reduce()`, if only memory is used in processing data. However, part of the data can be stored on disk if configured to use both memory and disk.

`groupByKey(numPartitions)`



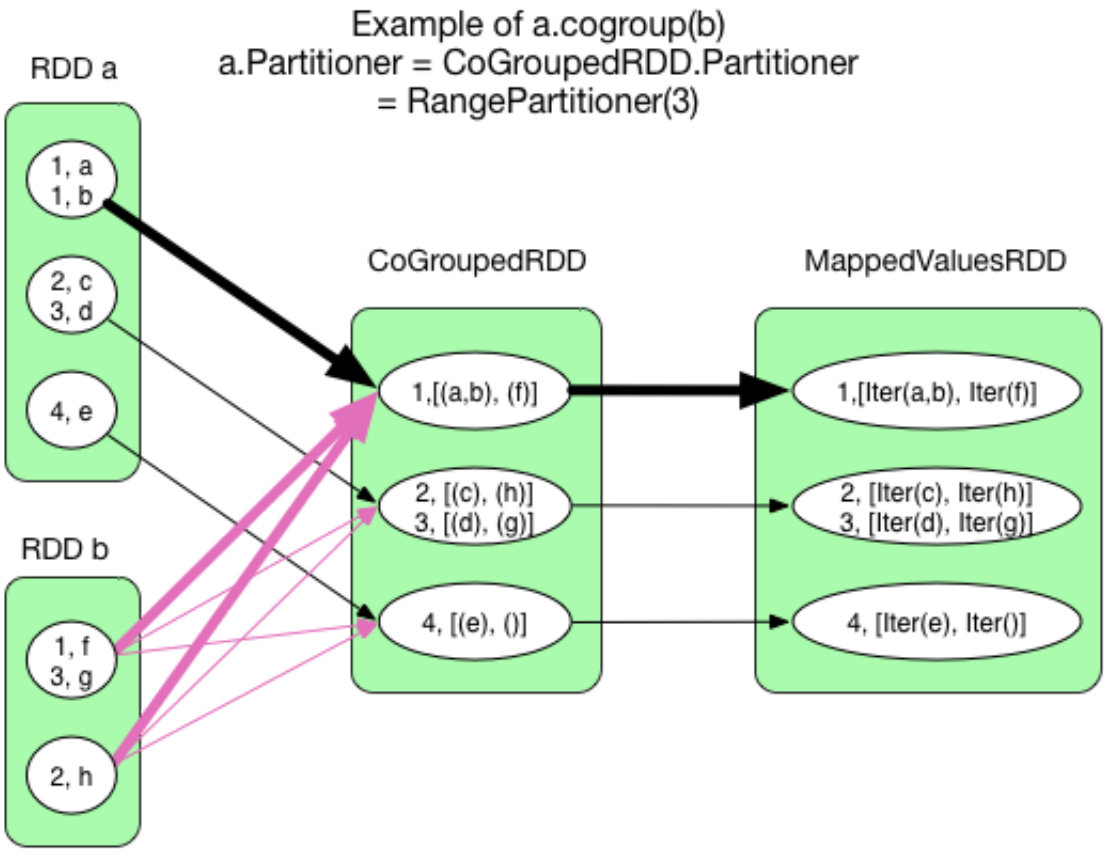
The process is similar to that of `reduceByKey()`. The `func` becomes `result = result ++ result.value`. This means that each key's values are grouped together without further aggregation.

`distinct(numPartitions)`



Similar to `reduceByKey()`. The `func` is `result = result == null ? record.value : result`. This means that we check the existence of the record in the `HashMap`. If it exists, reject the record, otherwise insert it into the map. Like `reduceByKey()`, there's map side `combine()`.

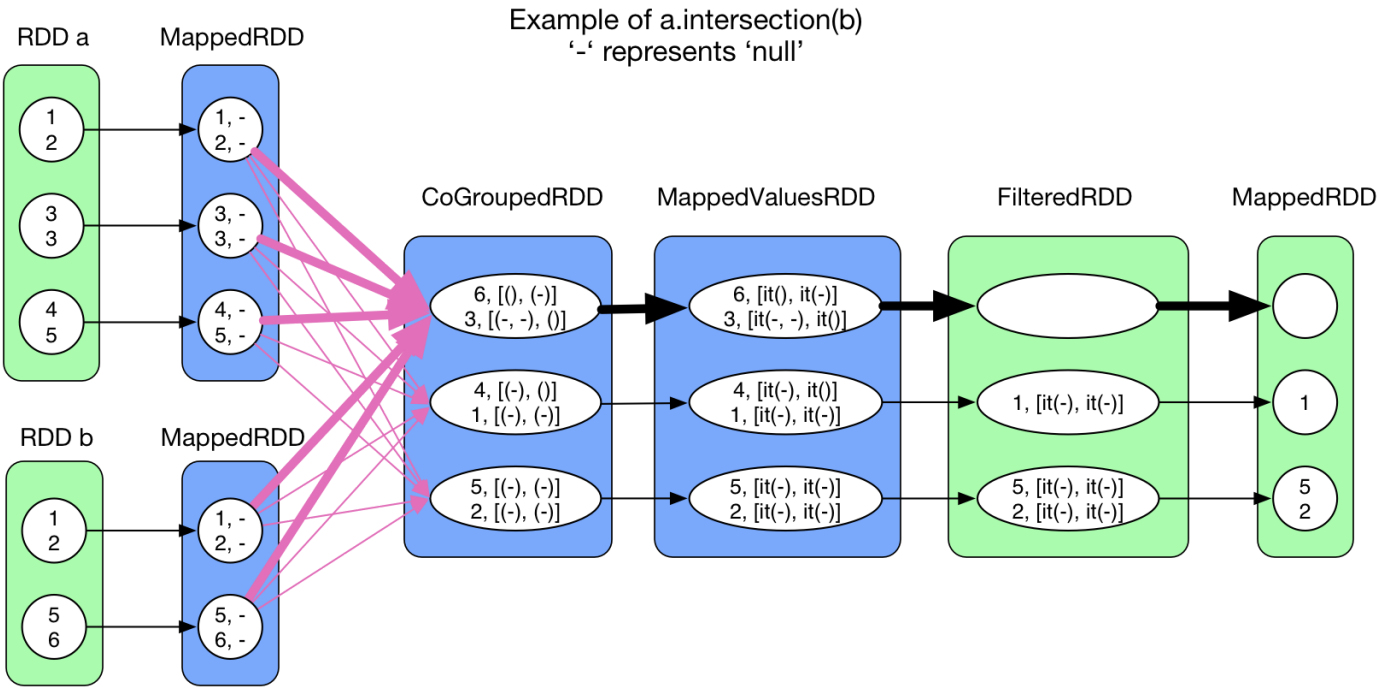
`cogroup(otherRDD, numPartitions)`

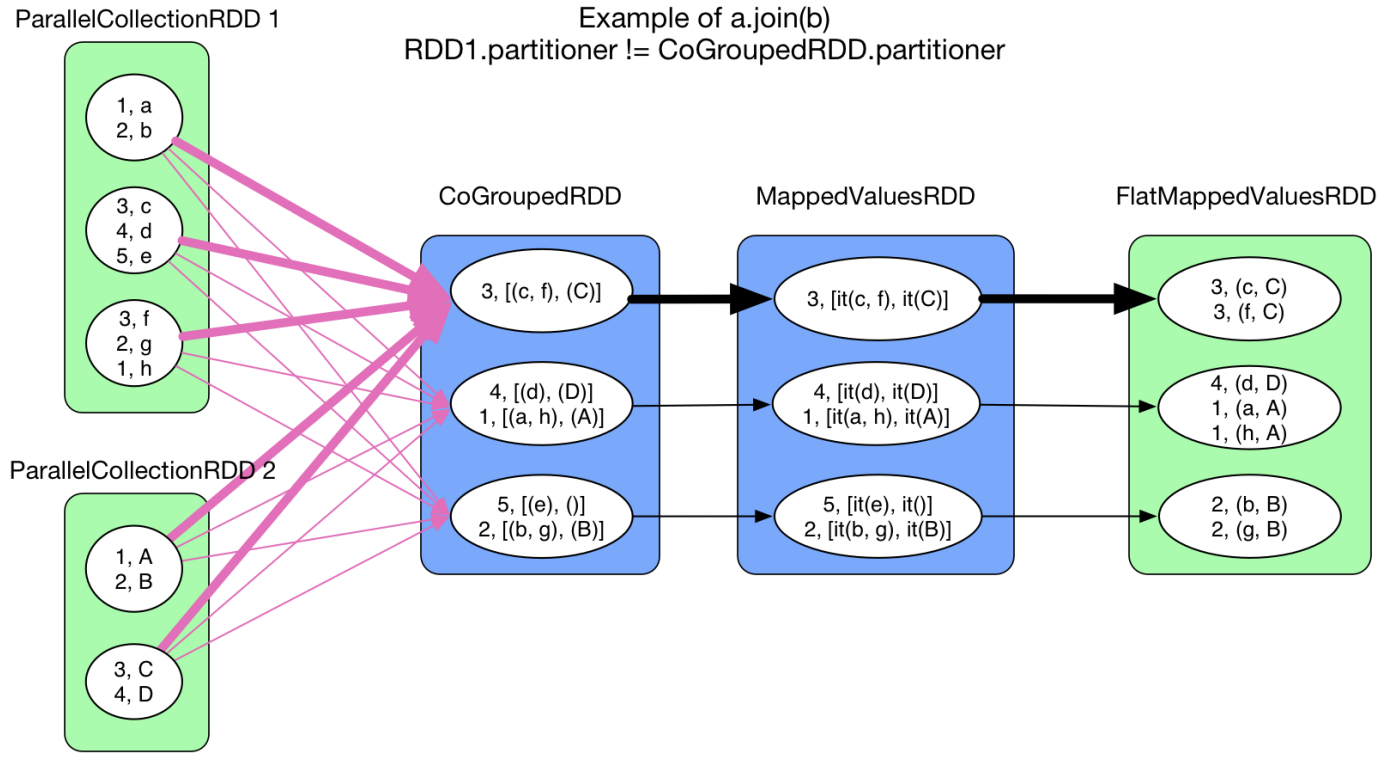


There could be 0, 1 or multiple `ShuffleDependency` for a `CoGroupedRDD`. But in the shuffle process we don't create a hash map for each shuffle dependency, but one hash map for all of them. Different from `reduceByKey`, the hash map is constructed in RDD's `compute()` rather than in `mapPartitionsWithContext()`.

A task of this RDD's execution will allocate an `Array[ArrayBuffer]`. This array contains the same number of empty `ArrayBuffer`s as the number of input RDDs. So in the example we have 2 `ArrayBuffers` in each task. When a key-value pair comes from RDD A, we add it to the first `ArrayBuffer`. If a key-value pair comes from RDD B, then it goes to the second `ArrayBuffer`. Finally a `mapValues()` operation transforms the values into the correct type: `(ArrayBuffer, ArrayBuffer)` => `(Iterable[V], Iterable[W])`.

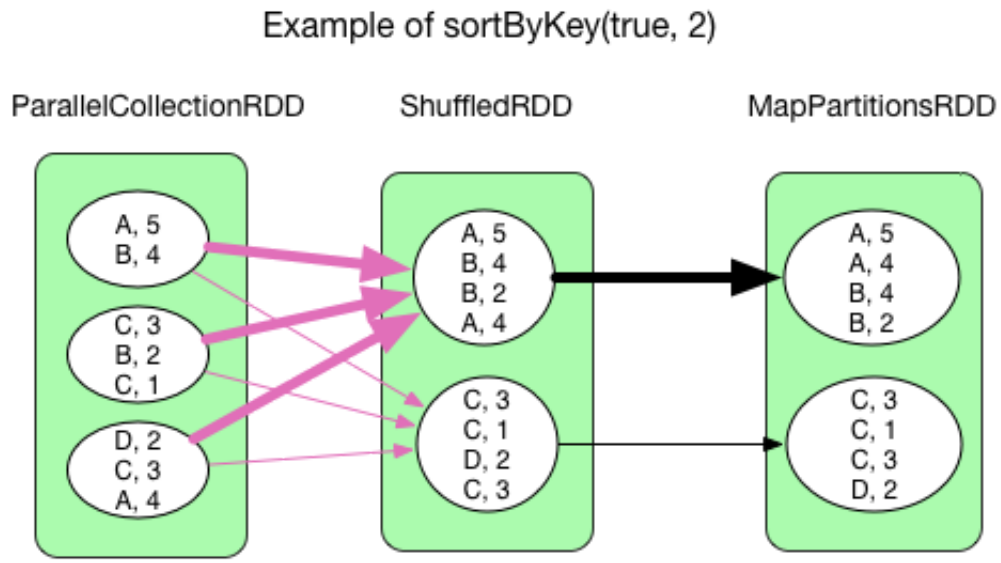
`intersection(otherRDD)` and `join(otherRDD, numPartitions)`





