1. Pen down the limitations of MapReduce.

Hadoop MapReduce Limitations

➢ Processing Speed

- In Hadoop, with a parallel and distributed algorithm, MapReduce processes large data sets. MapReduce requires a lot of time to perform **"map and reduce" tasks** thereby increasing latency.
- Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.
- Moreover, MapReduce **reads and writes from disk** as a result it slows down the processing speed.

Solution:

Spark has overcome this issue, by **in-memory processing** of data. In-memory processing is faster as no time is spent in moving the data/processes in and out of the disk. Spark runs applications in Hadoop clusters up to **100x faster in memory and 10x faster on disk**. Flink is also used, as it processes faster than spark because of its streaming architecture and Flink may be instructed to process only the parts of the data that have actually changed, thus significantly increases the performance of the job.

➢ Support for batch processing only, not suitable for real time processing

Hadoop Mapreduce supports batch processing only, it does not process streamed data, and if we want real-time options on top of it, we'll have to use platforms like Storm and Impala, Giraph (for graph processing). MapReduce framework of Hadoop does not leverage the memory of the Hadoop cluster to the maximum.

Solution:

Spark can process real time data i.e. data coming from the real-time event streams at the rate of millions of events per second, e.g. Twitter data for instance or Facebook sharing/posting. Spark's strength is the ability to process live streams efficiently, but Spark stream processing is not as much efficient as Flink as it uses micro-batch processing. Flink improves the overall performance as it provides single run-time for the streaming as well as batch processing. Flink uses native closed loop iteration operators which make machine learning and graph processing faster.

➢ Iterative Processing

Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow (i.e. a chain of stages in which each output of the previous stage is the input to the next stage) and **needs a sequence of MR jobs to run iterative tasks.**

Solution

Apache Spark can be used to overcome this issue, as it accesses data from RAM instead of disk, which dramatically improves the performance of iterative algorithms that access the same dataset repeatedly. Spark iterates its data in batches. For iterative processing in Spark, each iteration has to be scheduled and executed separately.

➢ Latency

In Hadoop, MapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data. In MapReduce, Map takes a set of data and converts it into another set of data, where individual elements are broken down into key value pair and Reduce takes the output from the map as input and process further and MapReduce requires a lot of time to perform these tasks thereby increasing latency.

Solution

Spark is used to reduce this issue, Apache spark is yet another batch system but it is relatively faster since it caches much of the input data on memory by RDD and keeps intermediate data in memory itself. Flink's data streaming achieves low latency and high throughput.

➢ Ease of Use

In Hadoop, MapReduce developers need to hand code for each and every operation written in Java which makes it very difficult to work. Pig however makes it easier (need to learn syntax) and Hive adds SQL compatibility. **MapReduce has no interactive mode**, but adding one such as hive and pig makes working with MapReduce a little easier for adopters.

Solution

While Spark can be used for such issue, Spark has interactive mode so that developers and users alike can have intermediate feedback for queries and other actions. Spark is easy to program as it has tons of high-level operators. Flink can also be easily used as it also has high-level operators.

➢ Abstraction

Hadoop does not have any type of abstraction so MapReduce developers need to hand code for each and every operation and handle low-level APIs to process the data, which makes it very difficult to work.

Solution

To overcome this, Spark is used in which, for batch we have RDD abstraction. Flink has Dataset abstraction.

➢ Caching

Hadoop is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.

Solution

Spark and Flink can overcome this, as Spark and Flink cache data in memory for further iterations which enhance the overall performance.

➢ Lines of Code

Hadoop 2.0 has 1, 20,000 lines of code, the number of lines produces the number of bugs and it will take more time to execute the program.

Solution

Although Spark and Flink are written in scala and java but they are implemented in scala, so the number of lines of code is lesser than Hadoop (Apache Spark is developed in merely 20000 lines of codes.). So it will also take less time to execute the program.

