**Abstract:**

**Resilient Distributed Datasets**- a distributed memory abstraction that lets programmers perform in memory computations on large clusters, very efficiently in fault tolerant manner

1. **Introduction**

Spark is cluster computing framework designed to handle large datasets. Spark makes unique in this type computing because its design to cover wide range of workloads in one place. It reduces burden of maintaining different tools (Like in Hadoop systems). Spark can run on clusters and access Hadoop data. It provides API in python, java, Scala and SQL.

**Spark Stack:**

* ***Spark Core*:** Spark provides closely integrated components at its core, provides many benefits like when core engine adds optimization, its Machine learning libraries also benefits from core. It Provides Task Scheduling, memory management, fault recovery, interacting with storage. Mainly it provides API to define RDD (Resilient distributes datasets) which will represent collection of Items distributed across nodes to manipulate in parallel.
* ***Spark SQL*:** Spark Package to work with structured data. Both SQL and hiveSQL are used to query data from design formats like Hive table, Parquet, JSON formats.
* ***Spark Streaming*:** Its used to process live streams of data. Ex: Log files on web servers. Most of the API similar to Core RDD API making it easy to understand to move between applications.
* ***MLIB and GraphX:*** MLIB provides Machine Learning algorithms and GraphX is for manipulating graphs.

There are two kind of tasks spark will provide

* **Data Science Tasks:**

Spark shell is easy to interact for data scientists. There is also Spark SQL shell, used to do data exploration using SQL. There is a support for calling external programs in matlab or R.

* **Data Processing Applications:**

Software developers use spark to build application and spark will parallelize these applications across clusters and hides complexity of distributed system programming, communication and fault tolerance.

Spark in memory computing makes applications improve more than 20 percent for certain jobs. Spark can run many cluster managers like Hadoop YARN, Apache Mesos. Spark can run on any Hadoop distributed file system (HDFS) or other storage systems supported by Hadoop like Cassandra, Hive, Hbase etc. Spark support text files, Sequence Files, Avro, Parquet and any other Hadoop input format.

Most of the big data related computations need reuse of data to run iterative algorithms. Many existing systems will reuse data is to write into into disk each time to access. Some frameworks like Pregal and haloop allow in memory for only specific kind of calculations not like RDD which will generalize many purposes which make it better than other systems to use iterative algorithms.

1. **Resilient Distributed Dataset(RDDs):**

**RDD** provides in memory computation for many generic usages and also provides fault tolerance by making RDD coarse grained. RDD’s are read only created either from Storage or from other transformation. RDD’s are not always materialized. They have enough information via lineage graph how it was derived. User can choose how it can be stored like Persistence or Partitioning. Transformations define RDD (ex: map, filter) They are lazily computed when an action is called on RDD. User can choose what type persistence they choose and also persistence priority (First disk or memory).

**Example: Log files**: Suppose we want to find errors in logs this is the Scala code to find errors.

1. *lines = spark.textFile("hdfs://...")*
2. *errors = lines.filter(\_.startsWith("ERROR"))*
3. *errors.persist()*

So far no operation performed in disk or cluster**.**

1. *errors.filter(\_.contains("MySQL")).count()*

After 4th line, **errors** are stored in partition. Base RDD, lines is not stored in cluster. Only lineage graph drawn. It is efficient because errors are small part of lines. If any of the partitions is lost for errors only last partition rebuilds by applying corresponding filter of lines.

**Advantages of RDD**

* Distributed shared memory (DSM) also general abstraction and but DSM allows reads and writes to each memory location. This feature makes it very hard in fault tolerant. RDD only allows bulk writes (Through map operations) makes it easier to achieve fault tolerance.
* Another benefit of RDD is immutable in nature lets a system can have slow nodes by running backup copies. Backup copies are hard to implements in DSM as two copies of a task would access the same memory location and interface with each other’s updates
* RDD degrade gracefully when there is not enough memory

**RDD VS Distributed Shared memory**:

RDD’s are coarse grained to better for fault tolerant. DSM’s are fine grained used for application which may involve more writes. RDD do need check pointing because lineage graph. Partitions of RDD which do not fit in RAM will spilled into disk will also provide similar performance.