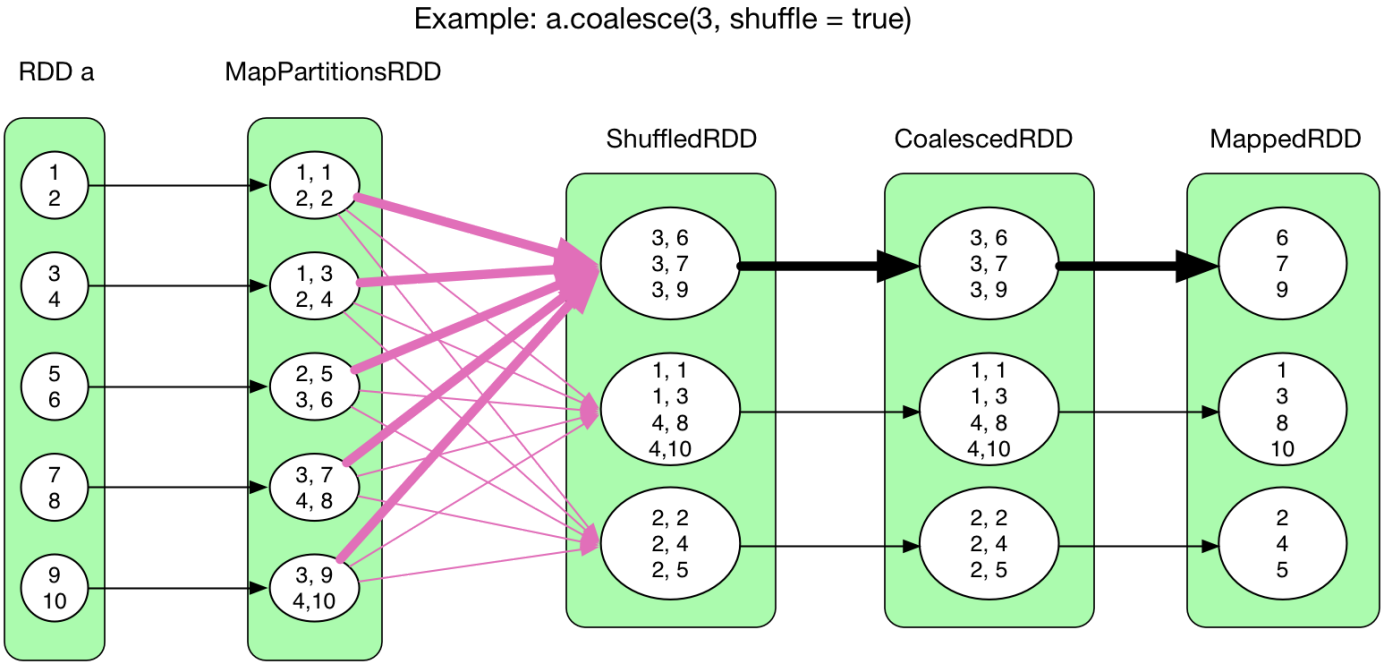
This two operations both use `cogroup`, so their shuffle process is identical to `cogroup`.

sortByKey(ascending, numPartition)



The processing logic of `sortByKey()` is a little different from `reduceByKey()` as it does not use a `HashMap` to handle incoming fetched records. Instead, all key-value pairs are range partitioned. The records of the same partition is sorted by key.

`coalesce(numPartitions, shuffle = true)`



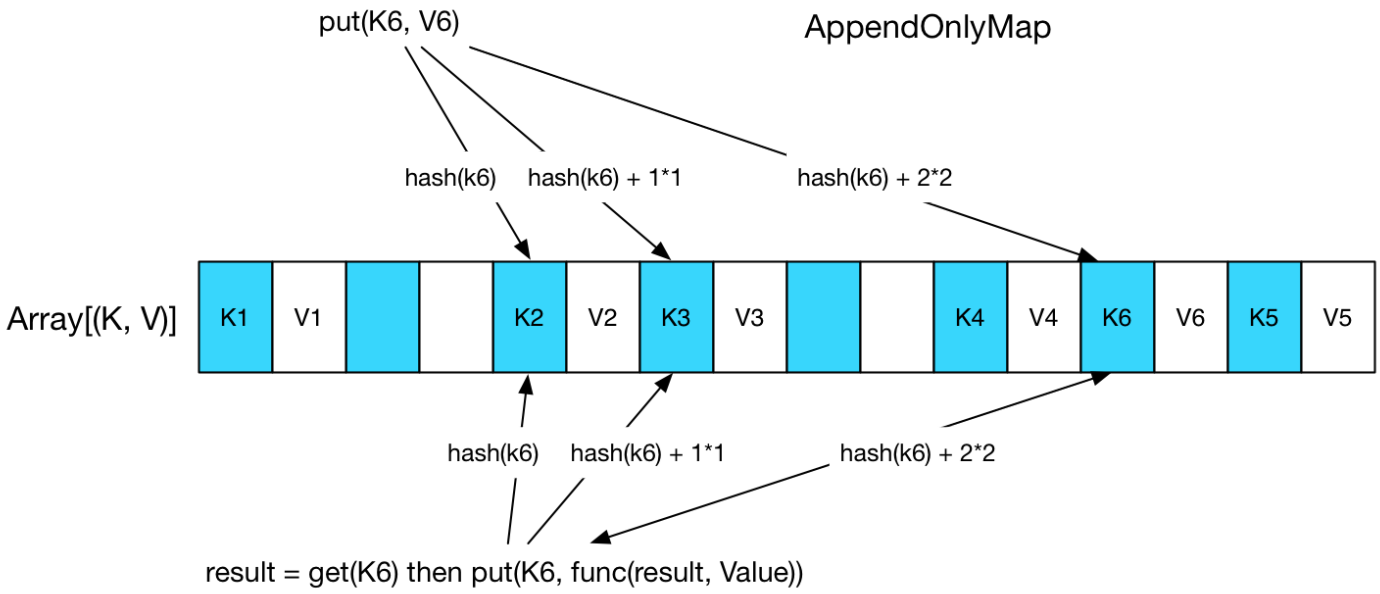
`coalesce()` would create a `ShuffleDependency`, but it actually does not need to aggregate the fetched records, so no hash map is needed.

HashMap in Shuffle Read

So as we have seen, hash map is a frequently used data structure in Spark's shuffle process. Spark has 2 versions of specialized hash map: in memory `AppendOnlyMap` and memory-disk hybrid `ExternalAppendOnlyMap`. Let's look at some details of these two hash map implementations.

`AppendOnlyMap`

The Spark documentation describes `AppendOnlyMap` as "A simple open hash table optimized for the append-only use case, where keys are never removed, but the value for each key may be changed". Its implementation is simple: allocate a big array of `Object`, as the following diagram shows. Keys are stored in the blue sections, and values are in the white sections.



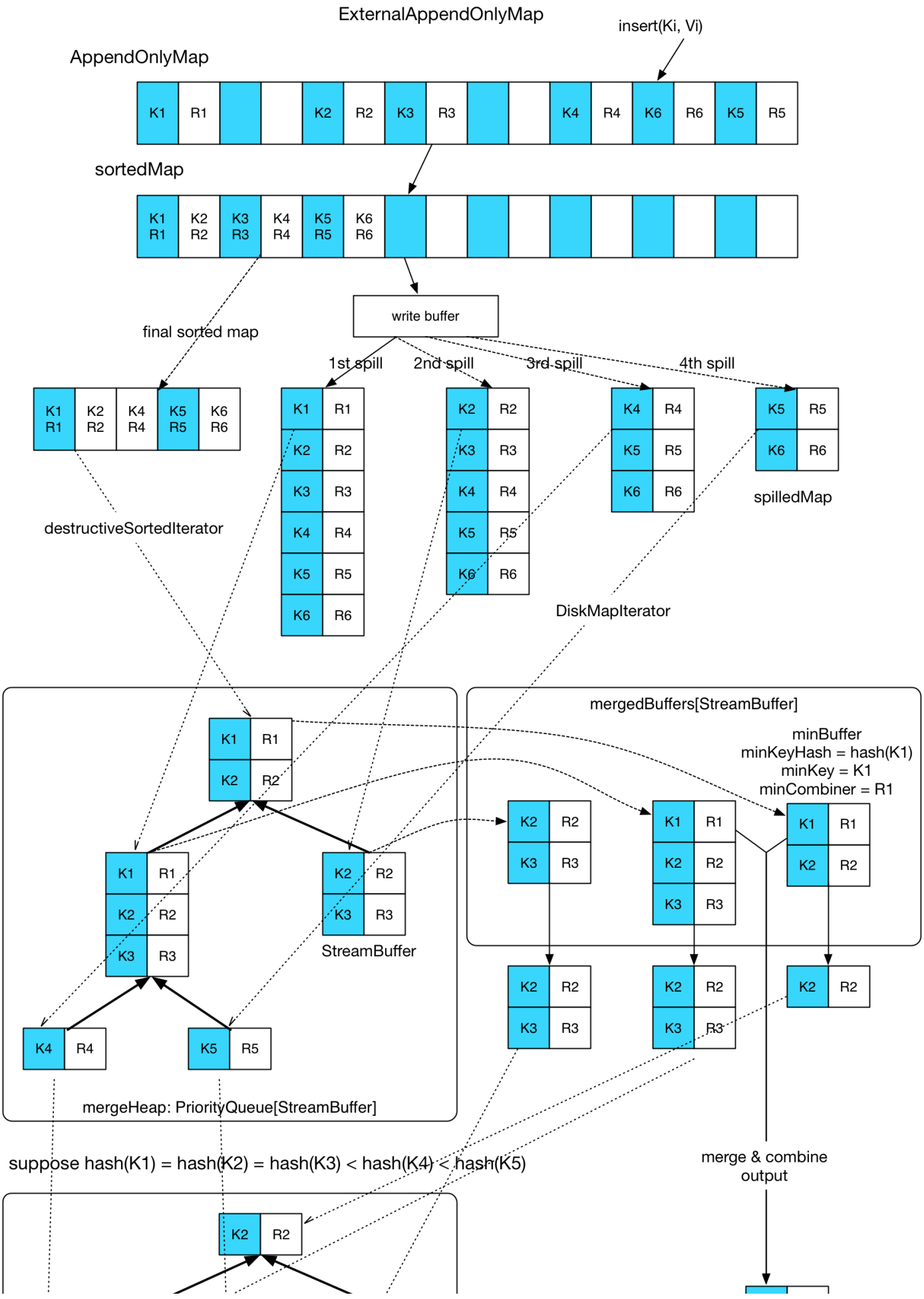
When a `put(K, V)` is issued, we locate the slot in the array by `hash(K)`. **If the position is already occupied, then quadratic probing technique is used to find the next slot.**. For the example in the diagram, `K6`, a third probing has found an empty slot after `K4`, then the value is inserted after the key. When `get(K6)`, we use the same technique to find the slot, get `V6` from the next slot, compute a new value, then write it to the position of `V6`.

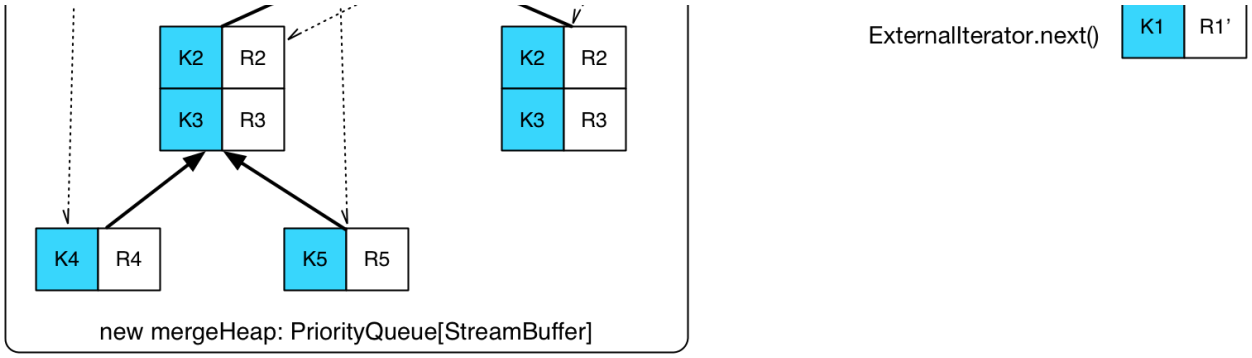
Iteration over the `AppendOnlyMap` is just a scan of the array.

If 70% of the allocated array is used, then it will grow twice as large. Keys will be rehashed and the positions re-organized.

There's a `destructiveSortedIterator(): Iterator[(K, V)]` method in `AppendOnlyMap`. It returns sorted key-value pairs. It's implemented like this: first compact all key-value pairs to the front of the array and make each key-value pair in a single slot. Then `Array.sort()` is called to sort the array. As its name indicates, this operation will destroy the structure.

`ExternalAppendOnlyMap`


ExternalAppendOnlyMap
insert(Ki, Vi)
AppendOnlyMap
K1 R1 K2 R2 K3 R3 K4 R4 K6 R6 K5 R5
sortedMap
K1 R1 K2 R2 K3 R3 K4 R4 K5 R5 K6 R6
write buffer
final sorted map
1st spill
2nd spill
3rd spill
4th spill
K1 R1 K2 R2 K4 R4 K5 R5 K6 R6
destructiveSortedIterator
spilledMap
DiskMapIterator
mergedBuffers[StreamBuffer]
minBuffer
minKeyHash = hash(K1)
minKey = K1
minCombiner = R1
StreamBuffer
mergeHeap: PriorityQueue[StreamBuffer]
suppose hash(K1) = hash(K2) = hash(K3) < hash(K4) < hash(K5)
merge & combine
output



Compared with `AppendOnlyMap`, the implementation of `ExternalAppendOnlyMap` is more sophisticated. Its concept is similar to the `shuffle-merge-combine-sort` process in Hadoop.

