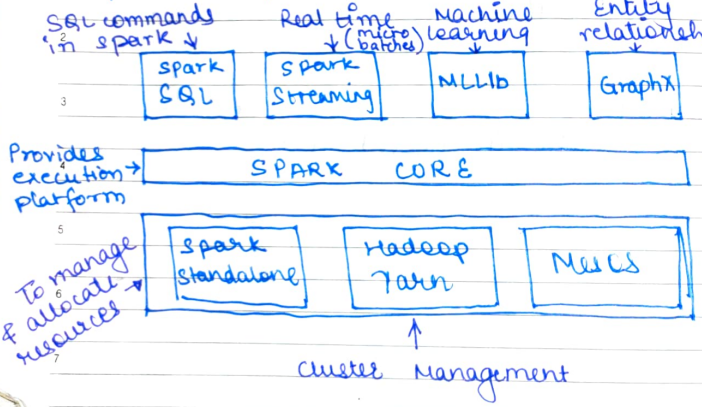
★ SPARK.

→ It is an independent framework.
→ It is a low latency, lightning fast cluster computing platform.
→ Uses in-memory storage to process the data.
→ Can perform batch as well as stream processing
→ Reads data from HDFS | HIVE | DBS | cloud
  → processes & stores in HDFS | DBS

• 1. Spark Components



1. Spark Core
→ Foundation of parallel & distributed processing of huge dataset.
→ Delivers speed by providing in-memory computation capability.

→ Features:
- In charge of essential I/O functionalities.
- Task dispatching
- Fault recovery
- Embedded with RDDs (Resilient distributed dataset).
  → handles partitioning data across all nodes in cluster.
  → It holds them in memory pool of the cluster as a single unit.

2. Spark SQL
→ works to access structured & semi-structured info.
→ Enables powerful, interactive, analytical app. across both streaming & historical data.

→ Features:
- Cost based optimizer
- Mid query fault tolerant
- Full compatibility with existing hive data

3. Spark Streaming
→ Add on to core spark API which allows scalable, high-throughput, fault-tolerant stream processing of live data streams.
→ Access data from sources like kafka, Flume, kinesis or TCP socket
→ It uses Micro-batching for real-time streaming

4. Spark MLlib
→ It is a scalable M L library that performs high-quality algorithm & high speed.

5. GraphX
→ API for graphs & graph parallel execution.
→ It is network graph analytics engine & data store.

• Features of spark

1. Swift processing
→ high data processing speed, made possible by reducing no. of read-write to disk.

2. Dynamic in nature
→ easily develop a parallel application

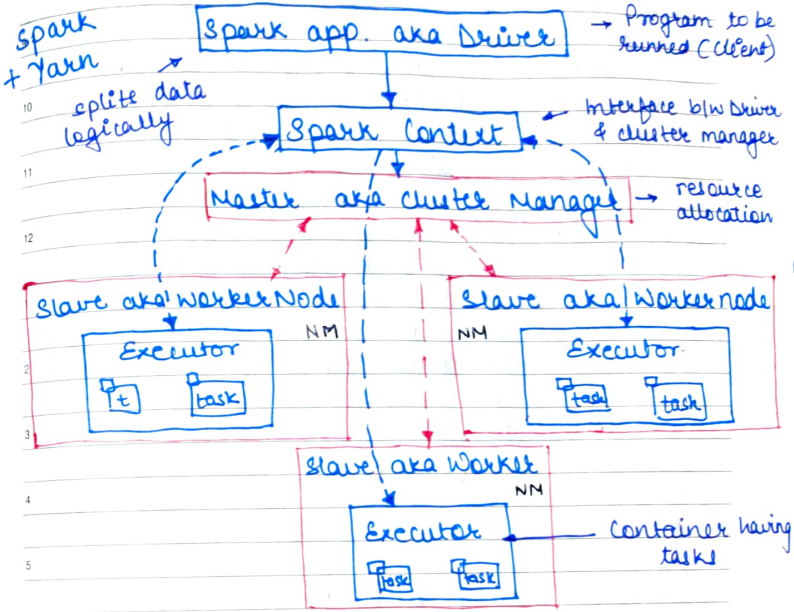
3. In-Memory computation in Spark
→ Processing speed is increased.
→ Data is cached so we need not fetch data from disk every time, thus time is saved.