**Applications not suitable for RDD’S**

Applications not suitable for RDD: RDD are best suitable for batch jobs. Each transformation as one item in lineage graph and can recover lost partitions without having to log large amounts of data. RDD less suitable for applications that makes fine grained updates like a web application storage system or web crawler.

1. **Spark Programming interface:**

Spark API written in Scala which is statically typed functional language which will run on JAVA virtual machine. Spark uses Scala because of combination of Conciseness and efficiency (Static typing). To use spark developers, write driver program that connects to a cluster of workers. Driver defines one or more RDD and invokes them and draw a lineage graph. Workers are long lived processes that can store RDD partitions in RAM across operations.

1. **Representing RDD’S:**

Representing RDD through a common interface that exposes five pieces of information

* A set of partitions which are atomic pieces of data set.
* Preferred locations
* A set of dependencies on Parent’s RDD which is a function computing the dataset based on its parent RDD’s
* Meta data about partitioning scheme and data placement(Partitionar())
* Iterator (p, parentIters)

Two types dependencies between RDD’s narrow dependencies and Wider dependencies. Map is narrow dependency and Join is wider dependency. Narrow dependency where each partition of parent use at most one child partition whereas in wider dependency multiple child partitions depend on it.

Recovery after node failure is efficient in narrow dependencies where lost partitions can be determined where in wide dependencies all partitions needs to be determined.

1. **Implementation:**

Spark written on Scala around 14000 lines of code which very small code compared to Hadoop. Spark runs on Mesos cluster manager. Each spark program runs as a separate Mesos application, with its own driver and workers and resources sharing between these applications is handled by Mesos.

**Job Scheduling:**

Whenever user runs an action spark scheduler examines the RDD lineage graph to build a DAG of stages to execute. Each stage contains many pipelined transformations with narrow dependencies as possible. The boundaries of the stages are the shuffle operations required for wider dependencies or any already computed partitions that can short circuit the computation of parent RDD. The scheduler then computes missing partitions from each stage until it computed the target RDD.

**Interpreter Integration:**

Scala provides interactive shell. Because of low latencies. The Scala interpreter compile each line as a separate and loads into JVM and invokes it. Spark interpreter more useful when in processing large datasets obtained from research projects.

**Memory Management:**

There are three types of memory management

1. RDDs in memory storage as desterilized Java objects.
2. in memory storage as serialized.
3. ON-DISK storage.

First option provides fastest performance because JAVA VM will read RDD natively. Second option will choose more memory efficient representation.3rd option useful when RDD too large

To manage limited memory Spark will use LRU evicted policy (LEAST RECENTLY USED). When a new partition is computed and there is not enough space partition from least used RDD is stored in memory. if it’s from same RDD older partitions are return in memory. Even though this policy works better for most of the application, the user also given more control via “persistence priority”. Each Spark instance have separate memory space to store RDD.

**Check Pointing:**

Although lineage will provide recover RDD sometimes lineage graph contains much wider dependencies and long lineage chains. In that cases checkpoint RDD in some stage is helpful. Rank dataset in page rank is one example for this. For some other cases which has narrow dependencies check pointing is not worth to implement.

Spark currently provides a flag to persist via API, leaves the decision to user. But in future as scheduler know size of RDD and how much time will it take to recalculate spark will provide automatic check pointing. **Check pointing** is simpler than general shared memory because RDD in background will never application to slow down

1. **Evaluation:**

Spark Benchmarking provides fallowing data:

* Spark out performs Hadoop by 20 times fast in iterative machine learning and graph applications. The speed comes from data stored in memory java objects.
* Applications developed by users scale well because analytics build by users report running on Hadoop speed up by 40X
* Spark build RDD very quickly when failure by calculating only lost partitions.
* Spark used to query a 1 TB with speed of 5.7 secs

Examples: Conviva Inc, a video distribution company, used Spark to accelerate a number of data analytics reports that previously ran over Hadoop. By implementing queries in spark this company was able to speed up the report by 40X.