Assignment #1 - Spark & RDD

Student: Sheetal Gangakhedkar **Date:** April 16, 2017 **Instructor:** Hien Luu

Q: In your own words, describe RDD

Spark processing engine operates on data abstractions called Resilient Distributed Datasets (RDDs), these once created are read-only, partitioned collection of records. These objects are fault-tolerant, as they can easily be reconstructed from the lineage information in them. RDDs are parallel data structures allowing reuse, by allowing applications to persist intermediate RDDs in memory, in a broad range of iterative applications. Applications can define their own partitioning method, to partition across machines by a key in the record, leading to optimized data placement. RDDs can only be defined or created from data on stable storage or other RDDs, by performing *transformations* (map, filter, and join). Applications use RDDs in actions, which are operations on RDDs, like count, collect and save. Spark supports a rich set of operations.

Q: What are the main use cases that Spark was designed for?

Spark was designed for iterative applications, that can reuse intermediate results across multiple computations in a cluster, like the PageRank, K-means clustering, and logistic regression applications. Spark provides data abstractions called RDDs for leveraging distributed memory. Compelling use cases include iterative data mining. Since iterative applications naturally apply the same operation to multiple data items, RDDs are a good fit for batch applications and parallel applications. RDDs are not suitable for asynchronous updates of shared state, like storage system for web applications or like incremental web crawler.

Q: What are the differences between transformations and actions and what are the advantages of having transformations lazily evaluated?

Transformations are operations that lead to creation of new RDDs. Transformations operations like map, filter and join performed on data on stable storage or other RDDs results in creation of new RDDs. Actions are operations performed on RDDs to generate results or data values or export data to a storage system. Example of actions include count (number of elements in a dataset), collect (returns elements themselves) and save (outputs dataset to a storage system). Transformations are lazy operations that lead to definition or creation of a new RDD. Actions are compute operations on the RDDs to generate results or values and also can lead to writing data to external storage.

Q: How does RDD handle fault tolerance?

RDDs remember the transformations that were performed, starting from the initial transforms performed on data on stable storage to all the intermediate transforms that led to creation of intermediate RDDs, all the way to this RDD creation. This information in RDD is called ‘lineage’. RDD provides fault tolerance by replacing a corrupt or missing RDD by regenerating the RDD using the lineage data in the RDDs. This is key to providing data fault-tolerance for Spark applications, basically a program cannot reference an RDD that cannot be rebuilt.

Q: What are the different options for storage of persistent RDDs?

Applications can use Spark actions like save and persist on RDDs. The save action allows the user to output the dataset to a storage system. The persist action or method is a hint to Spark system to keep the RDD around for future reuse and operations. Spark by default, keeps these RDDs marked with 'persist' flag in memory, but if the system is going out-of-memory threshold, the application can pass flags that define persistent strategies, like saving RDD to disk or replicating RDD across some nodes in a Spark cluster. The persist method also takes a priority flag, that defines which RDDs in priority order should get saved to disk.

Q: What are the differences between narrow and wide dependencies?

RDDs carry five pieces of information so they can be reconstructed from this lineage data. They are 1) set of partitions 2) set of parent partition dependencies 3) partition reconstruction compute function 4) partitioning scheme metadata 5) preferred data placement across nodes. The parent partition dependencies to reconstruct the RDD'RDD programming interface allows lazy evaluation of transformations, so the actions can be optimized before the actions trigger a forced evaluation.s partition 'p' can be classified into two type: narrow and wide dependencies. Narrow dependencies are where each partition of parent RDD is used by at most one partition of the child RDD, this allows for pipelined execution on one cluster node. Wide dependencies are where multiple child partitions can depend on one parent RDD partition, which makes it hard to reconstruct the child RDD, as the data needs to be shuffled across nodes to rebuild this partition.

Q: Compare and contrast between RDD and MapReduce?

RDDs are well suited for a variety of parallel applications and many cluster programming models, including interactive data mining applications, without introducing new frameworks or separate systems (MapReduce, Pregel, HaLoop, DryadLINQ). MapReduce applications are mostly batch jobs with high-latency, and most processing happens by loading data from disk, there is lot of disk I/O with MapReduce, whereas RDDs are mostly in-memory and RDD transforms are also carried out in-memory, and RDDs are suitable for iterative, interactive, graph processing and streaming applications. RDDs provide resiliency by holding on to lineage data from which they can be reconstructed, MapReduce relies on HDFS data replication. RDD programming interface allows lazy evaluation of transformations, so the actions can be optimized before the actions trigger a forced evaluation.

Q: Describe a concept or idea that you really like about Spark and it hasn't been asked from any of the above questions?

I like their three implementation ideas. To achieve lighting-fast performance the RDDs have to be in-memory, and their use of LRU eviction policy for RDDs is a good idea. When a new RDD partition is computed and it does not have enough space to store, the least recently used partition is evicted from memory to disk, ensuring it is not the newly created 'unused' partitions. The feature of checkpointing (using REPLICATE flag) some RDDs with long lineage to stable storage is also an excellent idea, if Spark can automatically handle this it will be much better. The third idea, of lazy evaluation is just amazing, allowing the Spark system to optimize the DAG stages (Directed Acyclic Graph) of operations on RDDs before evaluation of the user actions. Each DAG stage is optimized to contains as many pipelined transformations with narrow dependencies as possible.