4. Fault tolerant
- Provided through RDDs.

5. Real-Time Stream processing

6. Lazy Evaluation
- All the transformation we make in RDD are lazy in nature.
It does not give result until action command are called.

• Architecture



→ Process
→ flat file (1000 rows) → Driver → Spark Context
① load
② filter
Worker nodes → Executor → loading data in memory → RDD1 (logical)
Slave Nodes → Executor → Tasks on Data → Storing in memory → RDD 2

1. Driver
→ It is a process that runs the main() fn of the application & creates spark context object
→ It is the controller of a spark Application & maintains all of the state of spark cluster
→ It must interface with cluster manager in order to actually get physical resources & launch executors.

2. Spark Context
→ Coordinate the spark applications, running as independent sets of processes on a cluster.
→ It acquires executors on nodes in cluster.
→ Then, it sends your app. code to the executor
→ At last, the spark Context sends task to the executors to run.

3. Cluster Manager
→ Allocates resources accross app.

4. Worker Node | slave node.
→ role is to run the app. code in cluster.

5. Executor → sends results to driver.
- An executor is a process launched for an application on worker node.
- It runs tasks & keeps data in memory or disk storage across them.
- It read & write data to external sources
- Every app. contains its executor
- A unit of work that will be sent to one executor.

* RDD (Resilient Distributed Dataset).

→ It is a fundamental data structure of spark.
→ It is an immutable collection of objects which computes on the different node of the cluster.
→ every dataset in RDD is logically partitioned across servers to be computed on diff. nodes.

→ Resilient, i.e fault tolerant with the help of RDD lineage graph (DAG). to recompute missing or damaged partitions due to node failures.
→ Distributed, data residing on multiple nodes
→ Dataset, represents records of data you work with.

• Features
- In-memory computation
  → stores everything in RAM
- Lazy Evaluations
  → spark computes transformation when an action requires a result for driver program.
- Fault Tolerance.
  → Because of lineage.
- Immutability
  → cannot be changed or updated
- Partitioning
  → Fundamental unit of parallelism in spark RDD
  → Can create a partition thr. some transformation on existing partitions.
- Persistence
  → Users can state which RDDs they will reuse & choose a storage strategy for them.

- Limitations
  - No inbuilt optimization engine.
  - Handling structured data
    - → need to specify the schema.
  - Performance limitation
    - → As it involves the overhead of Garbage collection & Java serialization which are expensive when data grows.
  - Storage limitation
    - → spill over, if size of RDD is larger than RAM then it uses disk for remaining RDD size.

- Ways to Create RDDs in Spark.

1. Parallelized collection (parallelizing)
   - → by taking an existing collection in the program & passing it to SparkContext's parallelize() method.
   - → Used in initial stage, as it creates RDD quickly

2. External Dataset (Referencing a dataset).
   - → We can use textFile method.
   - → It takes URL of file & read it as a collection of line.
   - → loading CSV, JSON, textFile.

3. Creating RDD from existing RDD.
   - → transformation is the way to create an RDD from already existing RDD.

- RDD Persistence & Caching

→ Persistance is an optimization technique in which saves the result of RDD evaluation.
→ Using this we save the intermediate result so that we can use it if required.
→ It reduces the computation overhead.
→ When we use the cache() method we can store all the RDD in-memory.
→ we can persist the RDD in memory & use it efficiently across parallel operations.

→ The diff. b/w cache() & persist() is that using cache() the default storage level is MEMORY_ONLY while using persist() we can use various storage levels.

→ When we persist RDD each node stores any partition of it that it computes in memory & makes it reusable for future use. This process speed up the computation.