`ExternalAppendOnlyMap` holds an `AppendOnlyMap`. Incoming key-value pairs are inserted into the `AppendOnlyMap`. **When `AppendOnlyMap` is about to grow its size, we'll check the available memory space. If there's still enough space, the `AppendOnlyMap` doubles its size, otherwise all its key-value pairs are sorted and then spilled onto local disk (by using `destructiveSortedIterator()`).** In the diagram, there're 4 spills of this map. In each spill, a `spillMap` file will be generated and a new, empty `AppendOnlyMap` will be instantiated to receive incoming key-value pairs. In `ExternalAppendOnlyMap`, when a key-value pair is inserted, it gets aggregated only with the in memory part (the `AppendOnlyMap`). So it means when asked for the final result, a global merge-aggregate needs to be performed on all spilled maps and the in memory `AppendOnlyMap`.

Global merge-aggregate runs as follows. Firstly the in memory part (`AppendOnlyMap`) is sorted to a `sortedMap`. Then `DestructiveSortedIterator` (for `sortedMap`) or `DiskMapIterator` (for on disk `spillMap`) will be used to read a part of the key-value pairs into a `StreamBuffer`. Then the `StreamBuffer` is inserted into a `mergeHeap`. In each `StreamBuffer`, all records have the same `hash(key)`. Suppose that in the example, we have `hash(K1) == hash(K2) == hash(K3) < hash(K4) < hash(K5)`. As a result, the first 3 records of the first spilled map are read into the same `StreamBuffer`. The merge is simple: get `StreamBuffer`s with the same key hash using a heap, then put them into an `ArrayBuffer[StreamBuffer]`(`mergedBuffers`) for merge. The first inserted `StreamBuffer` is called `minBuffer`, the key of its first key-value pair is `minKey`. One merge operation will aggregate all KV pairs with `minKey` in the `mergedBuffer` and then output the result. When a merge operation in `mergedBuffer` is over, remaining KV pairs will return to the `mergeHeap`, and empty `StreamBuffer` will be replaced by a new read from in-memory map or on-disk spill.

There're still 3 points needed to be discussed:

- Available memory check. Hadoop allocates 70% of the memory space of a reducer for shuffle-sort. Similarly, Spark has `spark.shuffle.memoryFraction * spark.shuffle.safetyFraction` (defaults to 0.3 * 0.8) for `ExternalAppendOnlyMap`. **It seems that Spark is more conservative. Moreover, this 24% of memory space is shared by all reducers in the same executor.** An executor holds a `ShuffleMemoryMap: HashMap[threadId, occupiedMemory]` to monitor memory usage

of all `ExternalAppendOnlyMap`s in each reducer. Before an `AppendOnlyMap` grows, the total memory usage after the growth will be computed using the information in `ShuffleMemoryrMap`, to see if there's enough space. Also notice that the first 1000 records will not trigger the spill check.

- `AppendOnlyMap` size estimation. To know the size of an `AppendOnlyMap`, we can compute the size of every object referenced in the structure during each growth. But this takes too much time. Spark has an estimation algorithm with O(1) complexity. Its core concept is to see how the map size changes after the insertion and aggregation of a certain amount of records to estimate the structure size. Details are in `SizeTrackingAppendOnlyMap` and `SizeEstimator`.

- Spill process. Like the shuffle write, Spark creates a buffer when spilling records to disk. Its size is `spark.shuffle.file.buffer.kb`, defaulting to 32KB. Since the serializer also allocates buffers to do its job, there'll be problems when we try to spill lots of records at the same time. Spark limits the records number that can be spilled at the same time to `spark.shuffle.spill.batchSize`, with a default value of 10000.

Discussion

As we've seen in this chapter, Spark is way more flexible in the shuffle process compared to Hadoop's fixed shuffle-combine-merge-reduce model. It's possible in Spark to combine different shuffle strategies with different data structures to design an appropriate shuffle process based on the semantic of the actual transformation.

So far we've discussed the shuffle process in Spark without sorting as well as how this process gets integrated into the actual execution of the RDD chain. We've also talked about memory and disk issues and compared some details with Hadoop. In the next chapter we'll try to describe job execution from an inter-process communication perspective. The shuffle data location problem will also be mentioned.

Plus to this chapter, thers's the outstanding blog (in Chinese) by Jerry Shao, Deep Dive into Spark's shuffle implementation.

JerryLead/SparkInternals

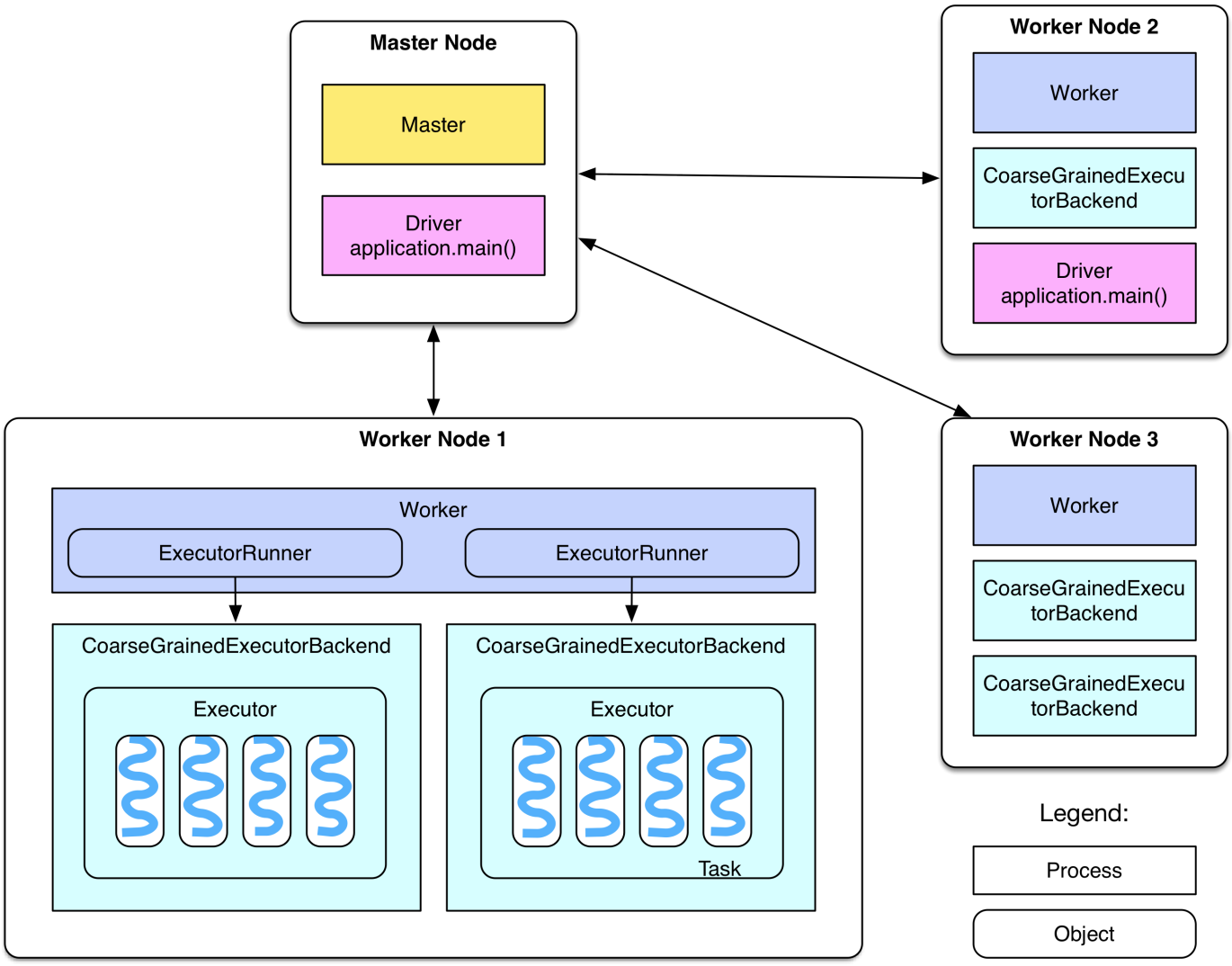
Architecture

We talked about spark jobs in chapter 3. In this chapter, we will talk about the architecture and how master, worker, driver and executors are coordinated to finish a job.

> Feel free to skip code if you prefer diagrams.

Deployment diagram

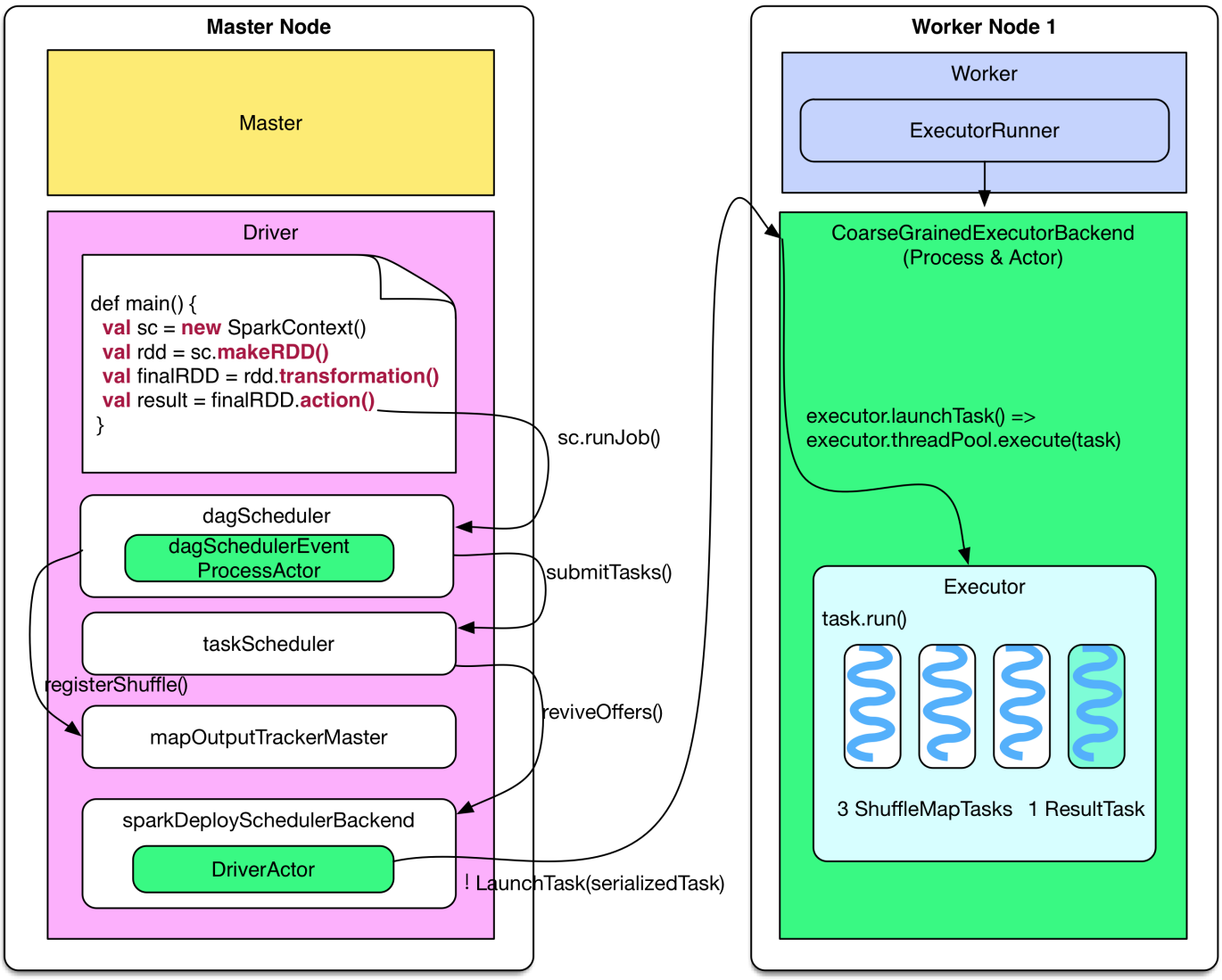
We have seen the following diagram in `overview` chapter.



Next, we will talk about some details about it.

Job submission

The diagram below illustrates how driver program (on master node) produces job, and then submits it to worker nodes.



Driver side behavior is equivalent to the code below:

```
finalRDD.action()
=> sc.runJob()

// generate job, stages and tasks
=> dagScheduler.runJob()
=> dagScheduler.submitJob()
=>   dagSchedulerEventProcessActor ! JobSubmitted
=> dagSchedulerEventProcessActor.JobSubmitted()
=> dagScheduler.handleJobSubmitted()
=> finalStage = newStage()
```

```
=>   mapOutputTracker.registerShuffle(shuffleId, rdd.partitions.size)
=> dagScheduler.submitStage()
=>   missingStages = dagScheduler.getMissingParentStages()
=> dagScheduler.subMissingTasks(readyStage)

// add tasks to the taskScheduler
=> taskScheduler.submitTasks(new TaskSet(tasks))
=> fifoSchedulableBuilder.addTaskSetManager(taskSet)

// send tasks
=> sparkDeploySchedulerBackend.reviveOffers()
=> driverActor ! ReviveOffers
=> sparkDeploySchedulerBackend.makeOffers()
=> sparkDeploySchedulerBackend.launchTasks()
=> foreach task
     CoarseGrainedExecutorBackend(executorId) ! LaunchTask(serializedTask)
```

Explanation:

When the following code is evaluated, the program will launch a bunch of driver communications, e.g. job's executors, threads, actors, etc.

```
val sc = new SparkContext(sparkConf)
```

This line defines the role of driver

Job logical plan

`transformation()` in driver program builds a computing chain (a series of `RDD`). In each `RDD`:

- `compute()` function defines the computation of records for its partitions
- `getDependencies()` function defines the dependency relationship across RDD partitions.

