# Resilient Distributed Datasets and Spark

## Issues with MapReduce

The MapReduce framework and its implementations, such as Hadoop is a powerful distributed computing infrastructure that incorporates:

- **Distributed File System (DFS).** All files on which MapReduce jobs are run reside on a distributed file system, making them available to all nodes in the cluster. While management of the DFS is not part of the core MapReduce framework, the two go hand in hand.
- Distributed execution of two operations: Map - for transforming data, and Reduce - for aggregating data.
- Automated Shuffle operation that directs output of the Map to the correct reducers, and that prepares the input for the Reduce operations.
- A variety of solutions for empowering Map and Reduce operations: distributed cache, combiners, and so on.

At the same time, MapReduce has a number of important **deficiencies**:

- **Limitation on operations.** There are only two types of operations: Map and Reduce. While this is enough to implement distributed versions of any and all operations, some operations, such as **joins** can only be implemented in relatively awkward ways.
- **Iterative jobs.** MapReduce frameworks treat each individual MapReduce program/operation in isolation, by creating a separate data pipeline for each operation. Any data sharing between multiple operations (storage of data in RAM) is impossible. This creates overhead for each MapReduce job.
- **Interactive Analysis.** MapReduce jobs run in batch mode. While there are extensions of **MapReduce** that provide declarative querying facilities (Hive, Pig, etc.) that work by translating declarative SQL-like syntax into MapReduce jobs, the use of such facilities is still restricted by the **Iterative jobs** issue.

# Resilient Distributed Datasets (RDDs)

**Resilient Distributed Datastets (RDD)** are read-only collections of objects partitioned across a set of machines. The core properties of RDDs are:

- **Volatile storage.** RDDs are constructed to reside both in persistent storage and in RAM.
- **Resilience.** This property is defined as the ability to restore/rebuild the entire RDD, or any of its parts if a connection to a node in a distributed cluster is lost, taking away a portion of, or the full contents of an RDD.
- **Data Transformations.** RDDs are supplied with a collection of *data transformation* operations that create new RDDs.
- **Actions.** RDDs also can have *actions* performed on them. An *action* is an operation on an RDD that results in output generated and passed back to the user.
- **Lazy Evaluation.** RDDs achieve high performance due to *lazy evaluation* of transformations. That is, transformations on RDDs are *not evaluated* (but rather - are buffered) for as long as no action on an RDD needs to be taken. When an action operation is performed, all transformations applied to the original RDD(s) are performed at the same time, and are optimized.

  Contrast it with MapReduce's *eager evaluation* of `Map` and `Reduce` operations.
- **Rich set of operations.** The sets of both *transformations* and *actions* on RDDs are rich, implementing traditional *relational algebra* operations of *selection* (fitering), *projection*, *duplicate elimination*, *join*, as well as traditional set operations as *language primitives*.

  Contrast it with the efforts required to express a join or a duplicate elimination operation in MapReduce framework.

# Transformations

The following transformation operations were originally defined on RDDs. Note, current implementation of Spark allows for additional transformations.

Below, let $T$ and $U$ represent *types of objects in RDDs*, $K$ represent keys in key-value pairs, $V$ and $W$ represent values in key-value pairs.

**map().** The RDD `map()` operation is a "classic" `map()` that takes as input a function $f(.)$, and applies this function to each element of the RDD, producing one object in the output per each input object. The description is as follows:

$$map(f : T \Longrightarrow U)$$
$$\texttt{RDD[T]} \Longrightarrow \texttt{RDD[U]}$$

**flatMap().** This is the analog of the `map()` method in the MapReduce framework. MapReduce's `map()` can emit multiple key-value pairs per given input. This is represented by making the input function $f()$ for `flatMap()` map input objects into sequences of output objects:

$$flatMap(f : T \Longrightarrow Sequence[U])$$
$$\texttt{RDD[T]} \Longrightarrow \texttt{RDD[U]}$$

**mapValues().** Another version of `map()`, used when the input RDD consists of Key-Value pairs, and when only the value needs to be transformed. The key reason why this transformation exists as a separate operation, is because it can be performed *while preserving the partition of the RDD.*

$$mapValues(f : V \Longrightarrow W)$$

$$\texttt{RDD}[(\texttt{K}, \texttt{V})] \Longrightarrow \texttt{RDD}[(\texttt{K}, \texttt{W})]$$

**filter().** This is the direct implementation of the relational algebra *selection* operation. `filter()` operation takes as input a function $f$ that for each object returns either `True` (object is preserved in the new RDD) or `False` (object is removed from the new RDD).

$$filter(f : T \Longrightarrow \texttt{Boolean})$$
$$\texttt{RDD}[\texttt{T}] \Longrightarrow \texttt{RDD}[\texttt{T}]$$

**sample().** This transformation (deterministically) selects a sample of the input RDD. The input is the fraction of the RDD to put into the output.

$$sample(fraction : \texttt{Float})$$
$$\texttt{RDD}[\texttt{T}] \Longrightarrow \texttt{RDD}[\texttt{T}]$$

