BITS Pilani
Pilani Campus

# Parallel Programming with Spark

K Hari Babu
Department of Computer Science & Information Systems

BITS Pilani

Pilani Campus

# Spark

# MapReduce and Spark

- Hadoop MapReduce provides
  - locality-aware scheduling
  - fault tolerance
  - and load balancing
- MapReduce is deficient in
  - Iterative jobs: Many common machine learning algorithms apply a function repeatedly to the same dataset until it converges
    - each job must reload the datafrom disk, incurring a significant performance penalty
  - Interactive analytics: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig and Hive
    - Load a dataset into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs a separate MapReduce job and reads data from disk.

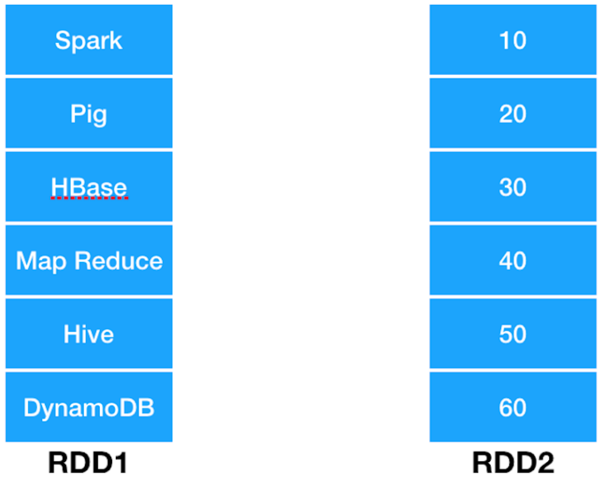
## Spark

- Spark supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce
- The main abstraction in Spark is that of are Resilient distributed dataset(RDD), a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost
- Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations

# Spark RDD

BITS Pilani

- RDD can be expanded as Resilient Distributed Data Sets
  - Resilient : fault-tolerant
  - Distributed: data which is residing across multiple interconnected nodes
  - Dataset: the dataset is a collection of partitioned data.
- Spark internally distributes the data contained in the RDDs among the clusters and parallelise the operation which you perform on them



## Spark PairedRDD

- Basic RDDs: treat all the data items as a single value
- Paired RDDs: the data items comprise of key-value pairs

| |
|---|
| Spark |
| Pig |
| HBase |
| Map Reduce |
| Hive |
| DynamoDB |

**RDD**

| | |
|---|---|
| India | 200 |
| Sri Lanka | 100 |
| Australia | 80 |
| Bangladesh | 75 |
| West Indies | 60 |
| South Africa | 50 |

**Paired RDD**

## RDD Persistence

- One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations
- Spark reuses them in other actions on that dataset (or datasets derived from it)
  - This allows future actions to be much faster (often by more than 10x).
  - Caching is a key tool for iterative algorithms and fast interactive use.