→ When the RDD is computed for first time, it is kept in the memory on the node. The cache memory of the spark is fault tolerant so whenever any partition of RDD is lost, it can be recovered by transformation operation that originally created it.

- Storage levels of persist()
1. Memory_only
2. Memory_and_disk (spill over in disk)
3. Memory_only_ser (serialized Java objects)
4. Memory_and_disk_ser
5. Disk_only.

- Unpersist RDD
→ Spark automatically drops out the old data partition in LRU (least recently used) fashion.
→ We can remove manually using RDD.unpersist().

- RDD Operations

1. Transformation
→ Produces new RDD from existing RDD.
→ Applying transformation built an RDD lineage with the entire parent RDDs of the final RDD dependency graph.
→ Are lazy in nature i.e they get executed when we call an action.
→ Two types of transformation:
1. Narrow transformation
→ All elements that are req. to compute the records in single partition live in the single partition of parent RDD.
→ map(), filter(), flatMap(), mapPartitions(), sample(), union().

2. Wide transformation
→ All elements that req. to compute the records in the single partition may live in many partitions of parent RDD.
→ eg. groupbyKey(), reducebyKey(), join(), intersection(), distinct(), cartesian(), repartition(), coalesce().

2. Action
→ Works with actual data when action is performed.
→ Does not form new RDD.
→ The values of action are stored to drivers or to the external storage system.
→ Action is one of the way of sending data from Executor to the driver.

→ count() - no. of elements is returned
collect() - returns the entire RDDs content to driver program.
take(n) - returns n no. of elements from RDD.
top() - extract top elements from RDD.
countByValue() - returns, occurence of each element
reduce() - operation like addition, takes two element as input.
fold() - Takes zero value as input.
diff. - reduce throws an exception for empty collection, but fold is defined for empty collection
aggregate(), foreach()

01 DAY 152-213 WEEK 23
WEDNESDAY
2022
JUNE

★ RDD Lineage

- When we create new RDD from an existing RDD, that new RDD also carries a pointer to the parent RDD.
- All the dependencies b/w RDDs those are logged in a graph, rather than actual data, called lineage graph.
- It is a graph of all the parent RDDs of an RDD.

★ Dataframe

→ Data organised into named columns.
→ Similar to table in RDBMS.
→ Can say that it is a relational table with good optimization technique.

→ Processes large amount of structured data
→ Contains schema (illustration of the structure of data).
→ Immutability, in-memory, resilient, distributed computing capability.
→ Sources data from structured data file, tables in Hive, external dbs or existing RDDs.
→ Available in Scala, Java, Python & R.

→ Data frame over RDD

① Provides memory management
- data is stored in off heap memory in binary format. Serialization is avoided

JUN 2022 M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S
• • 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 • • •

2022
JUNE
DAY 153-212 WEEK 23
THURSDAY
02

② Optimized Execution plan
- query optimizer, where execution plan is created for the execution of a query.

• Limitation of RDD
- Does not have any built-in optimization engine.
- No provision to handle structured data.

• Features of Dataframe
→ distributed collection of data organized in named column, equivalent to table in RDBMS.
→ Deals with structured & semi-structured data formats. eg. Avro, csv, elastic search, cassandra.
→ Deals with storage systems - HDFS, Hive, MySql etc.
→ Catalyst supports optimization, 4 phases:
1. Analyze logical plan to solve references
2. Logical plan optimization.
3. Physical planning
4. Code generation to compile part of query to java bytecode.

• Limitation
- Does not provide provision for compile time type safety. So, we need to make a structure in order to manipulate.

M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S JUL
• • • • 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 2022

★ Dataset

- Strongly typed & is map to relational schema.
- Represents structured queries with encoders.
- Provides both type safety & object-oriented programming interface.

• Features

- Optimized Query - (catalyst Query Optimizer).
- Analysis at compile time. - check syntax & analysis at compile time.
- Persistent storage - serializable & Queryable.
- Faster computation (than RDD).
- Less memory consumption - structure of data in dataset is known.