➢ **Needs integration with several other frameworks/tools to solve big data use-cases.**

It needs Apache Storm for stream data processing, Apache Mahout for machine learning.

However, Apache Spark can process real time data through **Spark Streaming**, and it has its own set of machine learning i.e. **MLlib.**

2. What is RDD? Explain few features of RDD?

It is the primary abstraction in Spark and is the core of Apache Spark. One could compare RDDs to collections in Scala, i.e. a RDD is computed on many JVMs while a Scala collection lives on a single JVM.

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures.
- **Distributed** with data residing on multiple nodes in a cluster.
- **Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects.

Why RDD?

- For iterative distributed computing, it's common to reuse and share data among multiple jobs or do parallel ad-hoc queries over a shared dataset.
- The persistent issue with data reuse or data sharing exists in distributed computing system (like MR) - that is, you need to store data in some intermediate stable distributed store such as HDFS or Amazon S3. This makes overall computation of jobs slower as it involves multiple IO operations, replications and serializations in the process.

Features of RDD:

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed** — RDD records have types, e.g. Long in RDD[Long] or (Int, String) in RDD[(Int, String)].
- **Partitioned** — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.
- **Location-Stickiness** — RDD can define placement preferences to compute partitions (as close to the records as possible).

3. List down few Spark RDD operations and explain each of them.

RDD supports two kinds of operations:

- **Actions** - operations that trigger computation and return values
- **Transformations** - lazy operations that return another RDD

Actions

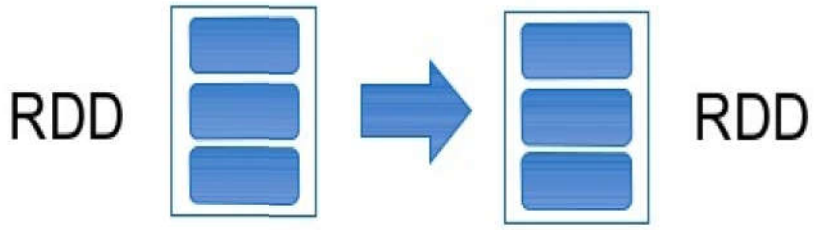
- **Actions are RDD operations that produce non-RDD values**. In other words, a RDD operation that returns a value of any type except RDD[T] is an action.
- They trigger execution of RDD transformations to return values. Simply put, an action evaluates the RDD lineage graph.
- You can think of actions as a valve and until action is fired, the data to be processed is not even in the pipes, i.e. transformations. Only actions can materialize the entire processing pipeline with real data.
- Actions are one of two ways to send data from executors to the driver (the other being accumulators).

Transformations

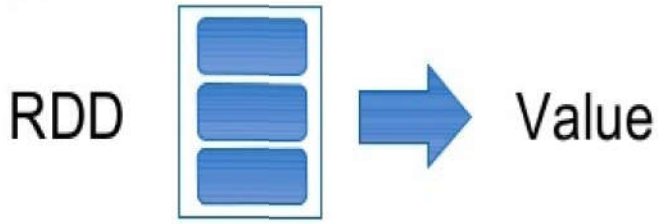
- **Transformations** are lazy operations on a RDD that create one or many new RDDs, e.g. **map, filter, reduceByKey, join, cogroup, randomSplit.**
- They are ***functions*** that take a RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable), but always produce one or more new RDDs by applying the computations they represent.
- Transformations are **lazy**, i.e. are not executed immediately. Only after calling an action are transformations executed.
- After executing a transformation, the result RDD(s) will always be different from their parents and can be **smaller (e.g. filter, count, distinct, sample)**, **bigger (e.g. flatMap, union, cartesian)** or the **same size (e.g. map).**

