**Resilient Distributed Datasets – A Fault Tolerant Abstraction for Sharing Data in Cluster Applications**

Raviteja Ayyagari

*Saint Peters University, Jersey City*

Resilient Distributed Datasets (RDDs), is a system for the abstraction of data. Iterative algorithms and interactive data mining tools are the two sources of motivation for the proposal of the RDDs. Regular cluster computing frameworks, which have been used for large scale data analytics have failed abstractions for distributed memory, which makes it inefficient for Data reuse. Haloop and Pregel have been developed to encounter this issue, but they are limited to specific computational patterns. RDDs, have been implemented for distributed abstraction of the data, for obtaining intermediate results for the memory. But the main challenge in designing RDDs is the fault tolerance.

The difference between existing systems and RDDs is the type of transformation, which is coarse grained in the case of RDDs, which allows fault tolerance by using the transformation rather than using the actual data. With this even if the partition is lost, the source of the partition is known and can be easily recovered. The parallel application of the RDDs makes up for its limitation in the beginning. The ability of RDDs to accommodate the applications without usage of new framework is evidence of its advantages. RDDs have been implemented with Spark, a convenient programming interface similar to DryadINQ and can access big dataset from Scala.

When we research more about RDD, which is a read only collection of records, using partitions, RDD knows where it is created from, which is its powerful property. Users can control two instances, persistence and partitioning. When we discuss about the Spark programming interface, users can count, collect and save the data or persist the system so as to reuse in the future. Using the spark system, in case we have errors in web services, we just need to load the error messages and interactively get things sorted. The biggest advantage of the RDDs is that it is fault tolerant and especially that it is coarse grained transformation. Its immutable ability to mitigate the slow nodes with backup copies is also an advantage. When bulk operations scheduled, a runtime task is created to improve performance and partitions that do not fit in the RAM can be stored on disk which is same as current parallel system. Spark programming interface which provides RDD abstraction, generally uses a driver program, that connects a cluster, where in the driver invokes actions on one or more RDDs. RDDs are statically typed objects, which are parameterized by elements.

The toughest challenges for the RDDs is choosing the representation that can track lineage across a wide range of transformations. It is very essential that a system implementing RDDs should provide rich set of transformations. The principal components that need to be checked while representing an RDD system are: partitions – to return list of partition objects, preferredLocation – where the partition p can be accessed faster, dependencies – we need to check for the list of dependencies, iterator – computing elements of the partition p, partitioner – return the metadata specifying if the RDD is hash/range partitioned. The most important factor for designing RDDs is how to represent the dependencies. Narrow and Wide dependencies, the former based on usage by each of the parent RDD by at the most one partition of the child RDD, the latter focusing on the usage by multiple partitions child RDDs. This implementation of dependencies in the interface of the Spark makes it possible for most transformations to be implemented within 20 minutes.

Implementation of Spark in Scala is generated in about 14000 lines. Spark can read data from any Hadoop input source. Spark implementation generally consists of four parts: Job Scheduling, Spark interpreter allowing interactive use, memory management and support for checkpointing. Job Scheduling is similar to that of Dryad’s, however the difference being that, Spark additionally takes into account the partitions of persistent RDDs which are available in memory. Scheduler assigns tasks to machines using delay scheduling. If a task processes a partition for which, the RDD supplies with the preferred locations. In case a task fails, re-run it on another node as long as it’s’ stage’s parents are still available. In case there are some stages that are not available, resubmitting tasks to compute the missing partitions in parallel will help. Even if all computations in Spark run in response to actions called in the driver program. In this case, tasks would need to tell the scheduler to compute the required partition if it is missing.

In case of interpreter integration, Scala includes an interactive shell similar to that of Ruby and Python. Interpreter operates by compiling a class for each line typed by the user, loading it into the JVM and then invoking a function on it. There had been two changes from Python in Spark, which are Class shipping and Modified code generation. The former lets interpreter serve the classes over HTTP while the latter deals by modifying the code generation logic with credit to the instance each line directly. Spark interpreter useful when dealing large traces and for exploring datasets in HDFS.

Memory management in Spark, there are three options, the Java objects, which provides fastest performance; second is the in-memory storage serialized data and the third on-disk storage. The second lets user opt more memory efficient representation and third option useful when RDDs are too huge to fit in the RAM. When support for checkpointing is considered, it is useful with long lineage graphs, which have wide dependencies. Spark uses API for checkpointing, however, user has the final authority. Read only feature of RDDs makes it simpler to checkpoint than usual shared memory.

Spark outperforms Hadoop by 20 times in iteration machine learning and graph applications. User written applications, are performance oriented and very productive. In case node fails, Spark knows where the source is and it recovers quickly. 1 TB of dataset can be queried. Sharing data over RDDs increases the speed and is quick for future iterations. Spark outperformed Hadoop by storing TDD elements directly within Java. By avoiding these overheads, Spark prevailed. Pagerank is another experiment by Spark, which when evaluated with a version of PageRank using the Pregel over Spark with 2.4x speedup over Hadoop 30 nodes. When spark evaluated the cost of reconstructing RDD partitions, spark generated around 400 tasks working on 100 GB of data. With a checkpoint based fault recovery principle, recovery will rerun at numerous checkpoints.

The user applications built with Spark are In-Memory Analytics, Traffic Modelling, Twitter Spam Classification. The first application performed on the same data, while performing mathematical functions like sum, average and count, etc. the speed was increased around 40X. Time taken across a road path is calculated using the traffic model. Logistic regression was used for twitter and then detect around two hundred thousand URLs related to the network.

Even though the RDDs might have limitations in programming interface because of immutable nature and coarse grained transformations, but for wide range of applications, it is very much useful. The existing programming models MapReduce, DryadINQ, SQL, Pregel, Iterative MapReduce, Batched Steam Processing are being able to express by RDDs, because the hindrances on RDDs will have very low impact. Parallel programs will use the same methods to many records. All said and done, the main reason why previous frameworks could not offer similar generality because, it was able to handle problems which MapReduce and Dryad could not handle well.

There are cluster programming models, caching systems, lineage, relational databases, which provide related work to cluster programming models. High level of programming interfaces for data flow systems with the help of operators like map and join. Some systems were able to allow user to do in-memory computation. Finally, RDDs are an efficient, regular and fault tolerant data abstraction for sharing data within cluster applications. RDDs are equipped with API base coarse transformation technique, which lets recovery of data very easily and efficiently.