Anatomy of RDD

A deep dive into the RDD data structure

https://github.com/phatak-dev/anatomy-of-rdd

- Madhukara Phatak
- Big data consultant and trainer at datamantra.io
- Consult in Hadoop, Spark and Scala
- www.madhukaraphatak.com

Agenda

- What is RDD?
- Immutable and Distributed
- Partitions
- Laziness
- Caching
- Extending Spark API

What is RDD?

Resilient Distributed Dataset

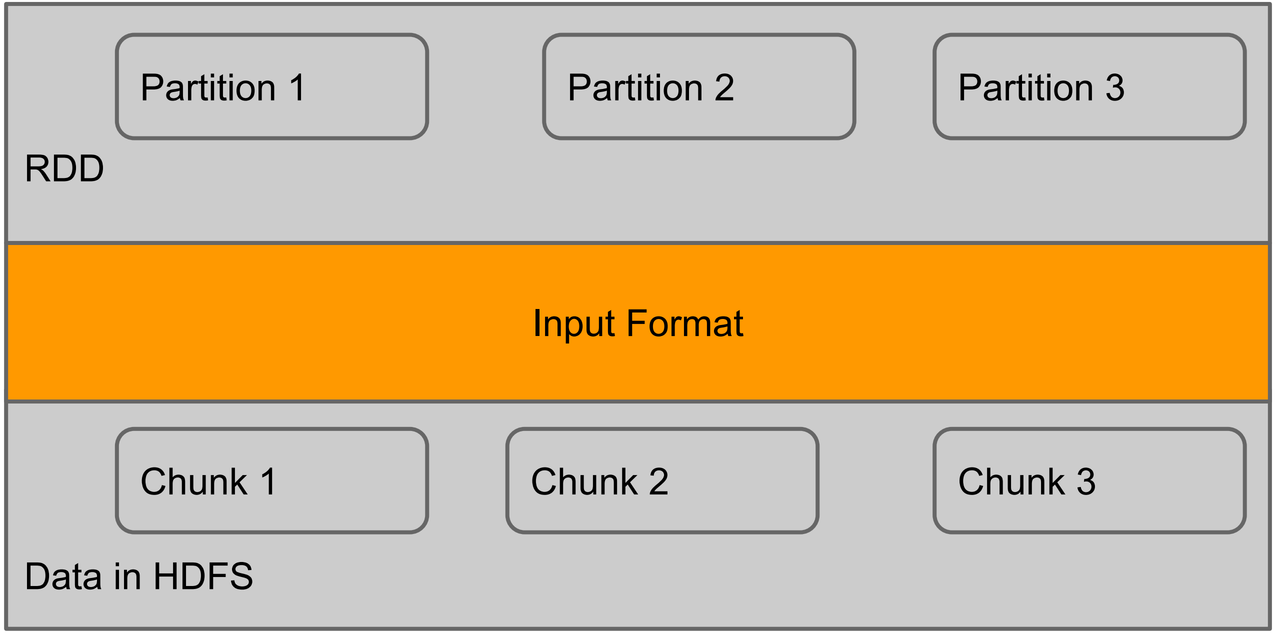
- A big collection of data with following properties
    - Immutable
    - Distributed
    - Lazily evaluated
    - Type inferred
    - Cacheable

Immutable and Distributed

Partitions

- Logical division of data
- Derived from Hadoop Map/Reduce
- All Input,Intermediate and output data will be represented as partitions
- Partitions are basic unit of parallelism
- RDD data is just collection of partitions

Partition from Input Data



Partition example

Partition and Immutability

- All partitions are immutable
- Every transformation generates new partition
- Partition immutability driven by underneath storage like HDFS
- Partition immutability allows for fault recovery

Partitions and Distribution

- Partitions derived from HDFS are distributed by default
- Partitions also location aware
- Location awareness of partitions allow for data locality
- For computed data, using caching we can distribute in memory also