Job physical plan

Each `action()` triggers a job:

- During `dagScheduler.runJob()`, different stages are defined
- During `submitStage()`, `ResultTasks` and `ShuffleMapTasks` needed by the stage are produced, then they are packaged in `TaskSet` and sent to `TaskScheduler`. If `TaskSet` can be executed, tasks will be submitted to `sparkDeploySchedulerBackend` which will distribute tasks.

Task distribution

After `sparkDeploySchedulerBackend` gets `TaskSet`, the `Driver Actor` sends serialized tasks to `CoarseGrainedExecutorBackend Actor` on worker node.

Job reception

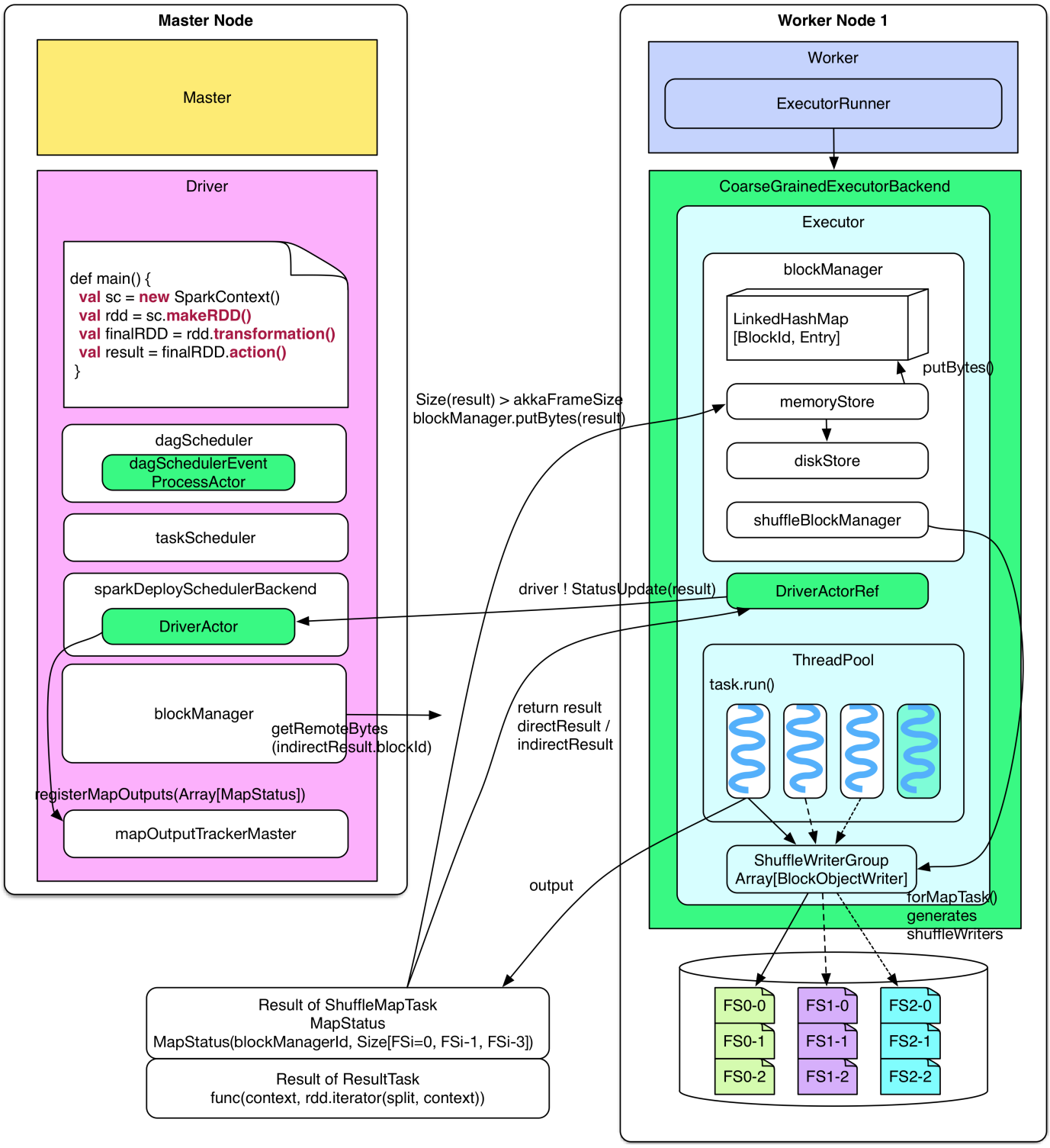
After receiving tasks, worker will do the following things:

```
coarseGrainedExecutorBackend ! LaunchTask(serializedTask)
=> executor.launchTask()
=> executor.threadPool.execute(new TaskRunner(taskId, serializedTask))
```

Executor packages each task into `taskRunner`, and picks a free thread to run the task. A `CoarseGrainedExecutorBackend` process has exactly one executor

Task execution

The diagram below shows the execution of a task received by worker node and how driver processes task results.



After receiving a serialized task, the executor deserializes it into a normal task, and then runs the task to get `directResult` which will be sent back to driver. It is noteworthy that data package sent from `Actor` can not be too big:

- If the result is too big (e.g. the one of `groupByKey`), it will be persisted to "memory + hard disk" and managed by `blockManager`. Driver will only get `indirectResult` containing the storage location. When result is needed, driver will fetch it via HTTP.
- If the result is not too big (less than `spark.akka.frameSize = 10MB`), then it will be directly sent to

driver.

Some more details about `blockManager`:

When `directResult > akka.frameSize`, the `memoryStore` of `BlockManager` creates a `LinkedHashMap` to hold the data stored in memory whose size should be less than `Runtime.getRuntime.maxMemory * spark.storage.memoryFraction(default 0.6)`. If `LinkedHashMap` has no space to save the incoming data, these data will be sent to `diskStore` which persists data to hard disk if the data `storageLevel` contains "disk"

```
In TaskRunner.run()
// deserialize task, run it and then send the result to
=> coarseGrainedExecutorBackend.statusUpdate()
=> task = ser.deserialize(serializedTask)
=> value = task.run(taskId)
=> directResult = new DirectTaskResult(ser.serialize(value))
=> if( directResult.size() > akkaFrameSize() )
       indirectResult = blockManager.putBytes(taskId, directResult,
MEMORY+DISK+SER)
   else
       return directResult
=> coarseGrainedExecutorBackend.statusUpdate(result)
=> driver ! StatusUpdate(executorId, taskId, result)
```

The results produced by `ShuffleMapTask` and `ResultTask` are different.

- `ShuffleMapTask` produces `MapStatus` containing 2 parts:
  - the `BlockManagerId` of the task's `BlockManager`: (executorId + host, port, nettyPort)
  - the size of each output `FileSegment` of a task
- `ResultTask` produces the execution result of the specified `function` on one partition e.g. The `function` of `count()` is simply for counting the number of records in a partition. Since `ShuffleMapTask` needs `FileSegment` for writing to disk, `OutputStream` writers are needed. These writers are produced and managed by `blockManger` of `shuffleBlockManager`

```
In task.run(taskId)
// if the task is ShuffleMapTask
=> shuffleMapTask.runTask(context)
=> shuffleWriterGroup = shuffleBlockManager.forMapTask(shuffleId,
partitionId, numOutputSplits)
=> shuffleWriterGroup.writers(bucketId).write(rdd.iterator(split, context))
```

```
=> return MapStatus(blockManager.blockManagerId,
Array[compressedSize(fileSegment)])

//If the task is ResultTask
=> return func(context, rdd.iterator(split, context))
```

A series of operations will be executed after driver gets a task's result:

`TaskScheduler` will be notified that the task is finished, and its result will be processed:

- If it is `indirectResult`, `BlockManager.getRemotedBytes()` will be invoked to fetch actual results.
  - If it is `ResultTask`, `ResultHandler()` invokes driver side computation on result (e.g. `count()` take `sum` operation on all ResultTask).
  - If it is `MapStatus` of `ShuffleMapTask`, then `MapStatus` will be put into `mapStatuses` of `mapOutputTrackerMaster`, which makes it more easy to be queried during reduce shuffle.
- If the received task on driver is the last task in the stage, then next stage will be submitted. If the stage is already the last one, `dagScheduler` will be informed that the job is finished.

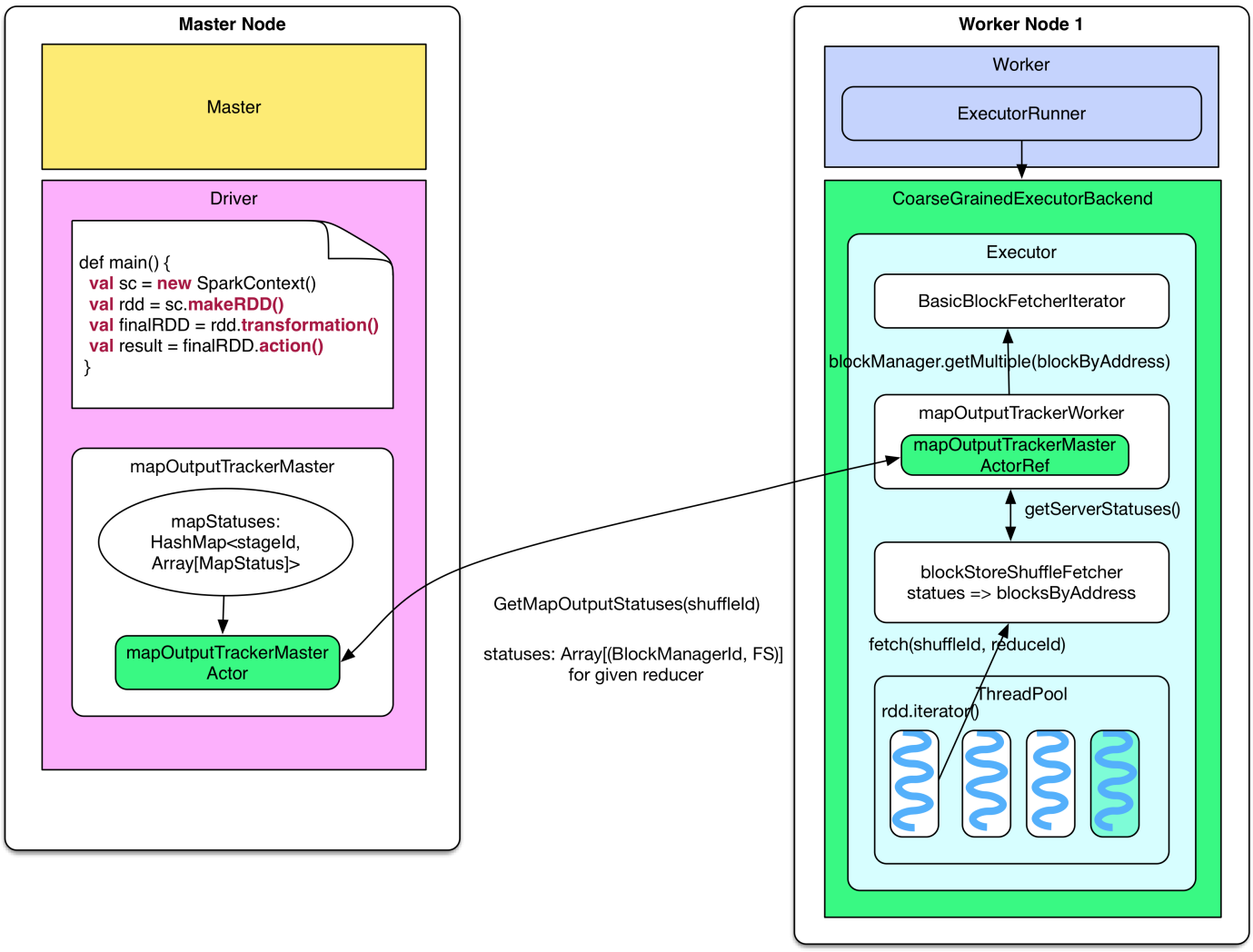
```
After driver receives StatusUpdate(result)
=> taskScheduler.statusUpdate(taskId, state, result.value)
=> taskResultGetter.enqueueSuccessfulTask(taskSet, tid, result)
=> if result is IndirectResult
      serializedTaskResult =
blockManager.getRemoteBytes(IndirectResult.blockId)
=> scheduler.handleSuccessfulTask(taskSetManager, tid, result)
=> taskSetManager.handleSuccessfulTask(tid, taskResult)
=> dagScheduler.taskEnded(result.value, result.accumUpdates)
=> dagSchedulerEventProcessActor ! CompletionEvent(result, accumUpdates)
=> dagScheduler.handleTaskCompletion(completion)
=> Accumulators.add(event.accumUpdates)

// If the finished task is ResultTask
=> if (job.numFinished == job.numPartitions)
      listenerBus.post(SparkListenerJobEnd(job.jobId, JobSucceeded))
=> job.listener.taskSucceeded(outputId, result)
=>    jobWaiter.taskSucceeded(index, result)
=>    resultHandler(index, result)

// If the finished task is ShuffleMapTask
=> stage.addOutputLoc(smt.partitionId, status)
```

```
=> if (all tasks in current stage have finished)
      mapOutputTrackerMaster.registerMapOutputs(shuffleId, Array[MapStatus])
      mapStatuses.put(shuffleId, Array[MapStatus]() ++ statuses)
=> submitStage(stage)
```

Shuffle read

In the preceding paragraph, we talked about task execution and result processing, now we will talk about how reducer (tasks needs shuffle) gets the input data. The shuffle read part in last chapter has already talked about how reducer processes input data.

How does reducer know where to fetch data ?