# RDD Persistence

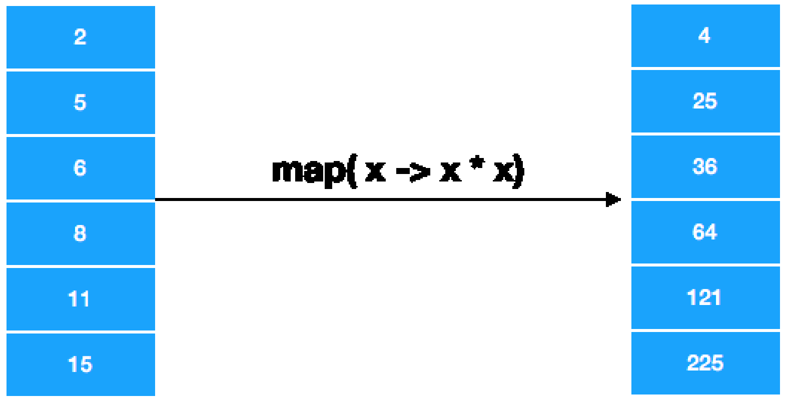
BITS Pilani

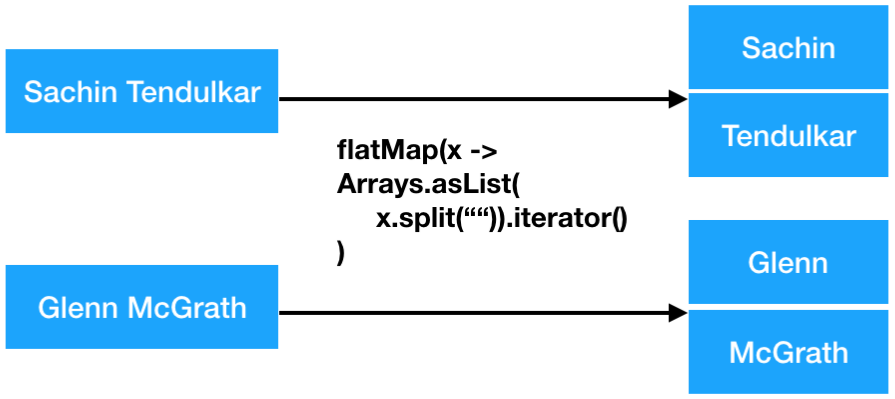
| Storage Level | Meaning |
|---|---|
| MEMORY_ONLY | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level. |
| MEMORY_AND_DISK | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed. |
| MEMORY_ONLY_SER (Java and Scala) | Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient |
| MEMORY_AND_DISK_SER (Java and Scala) | Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed. |
| DISK_ONLY | Store the RDD partitions only on disk. |
| MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc. | Same as the levels above, but replicate each partition on two cluster nodes. |

# Transformations

BITS Pilani

- Transformations are operations when applied on one RDD will result in another RDD
  - Map
  - Filter
  - flatMap





# Transformations on RDDs

| Transformation | Meaning |
|---|---|
| **map**(*func*) | Return a new distributed dataset formed by passing each element of the source through a function *func*. |
| **filter**(*func*) | Return a new dataset formed by selecting those elements of the source on which *func*returns true. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item). |
| **groupByKey**() | When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. |
| **reduceByKey**(*func*) | When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V |
| **sortByKey()** | When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument. |
| **join**(*otherDataset*, [*numPartitions*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin. |

# Actions on RDDs

BITS Pilani

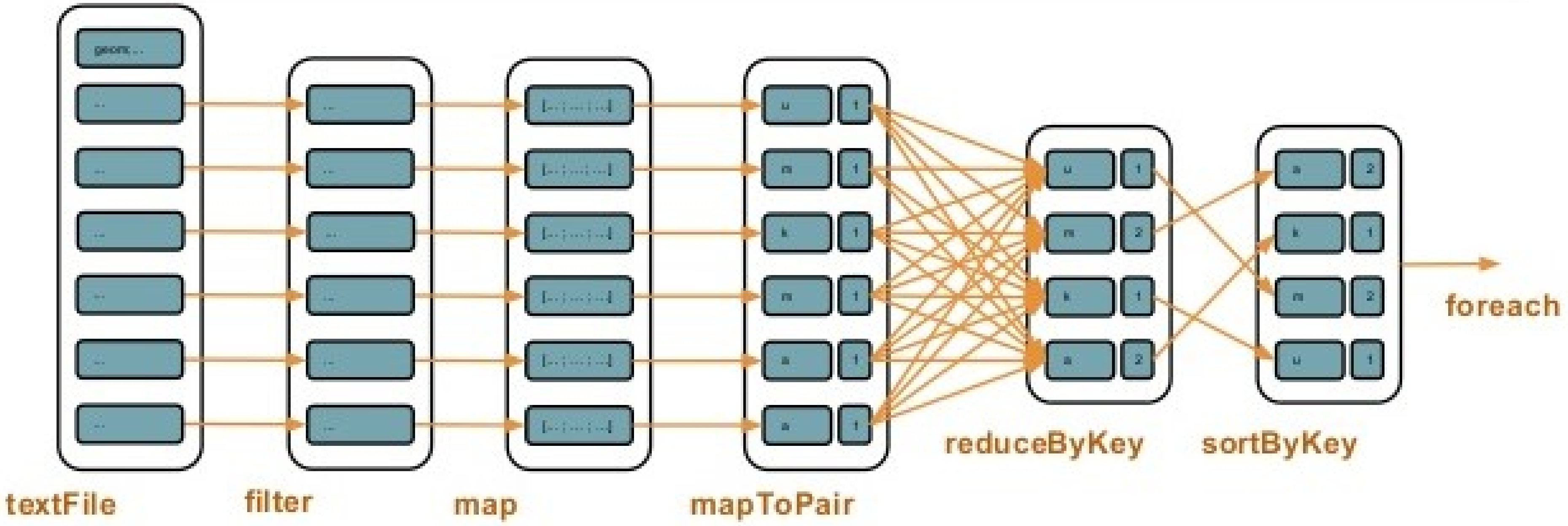
| Action | Meaning |
|---|---|
| reduce(func) | Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. |
| collect() | Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. |
| count() | Return the number of elements in the dataset. |
| first() | Return the first element of the dataset (similar to take(1)). |
| take(n) | Return an array with the first n elements of the dataset. |
| takeSample(withReplacement, num, [seed]) | Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed. |
| takeOrdered(n, [ordering]) | Return the first n elements of the RDD using either their natural order or a custom comparator. |
| saveAsTextFile(path) | Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. |
| saveAsSequenceFile(path) (Java and Scala) | Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). |
| saveAsObjectFile(path) (Java and Scala) | Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile(). |
| countByKey() | Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. |
| foreach(func) | Run a function func on each element of the dataset. |