Accessing partitions

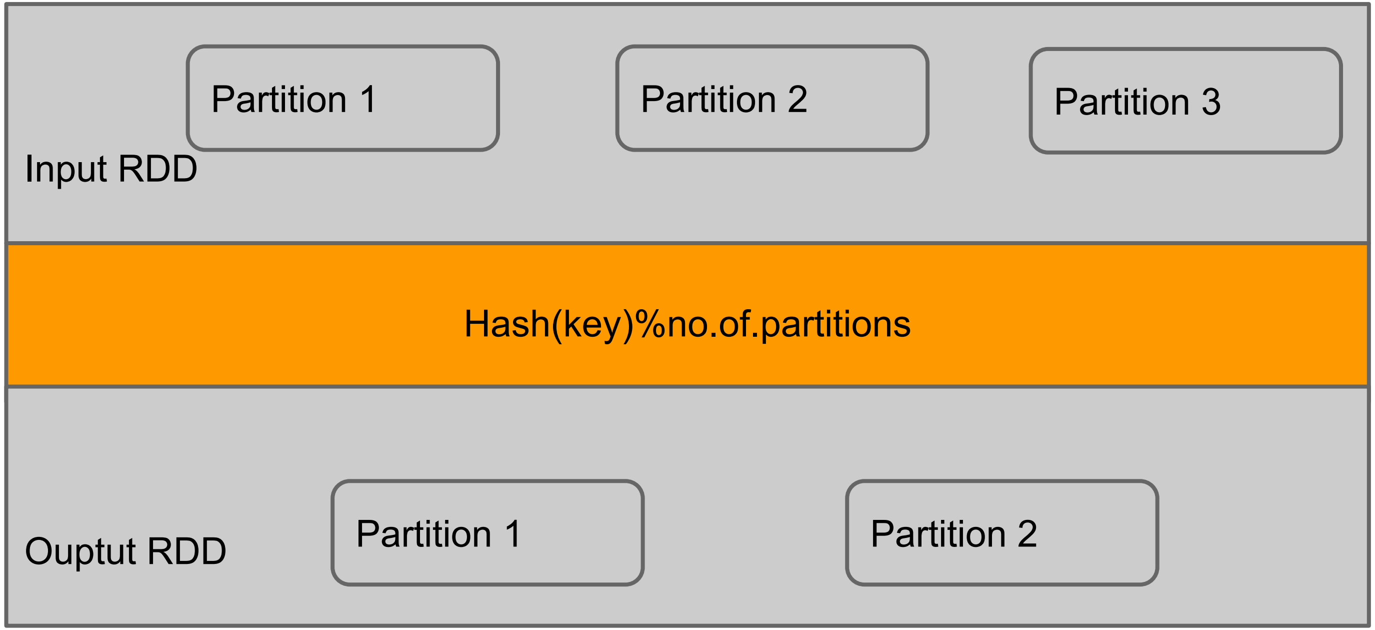
- We can access partition together rather single row at a time
- mapParititons API of RDD allows us that
- Accessing partition at a time allows us to do some partionwise operation which cannot be done by accessing single row.

Map partition example

Partition for transformed Data

- Partitioning will be different for key/value pairs that are generated by shuffle operation
- Partitioning is driven by partitioner specified
- By default HashPartitioner is used
- You can use your own partitioner also

Hash Partitioning



Hash partition example

Custom Partitioner

- Partition the data according to your data structure
- Custom partitioning allows control over no of partitions and the distribution of data across when grouping or reducing is done

Custom partition example

Look up operation

- Partitioning allows faster lookups
- Lookup operation allows to look up for a given value by specifying the key
- Using partitioner, lookup determines which partition look for
- Then it only need to look in that partition
- If no partition is specified, it will fallback to filter

Lookup example

Laziness

Parent(Dependency)

- Each RDD has access to it’s parent RDD
- Nil is the value of parent for first RDD
- Before computing it’s value, it always computes it’s parent
- This chain of running allows for laziness

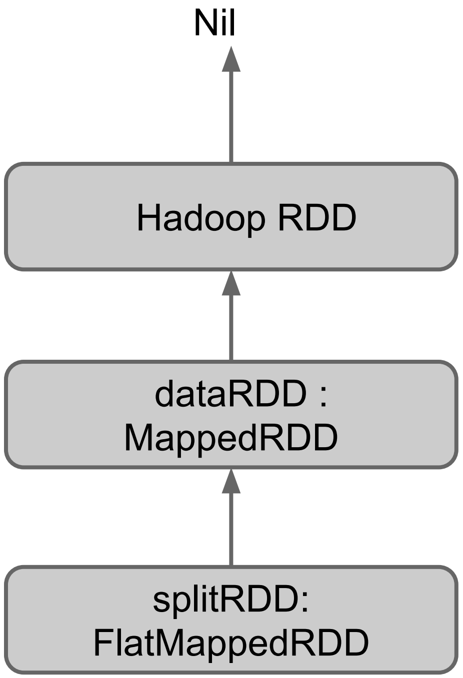
Sub classing

- Each spark operator, creates an instance of specific sub class of RDD
- map operator results in MappedRDD, flatMap in FlatMappedRDD etc
- Subclass allows RDD to remember the operation that is performed in the transformation