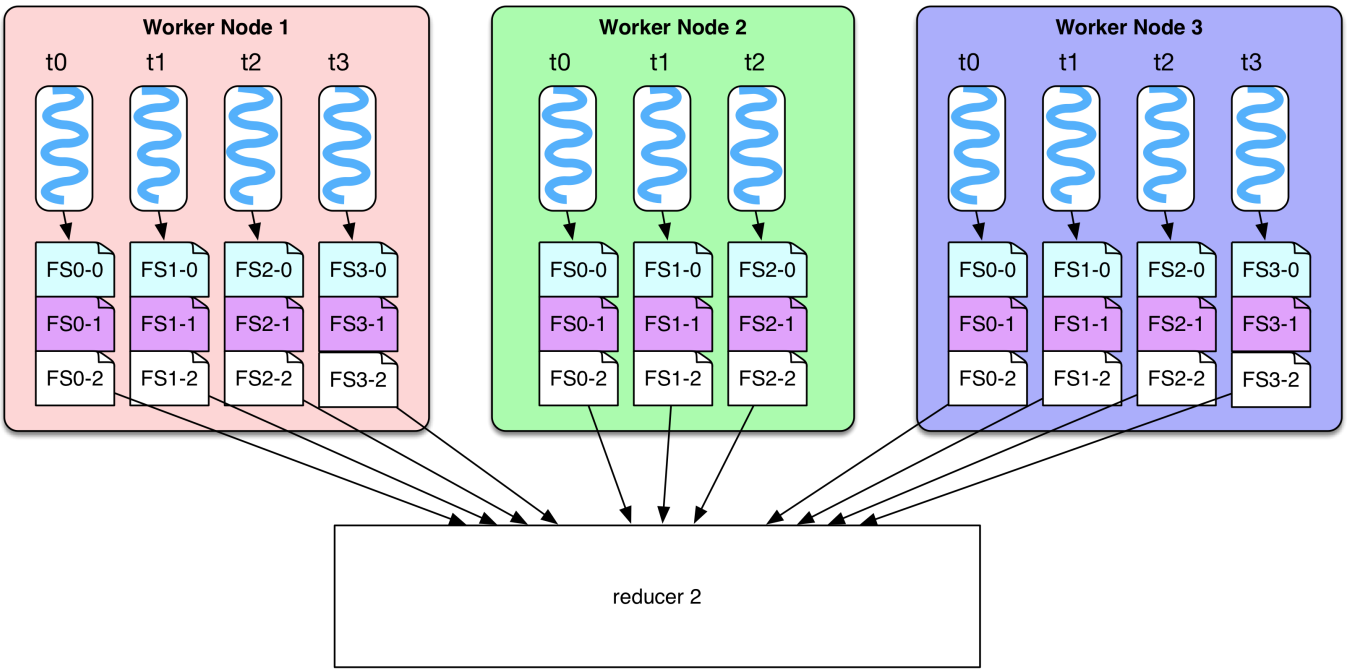


Reducer needs to know on which node the `FileSegments` produced by `ShuffleMapTask` of parent stage are. **This kind of information is sent to driver's `mapOutputTrackerMaster` when `ShuffleMapTask` is finished. The information is also stored in `mapStatuses: HashMap[stageId, Array[MapStatus]]`**. Given `stageId`, we can get`Array[MapStatus]` which contains information about `FileSegments` produced by `ShuffleMapTasks`. `Array(taskId)` contains

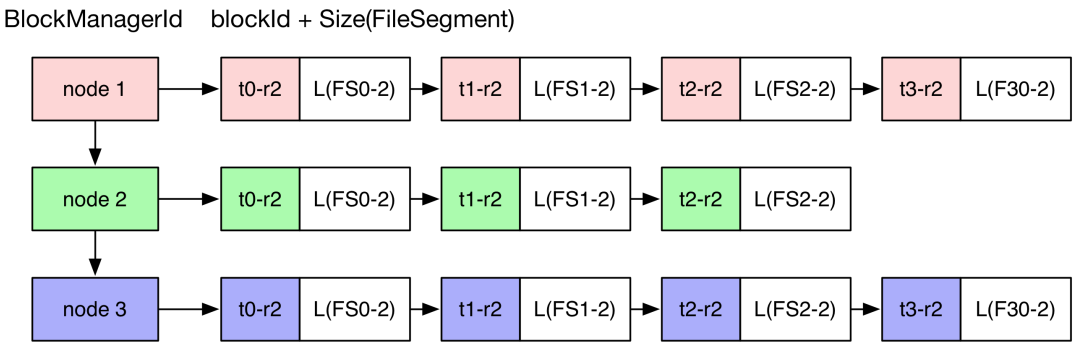
the location(`blockManagerId`) and the size of each `FileSegment`.

When reducer need fetch input data, it will first invoke `blockStoreShuffleFetcher` to get input data's location (`FileSegments`). `blockStoreShuffleFetcher` calls local `MapOutputTrackerWorker` to do the work. `MapOutputTrackerWorker` uses `mapOutputTrackerMasterActorRef` to communicate with `mapOutputTrackerMasterActor` in order to get `MapStatus`. `blockStoreShuffleFetcher` processes `MapStatus` and finds out where reducer should fetch `FileSegment` information, and then it stores this information in `blocksByAddress`. `blockStoreShuffleFetcher` tells `basicBlockFetcherIterator` to fetch `FileSegment` data.

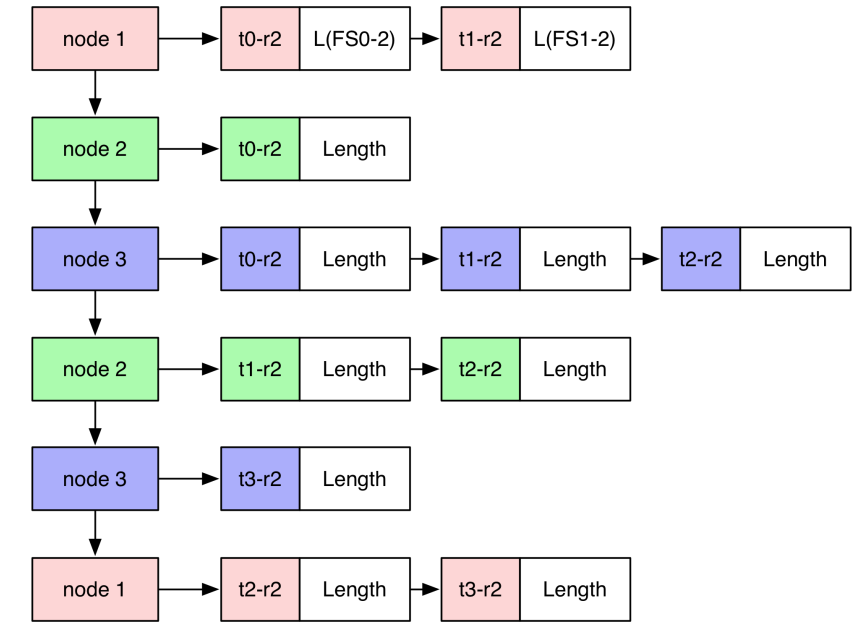
```
rdd.iterator()
=> rdd(e.g., ShuffledRDD/CoGroupedRDD).compute()
=> SparkEnv.get.shuffleFetcher.fetch(shuffledId, split.index, context, ser)
=> blockStoreShuffleFetcher.fetch(shuffleId, reduceId, context, serializer)
=> statuses = MapOutputTrackerWorker.getServerStatuses(shuffleId, reduceId)

=> blocksByAddress: Seq[(BlockManagerId, Seq[(BlockId, Long)])] =
compute(statuses)
=> basicBlockFetcherIterator = blockManager.getMultiple(blocksByAddress,
serializer)
=> itr = basicBlockFetcherIterator.flatMap(unpackBlock)
```



blocksByAddress: Array[(BlockManagerId, Array[(BlockId, Size(FileSegment))])]



fetchRequests



After `basicBlockFetcherIterator` has received the task of data retrieving, it produces several

`fetchRequest`s. **Each of them contains the tasks to fetch `FileSegment`s from several nodes.*

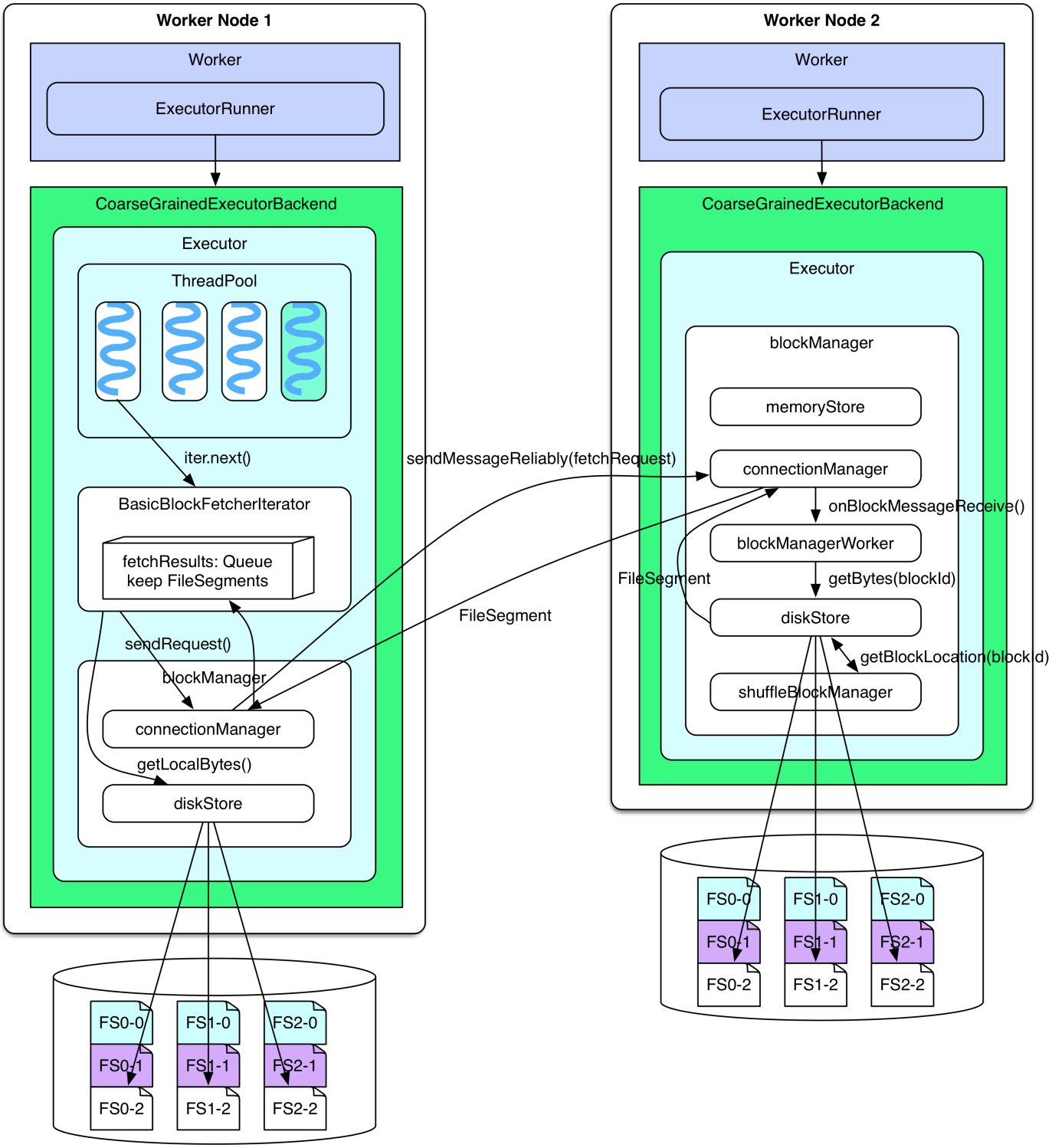
*According to the diagram above, we know that `reducer-2` needs to fetch `FileSegment`(FS)(in white) from 3 worker nodes. The global data fetching task can be represented by `blockByAddress`: 4 blocks from node 1, 3 blocks from node 2, and 4 blocks from node 3

In order to accelerate data fetching process, it makes sense to divide the global tasks into sub tasks(`fetchRequest`), then every task takes a thread to fetch data. Spark launches 5 parallel threads for each reducer (the same as Hadoop). Since the fetched data will be buffered into memory, one fetch is not able to take too much data (no more than `spark.reducer.maxMbInFlight = 48MB`). **Note that `48MB` is shared by the 5 fetch threads,** so each sub task should take no more than `48MB / 5 = 9.6MB`. In the diagram, on node 1, we have `size(FS0-2) + size(FS1-2) < 9.6MB, but size(FS0-2) + size(FS1-2) + size(FS2-2) > 9.6MB`, so we should break between `t1-r2` and `t2-r2`. As a result, we can see 2 `fetchRequest`s fetching data from node 1. **Will there be `fetchRequest` larger than 9.6MB?** The answer is yes. If one `FileSegment` is too large, it still needs to be fetched in one shot. Besides, if reducer needs some `FileSegment`s already existing on the local, it will do local read. At the end of shuffle read, it will deserialize fetched `FileSegment` and offer record iterators to `RDD.compute()`

```
In basicBlockFetcherIterator:

// generate the fetch requests
=> basicBlockFetcherIterator.initialize()
=> remoteRequests = splitLocalRemoteBlocks()
=> fetchRequests ++= Utils.randomize(remoteRequests)

// fetch remote blocks
=> sendRequest(fetchRequests.dequeue()) until Size(fetchRequests) >
maxBytesInFlight
=> blockManager.connectionManager.sendMessageReliably(cmId,
       blockMessageArray.toBufferMessage)
=> fetchResults.put(new FetchResult(blockId, sizeMap(blockId)))
=> dataDeserialize(blockId, blockMessage.getData, serializer)

// fetch local blocks
=> getLocalBlocks()
=> fetchResults.put(new FetchResult(id, 0, () => iter))
```

Some more details:

How does the reducer send `fetchRequest` to the target node? How does the target node process `fetchRequest`, read and send back `FileSegment` to reducer?