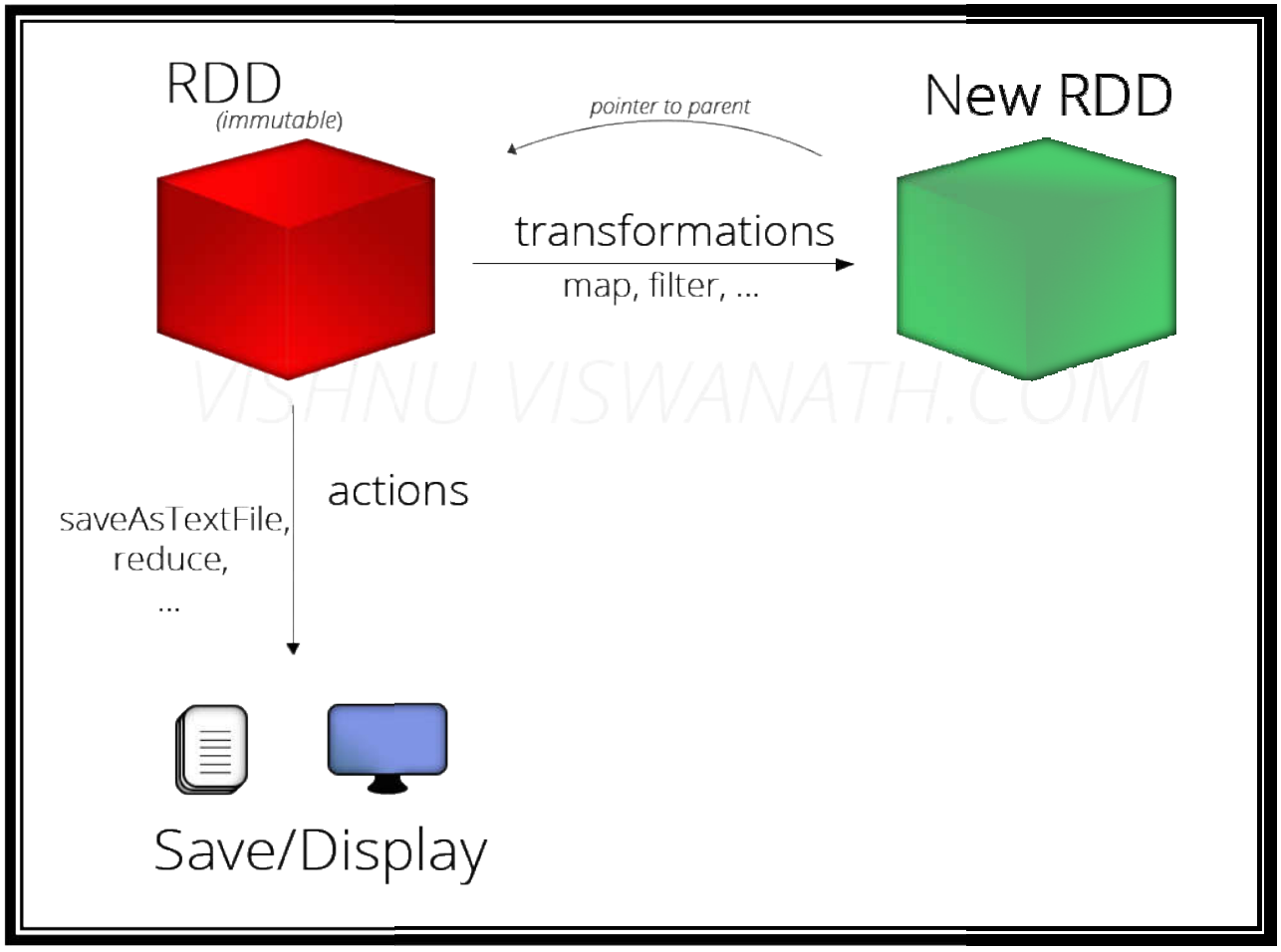
NOTE: Diagrams showing the difference between Transformation and Action are shown in next pages.