RDD transformations

```
val dataRDD =
sc.textFile(args
(1))

val splitRDD =
dataRDD.
flatMap(value =>
value.split(" ")
```



Laziness example

Compute

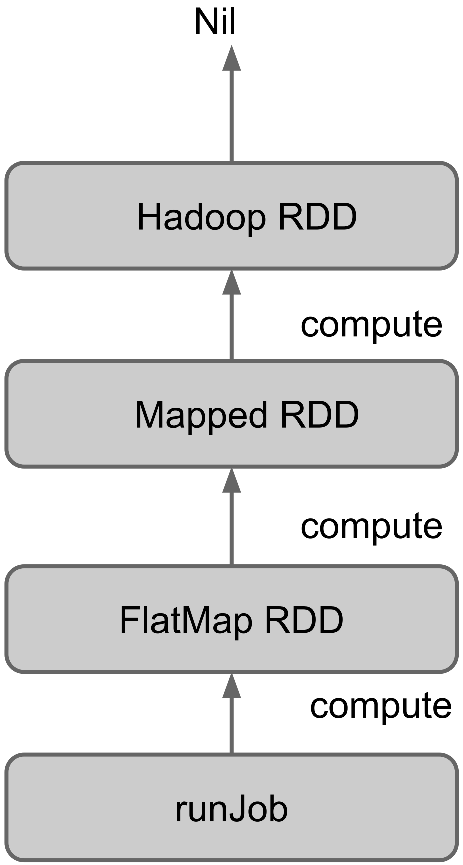
- Compute is the function for evaluation of each partition in RDD
- Compute is an abstract method of RDD
- Each sub class of RDD like MappedRDD, FilteredRDD have to override this method

RDD actions

val dataRDD = sc.
textFile(args(1))

val flatMapRDD =
dataRDD.flatMap
(value => value.split("
")

flatMapRDD.collect()



runJob API

- runJob API of RDD is the api to implement actions
- runJob allows to take each partition and allow you evaluate
- All spark actions internally use runJob api.

Run job example

Caching

- cache internally uses persist API
- persist sets a specific storage level for a given RDD
- Spark context tracks persistent RDD
- When first evaluates, partition will be put into memory by block manager

Block manager

- Handles all in memory data in spark
- Responsible for
  - Cached Data (BlockRDD)
  - Shuffle Data
  - Broadcast data
- Partition will be stored in Block with id (RDD.
  id, partition_index)

How caching works?

- Partition iterator checks the storage level
- if Storage level is set it calls

  cacheManager.getOrCompute(partition)

- as iterator is run for each RDD evaluation, its transparent to user

Caching example

Extending Spark API

Why?

- Domain specific operators
  - Allows developer to express domain specific calculation in cleaner way
  - Improves code readability
  - Easy to maintain
- Domain specific RDD’s
  - Better way of expressing domain data
  - Control over partitioning and distribution

DSL Example

- salesRecordRDD: RDD[SalesRecord]
- To make sum of sales
  - In plain spark
    - salesRecord.map(_.itemValue).sum
  - In our dsl
    - salesRecord.totalSales
- Our dsl hides internal representation and improves readability

How to Extend

- Custom operators to RDD
  - Domain specific operators to specific RDD’s
  - Uses scala implicit mechanism
  - Feels and works like built in operator
- Custom RDD
  - Extend RDD API to create our own RDD
  - Combined with RDD it’s very powerful

Implicits in Scala

- A way of extending Types on the fly
- Implicits also used to pass the parameters functions which are read from environment
- In our example, we just use the type extension facility
- All implicits are compile time checked.

Implicits example

Adding operators to RDD’s

- We use scala implicit facility, to add the custom operators on our RDD
- These operators only show up in our RDD’s
- All implicit conversion are handled by Scala not by Spark
- Spark internally use similar tricks for PairRDD’s

Custom operator Example

Extending RDD

- Extending RDD API allows to create our own custom RDD structure
- Custom RDD’s allows control over computation
- You can change partitions, locality and evaluation depending upon your requirement

Discount RDD example