**Apache Spark Interview Preparation**

Refer :

<https://luminousmen.com/post/dive-into-spark-memory>

<https://www.linkedin.com/pulse/apache-spark-memory-management-deep-dive-deepak-rajak/>

<https://www.waitingforcode.com/apache-spark/collecting-part-data-driver-rdd-tolocaiIterator/read>

<https://www.waitingforcode.com/apache-spark-sql/multiple-sparksession-one-sparkcontext/read>

<https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html>

<https://docs.databricks.com/delta/join-performance/skew-join.html>

<https://itnext.io/handling-data-skew-in-apache-spark-9f56343e58e8>

<https://www.waitingforcode.com/big-data-problems-solutions/skewed-data/read>

<https://www.waitingforcode.com/apache-spark-sql/whats-new-apache-spark-3-join-skew-optimization/read>

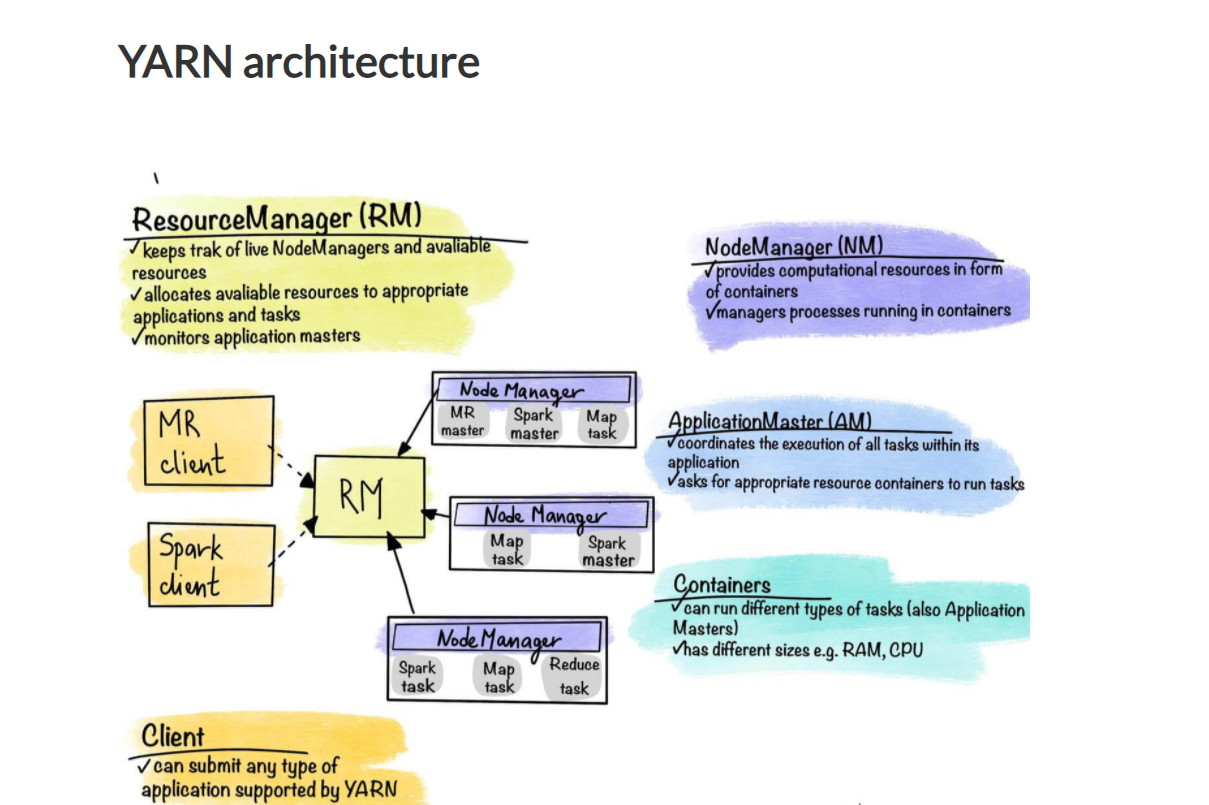
<https://dzone.com/articles/guide-to-partitions-calculation-for-processing-dat>

<https://nxtgen.com/hadoop-file-formats-when-and-what-to-use>

<https://blog.knoldus.com/understanding-sparks-logical-and-physical-plan-in-laymans-term/>