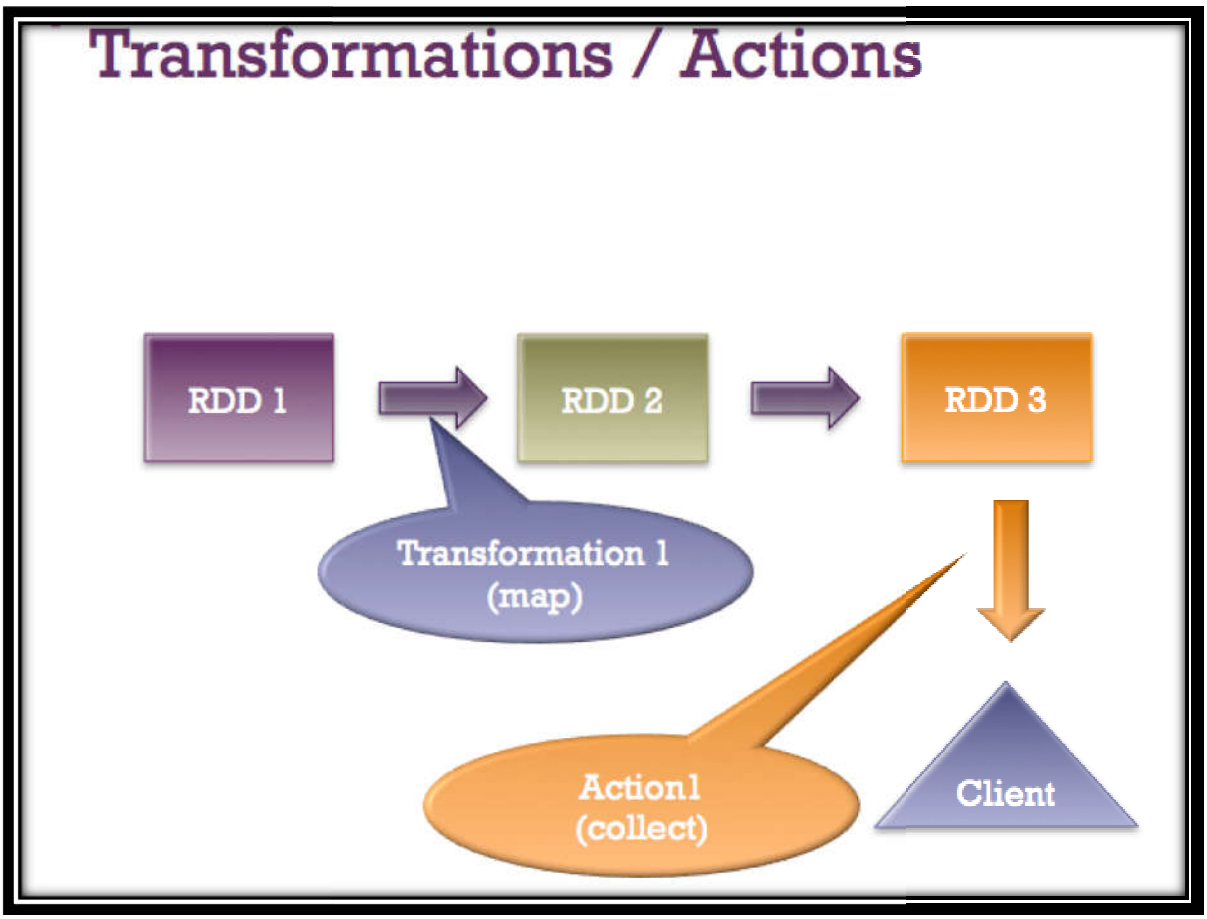
Transformations: define new RDDs based on current one. E.g. map, filter, reduce etc.



Actions: return values. E.g. count, sum, collect, etc.







Transformations

map (func)
flatMap(func)
filter(func)
groupByKey()
reduceByKey(func)
mapValues(func)
sample(...)
union(other)
distinct()
sortByKey()
...

Actions

reduce(func)
collect()
count()
first()
take(n)
saveAsTextFile(path)
countByKey()
foreach(func)
...