Resilient Distributed Datasets (RDDs)

RDDs allow apps to keep working sets in memory for efficient reuse. They retain the attractive properties of MR such as fault tolerance, data locality and scalability, and support a wide range of applications.

RDD Programming Model

RDDs are immutable, partitioned collection of objects. They are created through parallel transformations on data in stable storage, and can be cached for efficient reuse.

RDDs maintain lineage information that can be used to reconstruct lost partitions. Persisting is used to store frequently used RDDs to avoid recomputations.

The RDD abstraction is chosen because

1. Supports more than just map and reduce
2. Supports in-memory computation
3. Arbitrary computation of such operators
4. Simplifies scheduling

RDDs are best suited for batch applications that apply the same operation to all elements of a dataset. So they're not as good for applications that make async fine-grained updates to shared states.

RDDs are represented generally as a set of partitions (splits). Communication can be optimized by controlling the partitioning of RDDs.

1. They define a list of dependencies on parent RDDs
2. Partitions can be computed given parents
3. Optional preferred locations allowed for data locality
4. Optional partitioning information used for shuffling.

Operation	Meaning
partitions()	Returns a list of partition objects
preferredLocations(p)	List nodes where partition p can be accessed faster due to locality
dependencies()	Returns a list of dependencies
iterator(p, parentIters)	Compute the elements of a partition p given iterators for its parents
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Examples of RDDs

1. Hadoop RDD
    a. Partitions : 1 per block
    b. Dependencies : None
    c. Compute (partition) : read corresponding block
    d. Preferred locations : HDFS block location
    e. Partitioner : None
2. Filtered RDD
    a. Partitions : Same as parent
    b. Dependencies : 1-1 with parent
    c. Compute : Compute parent and filter it
    d. Preferred locations : Ask parent
    e. Partitioner : None
3. Joined RDD
    a. Partitions : 1 per reduce task
    b. Dependencies : Shuffle on each parent
    c. Compute : read and join shuffled data
    d. Preferred locations : none
    e. Partitioner : HashPartitioner (num tasks)

Dependencies between RDDs can be

1. Narrow
    a. Each partition of the parent is used by at most one partition of the child
    b. Does not need a shuffle, pipelined operations
    c. Map, flatMap, MapPartitions, Filter, Sample, Union
    d. Allow pipelined execution on one cluster node
    e. Recovery is easier as only that partition's parent must be recalculated
2. Wide
    a. Where multiple children can depend on the parent
    b. May need a shuffle
    c. Intersection, Distinct, ReduceByKey, GroupByKey, Join, Cartesian, Repartition, Coalesce
    d. Require data from all parent partitions to be available and shuffled across the nodes
    e. Recovery is harder as all parents need to be calculated

RDD Checkpointing

Recovery from lineage graphs may be time consuming with long lineage chains. Checkpointing is useful in this case, usually in the case of wide dependencies. This is not usually used in narrow dependencies as lineage information can be calculated much faster there.