# Transformations- example

BITS Pilani

```
JavaSparkContext sc = new JavaSparkContext("local", "arbres");

sc.textFile("data/arbresalignementparis2010.csv")
        .filter(line -> !line.startsWith("geom"))
        .map(line -> line.split(";"))
        .mapToPair(fields -> new Tuple2<String, Integer>(fields[4], 1))
        .reduceByKey((x, y) -> x + y)
        .sortByKey()
        .foreach(t -> System.out.println(t._1 + " : " + t._2));
```



Source: https://www.slideshare.net/aseigneurin/spark-alexis-seigneurin-english

## RDD lineage

BITS Pilani

- RDDs achieve fault tolerance through a notion of lineage:
  - if apartition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition
- RDDs contain information on how to compute themselves, including dependencies to other RDDs
  - In the event of a loss of a partition, partition can be recomputed
  - Fault Tolerance
- Effectively, RDDs create a directed acyclic graph of computations which is used to derive the latest RDD

# Spark- Word Count Example

```java
public final class JavaNetworkWordCount {
  private static final Pattern SPACE = Pattern.compile(" ");
  public static void main(String[] args) throws Exception {
    if (args.length < 2) {
      System.err.println("Usage: JavaNetworkWordCount <hostname> <port>");
      System.exit(1);
    }
    StreamingExamples.setStreamingLogLevels();
    // Create the context with a 1 second batch size
    SparkConf sparkConf = new SparkConf().setAppName("JavaNetworkWordCount");
    JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, Durations.seconds(1));

    // Create a JavaReceiverInputDStream on target ip:port and count the
    // words in input stream of \n delimited text (eg. generated by 'nc')
    // Note that no duplication in storage level only for running locally.
    // Replication necessary in distributed scenario for fault tolerance.
    JavaReceiverInputDStream<String> lines = ssc.socketTextStream(
            args[0], Integer.parseInt(args[1]), StorageLevels.MEMORY_AND_DISK_SER);
    JavaDStream<String> words = lines.flatMap(x -> Arrays.asList(SPACE.split(x)).iterator());
    JavaPairDStream<String, Integer> wordCounts = words.mapToPair(s -> new Tuple2<>(s, 1))
        .reduceByKey((i1, i2) -> i1 + i2);

    wordCounts.print();
    ssc.start();
    ssc.awaitTermination();
  }
}
```

## References

- https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf
- https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf
- RDD Programming Guide - Spark 3.3.2 Documentation (apache.org)

# Q&A

# Thank You