When `RDD.iterator()` meets `ShuffleDependency`, `BasicBlockFetcherIterator` will be called to fetch `FileSegment`s. `BasicBlockFetcherIterator` uses `connectionManager` of `blockManger` to send `fetchRequest` to `connectionManager`s on the other nodes. NIO is used for communication between `connectionManager`s. On the other nodes, for example, after `connectionManager` on worker node 2 receives a message, it will forward the message to `blockManager`. The latter uses `diskStore` to read `FileSegment`s requested by `fetchRequest` locally, they will still be sent back by `connectionManager`. If `FileConsolidation` is activated, `diskStore` needs the location of `blockId` given by `shuffleBolockManager`. If `FileSegment` is no

more than `spark.storage.memoryMapThreshold = 8KB`, then diskStore will put `FileSegment` into memory when reading it, otherwise, The memory mapping method in `FileChannel` of `RandomAccessFile` will be used to read `FileSegment`, thus large `FileSegment` can be loaded into memory.

When `BasicBlockFetcherIterator` receives serialized `FileSegments` from the other nodes, it will deserialize and put them in `fetchResults.Queue`. You may notice that **`fetchResults.Queue` is similar to `softBuffer` in `Shuffle detials` chapter.** If the `FileSegment`s needed by `BasicBlockFetcherIterator` are local, they will be found locally by `diskStore`, and put in `fetchResults`. Finally, reducer reads the records from `FileSegment` and processes them.

```
After the blockManager receives the fetch request

=> connectionManager.receiveMessage(bufferMessage)
=> handleMessage(connectionManagerId, message, connection)

// invoke blockManagerWorker to read the block (FileSegment)
=> blockManagerWorker.onBlockMessageReceive()
=> blockManagerWorker.processBlockMessage(blockMessage)
=> buffer = blockManager.getLocalBytes(blockId)
=> buffer = diskStore.getBytes(blockId)
=> fileSegment = diskManager.getBlockLocation(blockId)
=> shuffleManager.getBlockLocation()
=> if(fileSegment < minMemoryMapBytes)
     buffer = ByteBuffer.allocate(fileSegment)
   else
     channel.map(MapMode.READ_ONLY, segment.offset, segment.length)
```

Every reducer has a `BasicBlockFetcherIterator`, and one `BasicBlockFetcherIterator` could, in theory, hold 48MB of `fetchResults`. As soon as one `FileSegment` in `fetchResults` is read off, some `FileSegment`s will be fetched to fill that 48MB.

```
BasicBlockFetcherIterator.next()
=> result = results.task()
=> while (!fetchRequests.isEmpty &&
        (bytesInFlight == 0 || bytesInFlight + fetchRequests.front.size <=
maxBytesInFlight)) {
        sendRequest(fetchRequests.dequeue())
      }
=> result.deserialize()
```

Discussion

In terms of architecture design, functionalities and modules are pretty independent. `BlockManager` is well designed, but it seems to manage too many things (data block, memory, disk and network communication)

This chapter discussed how the modules of spark system are coordinated to finish a job (production, submission, execution, results collection, results computation and shuffle). A lot of code is pasted, many diagrams are drawn. More details can be found in source code, if you want.

If you also want to know more about `blockManager`, please refer to Jerry Shao's blog (in Chinese).

JerryLead/SparkInternals

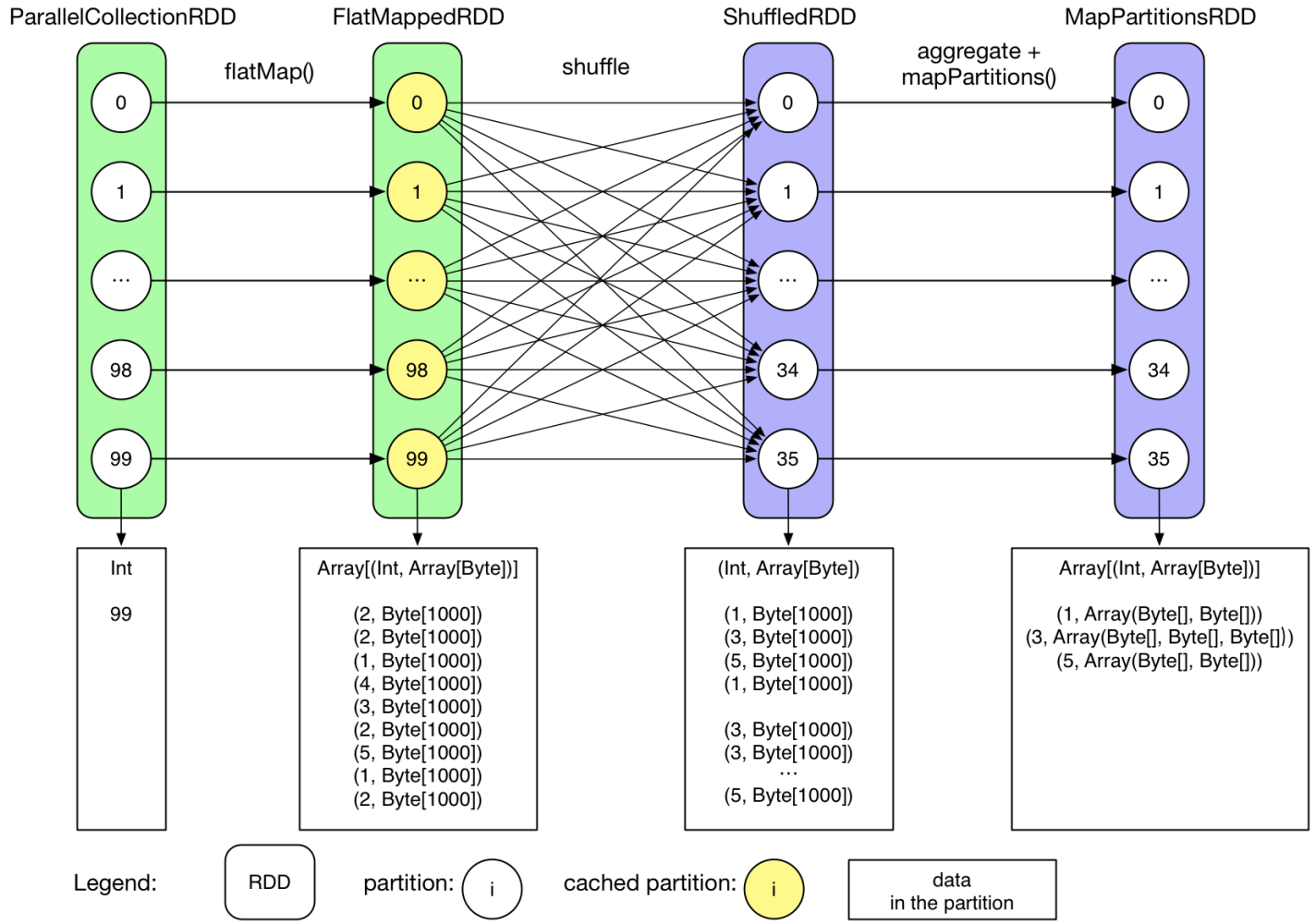
cache and checkpoint

`cache` (or `persist`) is an important feature which does not exist in Hadoop. It makes Spark much more fast to reuse a data set, e.g. iterative algorithm in machine learning, interactive data exploration, etc. Different from Hadoop MapReduce jobs, Spark's logical/physical plan can be very large, so the computing chain could be too long that it takes lots of time to compute RDD. If, unfortunately, some errors or exceptions occur during the execution of a task, the whole computing chain needs to be re-executed, which is considerably expensive. Therefore, we need to `checkpoint` some time-consuming RDDs. Thus, even if the following RDD goes wrong, it can continue with the data retrieved from checkpointed RDDs.

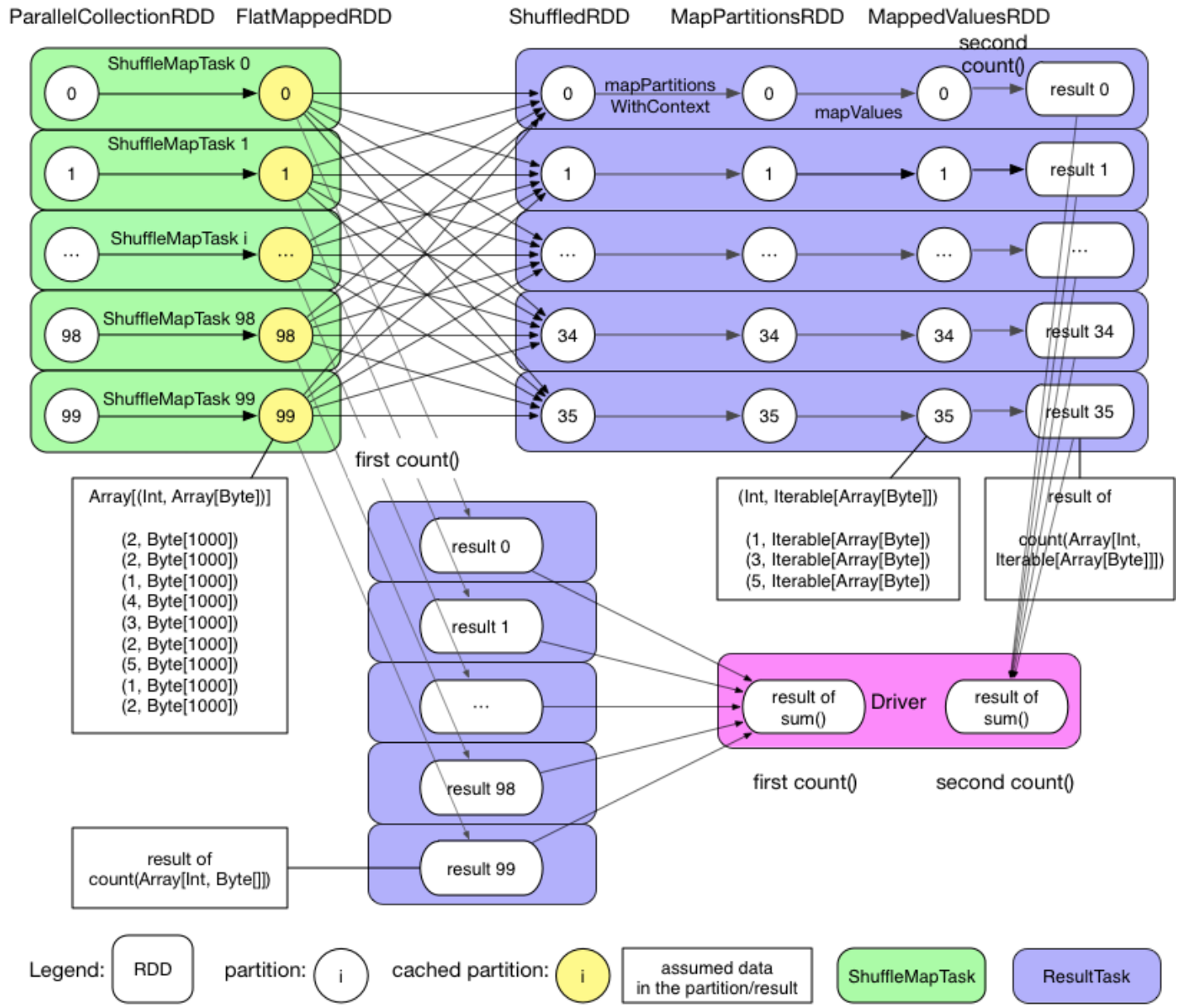
cache()

Let's take the `GroupByTest` in chapter Overview as an example, the `FlatMappedRDD` has been cached, so job 1 can just start with `FlatMappedRDD`, since `cache()` makes the repeated data get shared by jobs of the same application.

Logical plan:



Physical plan:



Q: What kind of RDD needs to be cached ?

Those which will be repeatedly computed and are not too large.

Q: How to cache an RDD ?

Just do a `rdd.cache()` in driver program, where `rdd` is the RDD accessible to users, e.g. RDD produced by `transformation()`, but some RDD produced by Spark (not user) during the execution of a transformation can not be cached by user, e.g. `ShuffledRDD`, `MapPartitionsRDD` during `reduceByKey()`, etc.