**groupByKey().** This transformation is applied to RDDs that consist of Key-Value pairs. For each unique Key, all values associated with it are placed into a sequence.

$$groupByKey()$$
$$\texttt{RDD}[(\texttt{K}, \texttt{V})] \Longrightarrow \texttt{RDD}[(\texttt{K}, \texttt{Sequence}[\texttt{V}])]$$

**reduceByKey().** This is the *transformation* version of the reduce operation. The result of this transformation is a new RDD (lazily evaluated), in which values belonging to the same key have been aggregated. The input is a function that takes two values and output one value (basically the "rolling" aggregation/reduction function).

$$reduceByKey(f : (V, V) \Longrightarrow V)$$
$$\texttt{RDD}[(\texttt{K}, \texttt{V})] \Longrightarrow \texttt{RDD}[(\texttt{K}, \texttt{V})]$$

**union().** The set union operation. The output RDD is the union of the two input RDDs.

$$union()$$
$$\texttt{RDD}[\texttt{T}], \texttt{RDD}[\texttt{T}] \Longrightarrow \texttt{RDD}[\texttt{T}]$$

**join().** A join-by-key (i.e., equijoin) version of the relational algebra *join* operation. Applied to RDDs that consist of Key-Value pairs. For each pair of objects from two different RDDs that share a key, an output object combining their values under the same key is created.

$$join()$$
$$\texttt{RDD}[(\texttt{K}, \texttt{V})], \texttt{RDD}[(\texttt{K}, \texttt{W})] \Longrightarrow \texttt{RDD}[(\texttt{K}, (\texttt{V}, \texttt{W}))]$$

**cogroup().** Another operation to combine information from two RDDs into a single one. Unlike join, where the combination is done on an object by object basis, `cogroup()`, applied to a pair of RDDs containing Key-Value pairs produces, for each Key, a single object containing two lists (sequences) of values: one from each of the input RDDs.

$$cogroup()$$
$$\texttt{RDD}[(\texttt{K},\texttt{V})],\texttt{RDD}[(\texttt{K},\texttt{W})] \Longrightarrow \texttt{RDD}[(\texttt{K},(\texttt{Sequence}[\texttt{V}],\texttt{Sequence}[\texttt{W}]))]$$

**crossporduct().** A cartesian product of two RDDs producing an object in the output for each pair of input objects from two input RDDs.

$$crossproduct()$$
$$\texttt{RDD}[\texttt{T}],\texttt{RDD}[\texttt{U}] \Longrightarrow \texttt{RDD}[\texttt{T},\texttt{U}]$$

**sort().** Sort the objects in the RDD based on a specific comparator.

$$sort(c : \texttt{Comparator}[\texttt{K}])$$
$$\texttt{RDD}[(\texttt{K},\texttt{V})] \Longrightarrow \texttt{RDD}[(\texttt{K},\texttt{V})]$$

**partitionBy().** Create a partition of the RDD (distribute portions of the RDD to different nodes) based on the input partition criterion.

$$partitionBy(p : \texttt{Partitioner}[\texttt{K}])$$
$$\texttt{RDD}[(\texttt{K},\texttt{V})] \Longrightarrow \texttt{RDD}[(\texttt{K},\texttt{V})]$$

# Actions

The following *actions* were initially defined on RDDs. At present, Spark supports some additional actions as well.

**collect().** Output the contents of the input RDD in a serialized form.

$$collect()$$
$$\texttt{RDD}[\texttt{T}] \Longrightarrow \texttt{Sequence}[\texttt{T}]$$

**save().** The `save()` action serializes the RDD and writes its contents back to disk (e.g., as an HDFS file).

$$save(path : \texttt{String})$$

**count().** Outputs the number of objects in the RDD.

$$count()$$
$$\texttt{RDD}[\texttt{T}] \Longrightarrow \texttt{Long}$$

**reduce().** This is the *action* version of the `reduce` operation, similar to the MapReduce `reduce`. The input is a function that aggregates objects in the RDD.

$$reduce(f : (T, T) \Longrightarrow T)$$
$$\texttt{RDD[T]} \Longrightarrow T$$

**lookup().** Given a key, `lookup` outputs the list of values stored under that key. This assumes that the RDD is a collection of Key-Value pairs.

$$lookup(k : K)$$
$$\texttt{RDD[(K, V)]} \Longrightarrow \texttt{Sequence[T]}$$

# RDD Representation

Resilient Distributed Datasets are

- **Immutable**.
- **Read-only**.
- **Lazily Evaluated**.

In general, an initial RDD is built by loading a data file from persistent storage into main memory.

All subsequent *transformations* over this RDD are recorded in a form of a *transformation graph*.

An *action* applied to an RDD causes the **materialization** of the RDD: i.e., it triggers the computation encoded by the *transformation graph*.

Because multiple operations in the transformation graph can be pipelined, this creates a possibility for significant improvement in performance over MapReduce.