# Resilient Distributed Datasets

Prakhar Srivastav (ps2894)

## Summary

Resilient Distributed Datasets (RDDs) is a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner which are particularly useful in two kinds of applications: iterative algorithms and interactive data mining tools.

### Motivation

Although current frameworks such as MapReduce provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. In these frameworks, they only way to share data is by writing to a file system, which can get quite expensive. Hence, these frameworks are inefficient for those applications that reuse intermediate results across multiple computations.

RDDs on the other hand, are a new kind of abstraction that enable efficient data reuse in these kinds of applications.RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators. The key idea in providing this kind of functionality was by providing an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items. This allows them to **efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage) rather than the actual data.** If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition. Thus, lost data can be recovered, often quite quickly, without requiring costly replication.

### RDDs

An RDD is a read-only, partitioned collection of records and can only be created through deterministic operations (called *transformations*) on either (1) data in stable storage or (2) other RDDs. Users can control two other aspects of RDDs: **persistence** and **partitioning** by indicating which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage). They can also ask that an RDD's elements be partitioned across machines based on a key in each record.

RDDs are typically a distributed memory abstraction and hence it makes sense to contrast them with Distributed Shared Memory (DSM) implementations.

Key differences between RDDs and DSM

- RDDs can only be created through coarse-grained transformations, while DSM allows reads and writes to each memory location.This restricts RDDs to applications that perform bulk writes, but allows for more efficient fault tolerance.
- RDDs do not need to incur the overhead of checkpointing, as they can be recovered using lineage.
- Only the lost partitions of an RDD need to be recomputed upon failure, and they can be recomputed in parallel on different nodes, without having to roll back the whole program.
- Running backup tasks is easier to run with RDDs due to the immutable nature.
- For bulk operations on RDDs, a runtime scheduler assigns tasks based on data locality to improve performance.
- RDDs degrade gracefully when there is not enough memory to store them, as long as they are only being used in scan-based operations.

**Representing RDDs**

One of the key challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations. In the paper, the authors choose a simple graph-based representation for RDDs that facilitates this. Each RDD is represented thorough a common interface that exposes five pieces of information -

1. a set of **partitions**
2. a set of **dependancies** on parent RDDs
3. a function for computing the dataset based on its parent
4. metadata about its partitioning scheme
5. data placement

To represent dependencies between RDDs, there two primary classifications -

- **Narrow dependencies** - where each partition of the parent RDD is used by at most one partition of the child RDD. These dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. Recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes
- **Wide dependencies** - where multiple child partitions may depend on it.Wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation. In a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution.

The benefit of this common interface for RDDs has made it possible to implement most transformations in Spark in less than 20 lines of code.

### Implementation

Spark is built on top of Apache Mesos where it runs as a separate application, with its own driver (master) and workers, and resource sharing between these applications is handled by Mesos. The key aspects of the system are described below -

**Job Scheduling** - Whenever a user runs an action on an RDD, the scheduler examines that RDD's lineage graph to build a DAG of stages to execute. The scheduler assigns tasks to machines based on data locality using delay scheduling.

**Interpreter Integration** - RDD's can be interacted with quickly using the scala interpreter. For better integration, two changes - *class shipping over HTTP* and *modified code generation in closures* were made to the interpreter.

**Memory Management** - Spark provides three options for storage of persistent RDDs: in-memory storage as deserialized Java objects, in-memory storage as serialized data, and on-disk storage. The trade-off between these alternatives is between performance and space efficiency.

**Checkpointing** - Checkpointing is useful for RDDs with long lineage graphs containing wide dependencies. Spark currently provides an API for checkpointing (a REPLICATE flag to persist), but leaves the decision of which data to checkpoint to the user.

### Conclusion

RDDs can express a wide range of parallel applications, including many specialized programming models that have been proposed for iterative computation, and new applications that these models do not capture. Unlike existing storage abstractions for clusters, which require data replication for fault tolerance, RDDs offer an API based on coarse-grained transformations that lets them recover data efficiently using lineage.