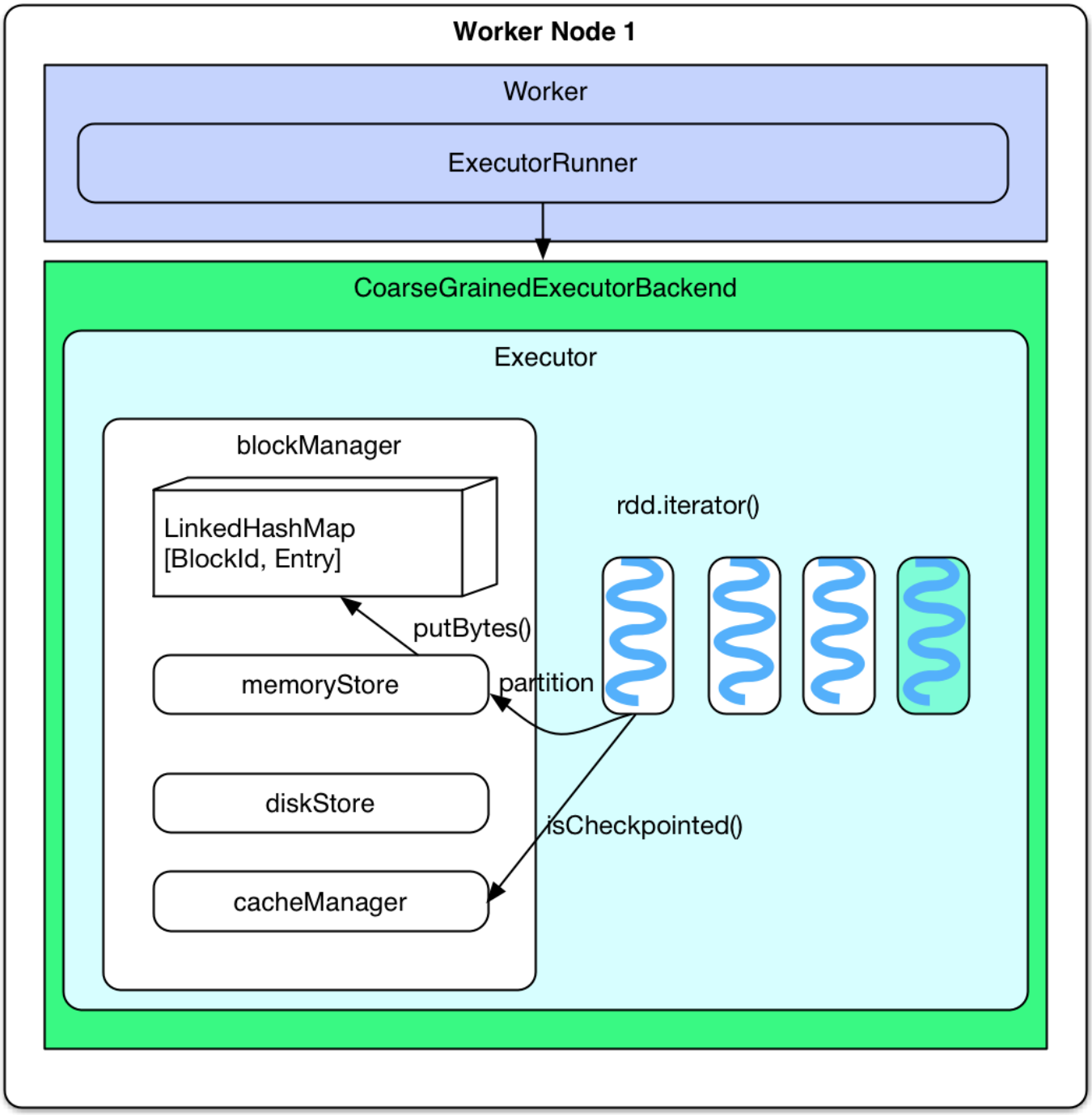
Q: How does Spark cache RDD ?

We can just make a guess. Intuitively, when a task gets the first record of an RDD, it will test if this RDD should be cached. If so, the first record and all the following records will be sent to `blockManager`'s `memoryStore`. If `memoryStore` can not hold all the records, `diskStore` will be used instead.

The implementation is similar to what we can guess, but the difference is that Spark will test whether

the RDD should be cached or not just before computing the first partition. If the RDD should be cached, the partition will be computed and cached into memory. `cache` only uses memory. Writing to disk is called `checkpoint`.

After calling `rdd.cache()`, `rdd` becomes `persistRDD` whose `storageLevel` is `MEMORY_ONLY`. `persistRDD` will tell `driver` that it needs to be persisted.



The above can be found in the following source code

```
rdd.iterator()
=> SparkEnv.get.cacheManager.getOrCompute(thisRDD, split, context,
```

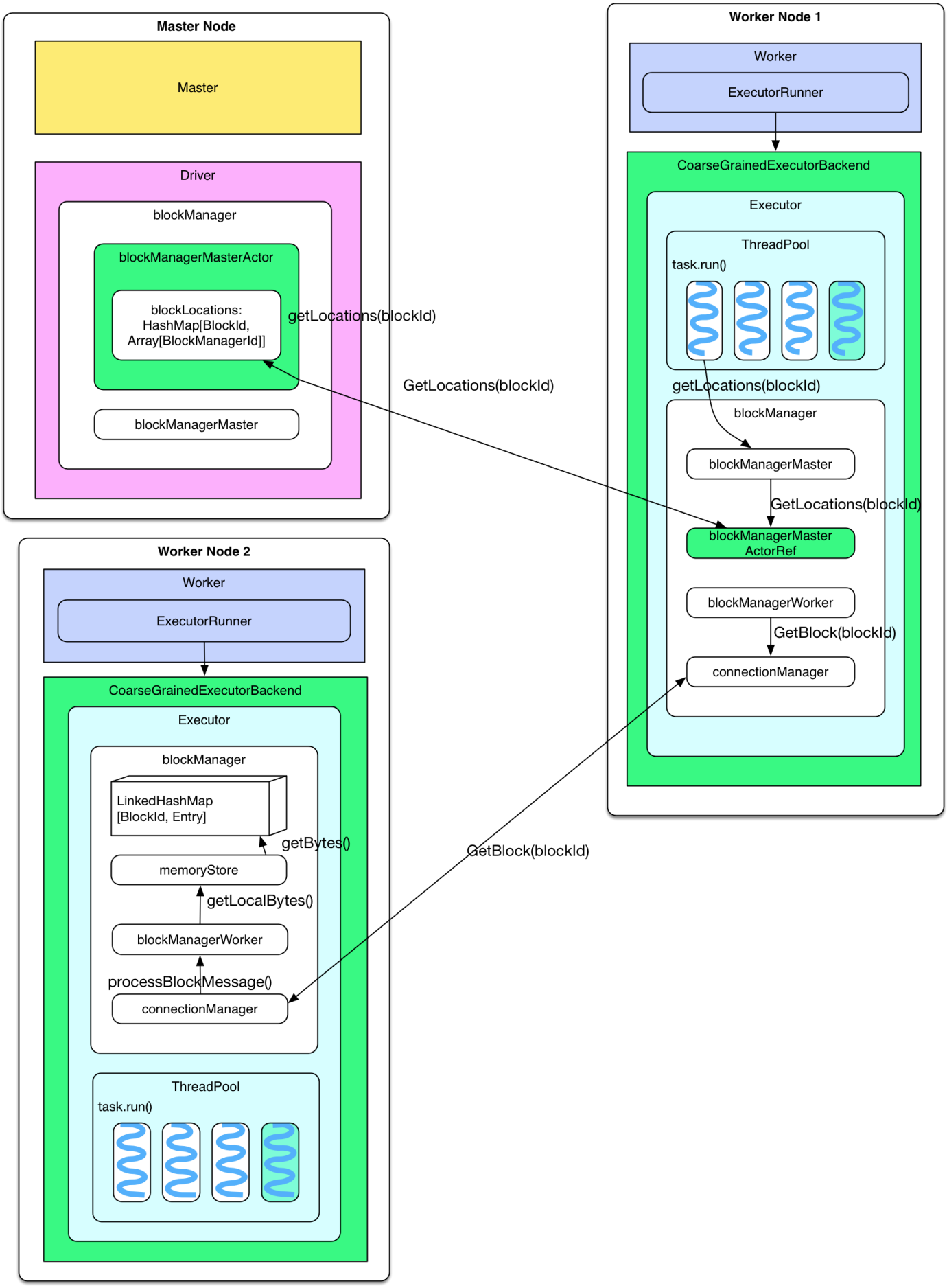
```
storageLevel)
=> key = RDDBlockId(rdd.id, split.index)
=> blockManager.get(key)
=> computedValues = rdd.computeOrReadCheckpoint(split, context)
      if (isCheckpointed) firstParent[T].iterator(split, context)
      else compute(split, context)
=> elements = new ArrayBuffer[Any]
=> elements ++= computedValues
=> updatedBlocks = blockManager.put(key, elements, tellMaster = true)
```

When `rdd.iterator()` is called to compute some partitions in the `rdd`, a `blockId` will be used to indicate which partition to cache, where `blockId` is of `RDDBlockId` type which is different from other data types in `memoryStore` like `result` of `task`. And then, partitions in `blockManger` will be checked to see whether they are checkpointed. If so, it will say that the task has already been executed, no more computation is needed for this partition. The `elements` of type `ArrayBuffer` will take all records of the partition from the check point. If not, the partition will be computed first, then all its records will be put into `elements`. Finally, `elements` will be submitted to `blockManager` for caching.

`blockManager` saves `elements` (partition) into `LinkedHashMap[BlockId, Entry]` inside `memoryStore`. If partition is bigger than `memoryStore`'s capacity (60% heap size), then just return by saying not being able to hold the data. If the size is ok, it will then drop some RDD partitions which was cached earlier in order to create space for the new incoming partitions. If the created space is enough, the new partition will be put into `LinkedHashMap`; if not, return by saying not enough space again. It is worth mentioning that the old partitions which belong to the RDD of the new partitions will not be dropped. Ideally, "first cached, first dropped".

Q: How to read cached RDD ?

When a cached RDD is being recomputed (in next job), `task` will read `blockManager` directly from `memoryStore`. Specifically, during the computation of some RDD partitions (by calling `rdd.iterator()`), `blockManager` will be asked whether they are cached or not. If the partition is cached in local, `blockManager.getLocal()` will be called to read data from `memoryStore`. If the partition was cached on the other nodes, `blockManager.getRemote()` will be called. See below:


Master Node
Master
Driver
blockManager
blockManagerMasterActor
blockLocations:
HashMap[BlockId,
Array[BlockManagerId]]
getLocations(blockId)
blockManagerMaster
GetLocations(blockId)
Worker Node 1
Worker
ExecutorRunner
CoarseGrainedExecutorBackend
Executor
ThreadPool
task.run()
getLocations(blockId)
blockManager
blockManagerMaster
GetLocations(blockId)
blockManagerMaster
ActorRef
blockManagerWorker
GetBlock(blockId)
connectionManager
Worker Node 2
Worker
ExecutorRunner
CoarseGrainedExecutorBackend
Executor
blockManager
LinkedHashMap
[BlockId, Entry]
getBytes()
memoryStore
getLocalBytes()
blockManagerWorker
processBlockMessage()
connectionManager
GetBlock(blockId)
ThreadPool
task.run()

the storage location of cached partition: the `blockManager` of the node on which a partition is cached will notify the `blockManagerMasterActor` on master by saying that an RDD partition is cached. This information will be stored in the `blockLocations: HashMap` of `blockMangerMasterActor`. When a task needs a cached RDD, it will send `blockManagerMaster.getLocations(blockId)` request to driver to get the partition's location, and the driver will lookup `blockLocations` to send back location info.

Read cached partition from the other nodes: a task gets cached partition's location info, and then it sends `getBlock(blockId)` request to the target node via `connectionManager`. The target node retrieves and sends back the cached partition from the `memoryStore` of the local `blockManager`.

Checkpoint

Q: What kind of RDD needs checkpoint ?

- the computation takes a long time
- the computing chain is too long
- depends too many RDDs

Actually, saving the output of `ShuffleMapTask` on local disk is also `checkpoint`, but it is just for data output of partition.

Q: When to checkpoint ?

As mentioned above, every time a computed partition needs to be cached, it is cached into memory. However, `checkpoint` does not follow the same principle. Instead, it waits until the end of a job, and launches another job to finish `checkpoint`. **An RDD which needs to be checkpointed will be computed twice; thus it is suggested to do a `rdd.cache()` before `rdd.checkpoint()`**. In this case, the second job will not recompute the RDD. Instead, it will just read cache. In fact, Spark offers `rdd.persist(StorageLevel.DISK_ONLY)` method, like caching on disk. Thus, it caches RDD on disk during its first computation, but this kind of `persist` and `checkpoint` are different, we will discuss the difference later.

Q: How to implement checkpoint ?

Here is the procedure:

RDD will be: [Initialized --> marked for checkpointing --> checkpointing in progress --> checkpointed]. In the end, it will be checkpointed.

Initialized

On driver side, after `rdd.checkpoint()` is called, the RDD will be managed by `RDDCheckpointData`. User should set the storage path for check point (on hdfs).

marked for checkpointing

After initialization, `RDDCheckpointData` will mark RDD `MarkedForCheckpoint`.

checkpointing in progress

When a job is finished, `finalRdd.doCheckpoint()` will be called. `finalRDD` scans the computing chain backward. When meeting an RDD which needs to be checkpointed, the RDD will be marked `CheckpointingInProgress`, and then the configuration files (for writing to hdfs), like core-site.xml, will be broadcast to `blockManager` of the other work nodes. After that, a job will be launched to finish `checkpoint`:

```
rdd.context.runJob(rdd, CheckpointRDD.writeToFile(path.toString,
broadcastedConf))
```

checkpointed

After the job finishes checkpoint, it will clean all the dependencies of the RDD and set the RDD to checkpointed. Then, **add a supplementary dependency and set the parent RDD as `CheckpointRDD`**. The `checkpointRDD` will be used in the future to read checkpoint files from file system and then generate RDD partitions

What's interesting is the following:

Two `RDD`s are checkpointed in driver program, but only the `result` (see code below) is successfully checkpointed. Not sure whether it is a bug or only that the downstream RDD will be intentionally checkpointed.

```
val data1 = Array[(Int, Char)]((1, 'a'), (2, 'b'), (3, 'c'),
    (4, 'd'), (5, 'e'), (3, 'f'), (2, 'g'), (1, 'h'))
val pairs1 = sc.parallelize(data1, 3)

val data2 = Array[(Int, Char)]((1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'))
val pairs2 = sc.parallelize(data2, 2)

pairs2.checkpoint

val result = pairs1.join(pairs2)
result.checkpoint
```

Q: How to read checkpointed RDD ?

`runJob()` will call `finalRDD.partitions()` to determine how many tasks there will be. `rdd.partitions()` checks if the RDD has been checkpointed via `RDDCheckpointData` which manages checkpointed RDD. If yes, return the partitions of the RDD (`Array[Partition]`). When `rdd.iterator()` is called to compute RDD's partition, `computeOrReadCheckpoint(split: Partition)` is also called to check if the RDD is checkpointed. If yes, the parent RDD's `iterator()`, a.k.a `CheckpointRDD.iterator()` will be called. `CheckpointRDD` reads files on file system to

produce RDD partition. **That's why a parent `CheckpointRDD` is added to checkpointed rdd trickly.**

Q: the difference between `cache` and `checkpoint` ?
Here is the an answer from Tathagata Das:

There is a significant difference between cache and checkpoint. Cache materializes the RDD and keeps it in memory (and/or disk). But the lineage（computing chain）of RDD (that is, seq of operations that generated the RDD) will be remembered, so that if there are node failures and parts of the cached RDDs are lost, they can be regenerated. However, **checkpoint saves the RDD to an HDFS file and actually forgets the lineage completely.** This allows long lineages to be truncated and the data to be saved reliably in HDFS, which is naturally fault tolerant by replication.

Furthermore, `rdd.persist(StorageLevel.DISK_ONLY)` is also different from `checkpoint`. Through the former can persist RDD partitions to disk, the partitions are managed by `blockManager`. Once driver program finishes, which means the thread where `CoarseGrainedExecutorBackend` lies in stops, `blockManager` will stop, the RDD cached to disk will be dropped (local files used by `blockManager` will be deleted). But `checkpoint` will persist RDD to HDFS or local directory. If not removed manually, they will always be on disk, so they can be used by the next driver program.

Discussion

When Hadoop MapReduce executes a job, it keeps persisting data (writing to HDFS) at the end of every task and every job. When executing a task, it keeps swapping between memory and disk, back and forth. The problem of Hadoop is that task needs to be re-executed if any error occurs, e.g. shuffle stopped by errors will have only half of the data persisted on disk, and then the persisted data will be recomputed for the next run of shuffle. Spark's advantage is that, when error occurs, the next run will read data from checkpoint, but the downside is that checkpoint needs to execute the job twice.

Example

```
package internals

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object groupByKeyTest {

   def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("GroupByKey").setMaster("local")
```

```
    val sc = new SparkContext(conf)
    sc.setCheckpointDir("/Users/xulijie/Documents/data/checkpoint")

    val data = Array[(Int, Char)]((1, 'a'), (2, 'b'),
                                  (3, 'c'), (4, 'd'),
                                  (5, 'e'), (3, 'f'),
                                  (2, 'g'), (1, 'h')
                                 )
    val pairs = sc.parallelize(data, 3)

    pairs.checkpoint
    pairs.count

    val result = pairs.groupByKey(2)

    result.foreachWith(i => i)((x, i) => println("[PartitionIndex " + i + "]
" + x))

    println(result.toDebugString)
   }
}
```

JerryLead/SparkInternals

Broadcast

As its name implies, broadcast means sending data from one node to all other nodes in the cluster. It's useful in many situations, for example we have a table in the driver, and other nodes need it as a lookup table. With broadcast we can send this table to all nodes and tasks will be able to do local lookups. Actually, it is challenging to implement a broadcast mechanism that is reliable and efficient. Spark's documentation says:

> Broadcast variables allow the programmer to keep a **read-only** variable cached on each **machine** rather than shipping a copy of it with **tasks**. They can be used, for example, to give every node a copy of a **large input dataset** in an efficient manner. Spark also attempts to distribute broadcast variables using **efficient** broadcast algorithms to reduce communication cost.

Why read-only?

This is a consistency problem. If a broadcasted variable can be mutated, once it's modified in some node, should we also update the copies on other nodes? If multiple nodes have updated their copies, how do we determine an order to synchronize these independent updates? There's also fault-tolerance problem coming in. To avoid all these tricky problems with data consistency, Spark only supports read-only broadcast variables.

Why broadcast to nodes but not tasks?

Each task runs inside a thread, and all tasks in a process belong to the same Spark application. So a single read-only copy in each node (executor) can be shared by all tasks.

How to use broadcast?

An example of a driver program:

```
val data = List(1, 2, 3, 4, 5, 6)
val bdata = sc.broadcast(data)

val rdd = sc.parallelize(1 to 6, 2)
val observedSizes = rdd.map(_ => bdata.value.size)
```

Driver uses `sc.broadcast()` to declare the data to broadcast. The type of `bdata` is `Broadcast`. `rdd.transformation(func)` uses `bdata` directly inside its function like a local variable.

How is broadcast implemented?

The implementation behind the broadcast feature is quite interesting.

Distribute metadata of the broadcast variable

Driver creates a local directory to store the data to be broadcasted and launches a `HttpServer` with access to the directory. The data is actually written into the directory when the broadcast is called (`val bdata = sc.broadcast(data)`). At the same time, the data is also written into driver's `blockManger` with a `StorageLevel` memory + disk. Block manager allocates a `blockId` (of type `BroadcastBlockId`) for the data. When a transformation function uses the broadcasted variable, the driver's `submitTask()` will serialize its metadata and send it along with the serialized function to all nodes. Akka system impose message size limits so we can not use it to broadcast the actual data.

> Why driver put the data in both local disk directory and block manager? The copy on the local disk directory is for the HttpServer, and the copy in block manager is to facilitate the usage of this data inside the driver program.

Then when the real data is broadcasted? When an executor deserializes the task it has received, it also gets the broadcast variable's metadata, in the form of a `Broadcast` object. It then calls the `readObject()` method of the metadata object (`bdata` variable). This method will first check the local block manager to see if there's already a local copy. If not, the data will be fetched from the driver. Once the data is fetched, it's stored in the local block manager for subsequent uses.

Spark has actually 2 different implementations of the data fetching.

HttpBroadcast

This method fetches data through an http connection between the executor and the driver.

Driver creates a `HttpBroadcast` object and it's this object's job to store the broadcast data into the driver's block manager. In the same time, as we described earlier, the data is written on the local disk, for example in a directory named `/var/folders/87/grpn1_fn4xq5wdqmxk31v0l00000gp/T/spark-6233b09c-3c72-4a4d-832b-6c0791d0eb9c/broadcast_0`.

> Driver and executor instantiate a `broadcastManger` object during initialization. The local directory is created by `HttpBroadcast.initialize()` method. This method also launches the http server.

The actual fetching is just data transmission between two nodes with an http connection.

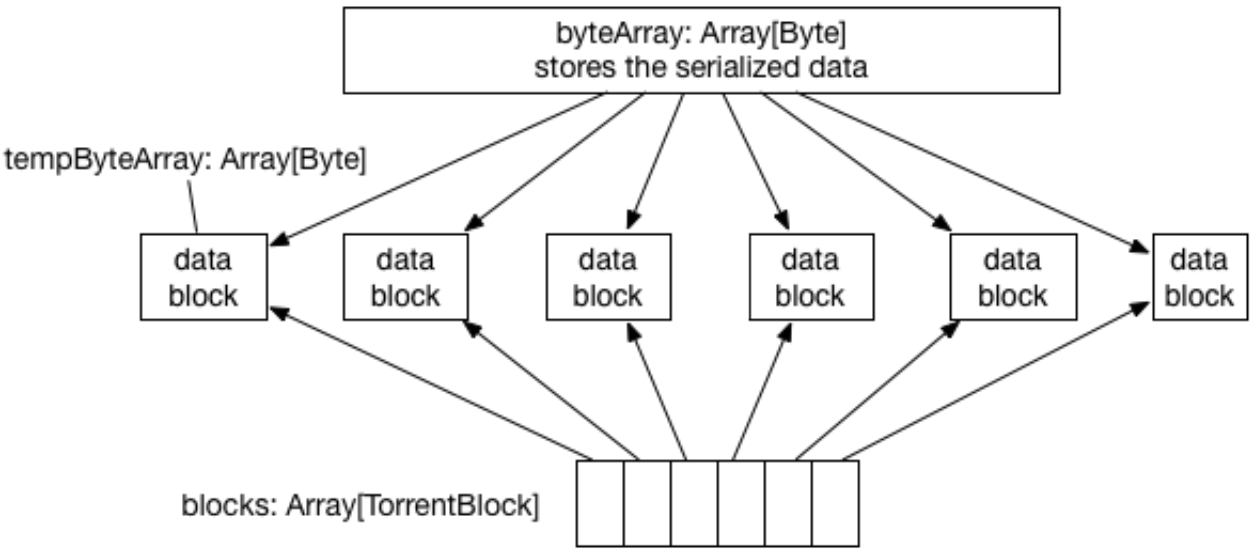
The problem of `HttpBroadcast` is network bandwidth bottleneck in the driver node since it sends data to all other worker nodes at the same time.

TorrentBroadcast

To solve the driver network bottleneck problem in `HttpBroadcast`, Spark introduced a new broadcast implementation called `TorrentBroadcast` which is inspired by BitTorrent. The basic concept of this method is to cut the broadcast data in blocks, and executors who have already fetched data blocks can themselves be the data source.

Unlike the data transfer in `HttpBroadcast`, `TorrentBroadcast` uses `blockManager.getRemote() => NIO ConnectionManager` to do the job. The actual sending and receiving process is quite similar with the cached rdd that we've talked about in the last chapter (check last diagram in CacheAndCheckpoint.

Let's see some details in `TorrentBroadcast`:



Driver

The driver serializes the data into `ByteArray` and then cut it into `BLOCK_SIZE` (defined by `spark.broadcast.blockSize = 4MB`) blocks. After the cut the original `ByteArray` will be collected but there's a temporary moment we have 2 copies of the data in memory.

After the cut, information about the blocks (called metadata) is stored in driver's block manager with a storage level at memory + disk. At this time, driver's `blockManagerMaster` will also be informed that the metadata is successfully stored. **This is an important step because blockManagerMaster can be accessed by all executors, meaning that block metadata has now become global data of the cluster.**

Driver then finishes its work by physically storing the data blocks under its block manager.

Executor

Upon receiving a serialized task, an executor deserializes it first. The deserialization also includes the broadcast metadata, whose type is `TorrentBroadcast`. Then its `TorrentBroadcast.readObject()` method is called. Similar to the general steps illustrated above, the local block manager will be checked

first to see if some data blocks have already been fetched. If not, executor will ask driver's `blockManagerMaster` for the data block's metadata. Then the BitTorrent process starts to fetch the data blocks.

BitTorrent process: an `arrayOfBlocks = new Array[TorrentBlock](totalBlocks)` is allocated locally to store the fetched data. `TorrentBlock` is a wrapper over a data block. The actual fetching order is randomized. For example if there's 5 blocks to fetch in total, the fetching order may be 3-1-2-4-5. Then the executor starts to fetch the data block one by one: local `blockManager` => local `connectionManager` => driver/other executor's blockManager => data. **Each fetched block is stored under local block manager and driver's `blockManagerMaster` is informed that this block has been successfully fetched.** As you'll guess, this is an important step because now all other nodes in the cluster know that there's a new data source for this block. If another node starts to fetch the same block, it'll randomly choose one data source. With more and more blocks being fetched, we'll have more data sources and the whole broadcast will be accelerated. There's a good illustration about BitTorrent on wikipedia.

When all the data blocks are fetched locally, a big `Array[Byte]` will be allocated to reconstitute the original broadcast data from blocks. Finally this array gets deserialized and is stored under local block manager. Notice that once we have the broadcast variable in local block manager, we can safely delete the fetched data blocks (which are also stored in local block manager).

One more question: what about broadcasting an RDD? In fact nothing bad will happen. This RDD will be evaluated in each executor so that each node has a copy of its result.

Discussion

Broadcasting shared variables is a very handy feature. In Hadoop we have the `DistributedCache` and it's used in many situations. For example, parameters of `-libjars` are sent to all nodes by using `DistributedCache`. However in Hadoop broadcasted data needs to be uploaded to HDFS first and it has no mechanism to share data between tasks in the same node. Say if some node needs to process 4 mappers coming from the same job, then the broadcast variable will be stored 4 times in this node (one copy in each mapper's working directory). An advantage of this approach is that by using HDFS we won't have the bottleneck problem since HDFS does the job of cutting data into blocks and distribute them across the cluster.

For Spark, broadcast cares about sending data to all nodes as well as letting tasks of the same node share data. Spark's block manager solves the problem of sharing data between tasks in the same node. Storing shared data in local block manager with a storage level at memory + disk guarantees that all local tasks can access the shared data, in this way we avoid storing multiple copies. Spark has 2 broadcast implementations. The traditional `HttpBroadcast` has the bottleneck problem around the driver node. `TorrentBroadcast` solves this problem but it starts slower since it only accelerate the broadcast after

some amount of blocks fetched by executors. Also in Spark, the reconstitution of original data from data blocks needs some extra memory space.

In fact Spark also tried an alternative called `TreeBroadcast`. Interested reader can check the technical report: Performance and Scalability of Broadcast in Spark.

In my opinion the broadcast feature can even be implemented by using multicast protocols. But multicast is UDP based, we'll need mechanisms on reliability in the application layer.