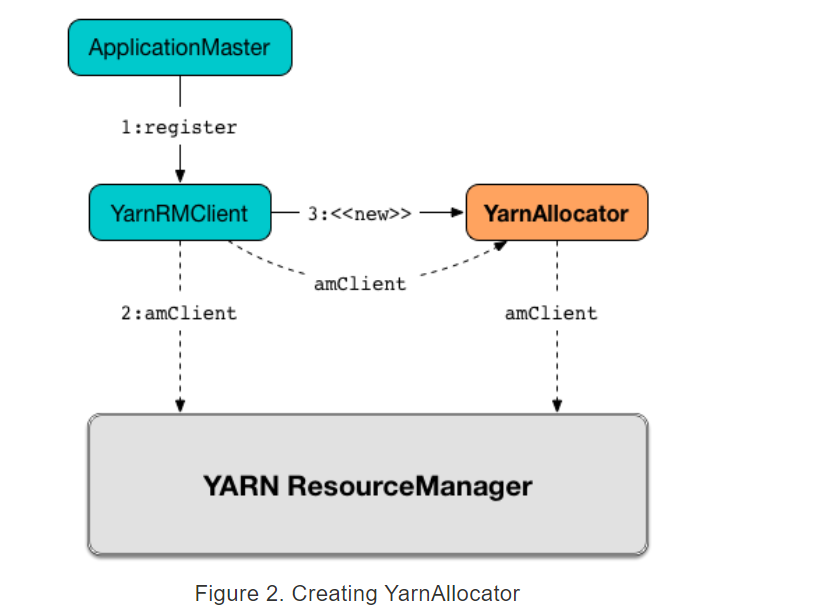
<https://mallikarjuna_g.gitbooks.io/spark/content/spark-sparkenv.html>

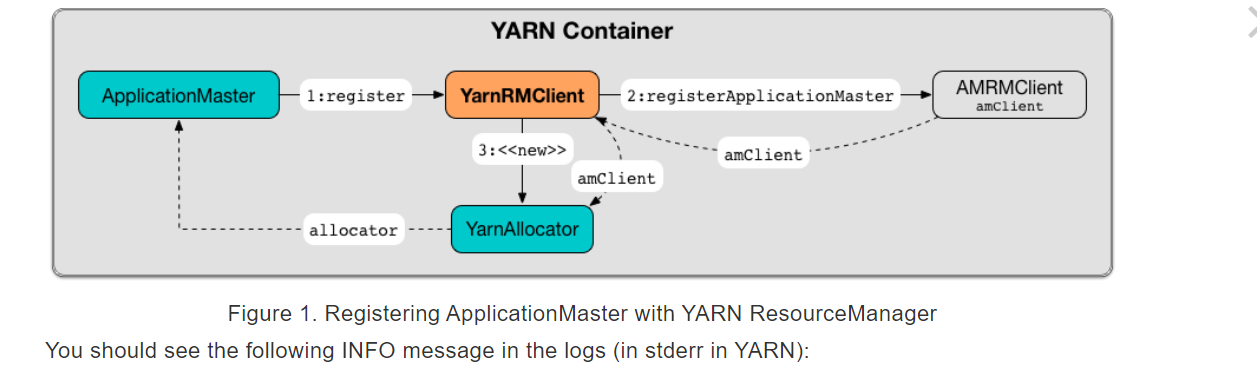
**Things you need to know about Hadoop and YARN being a Spark developer**



When you submit any application to YARN

1. When a client program submits an application to YARN, Resource manager will select necessary container where application master can start. (Its not only RM , but also LocalityPreferredContainerPlacementStrategy in spark framework also plays a roles in deciding the nodes(hosts) where container need to be launched)
2. Basically RM starts a YarnRMClient process which is responsible for [registering](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-yarnrmclient.html#register) and [unregistering](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-yarnrmclient.html#unregister) a Spark application that is started when [registering ApplicationMaster](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-yarnrmclient.html#register). Besides being responsible for [registration](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-yarnrmclient.html#register) and [unregistration](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-yarnrmclient.html" \l "unregister), it also knows the [application attempt identifiers](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-yarnrmclient.html#getAttemptId) and [tracks the maximum number of attempts to register ApplicationMaster](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-yarnrmclient.html#getMaxRegAttempts).
3. YarnAllocator [allocates resource containers](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-YarnAllocator.html#allocateResources) from [YARN ResourceManager](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-introduction.html#ResourceManager) to run Spark executors on and [releases them](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-YarnAllocator.html#internalReleaseContainer) when the Spark application no longer needs them.It talks directly to YARN ResourceManager through amClient that it gets when [created](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-YarnAllocator.html#creating-instance)





1. Once it is selected, resource manager will communicate to the corresponding node manager of node to launch the container and start application master with the resource specification provided by the user while submitting the client program
2. Application Master for each application is a framework-specific entity(i.e, for spark and MR it will be different. It will be specific to what distributed computing framework you are using in your program) that is tasked with negotiating resources with ResourceManager and working with NodeManager(s) to perform and monitor component tasks.
3. Once launched, ApplicationMaster will be responsible for the entire lifecycle of the distributed application. First of all, it will send resource requests to ResourceManager to get the containers needed to perform the application tasks. A resource request is simply a request for a number of containers that meets some resource requirements, for example:

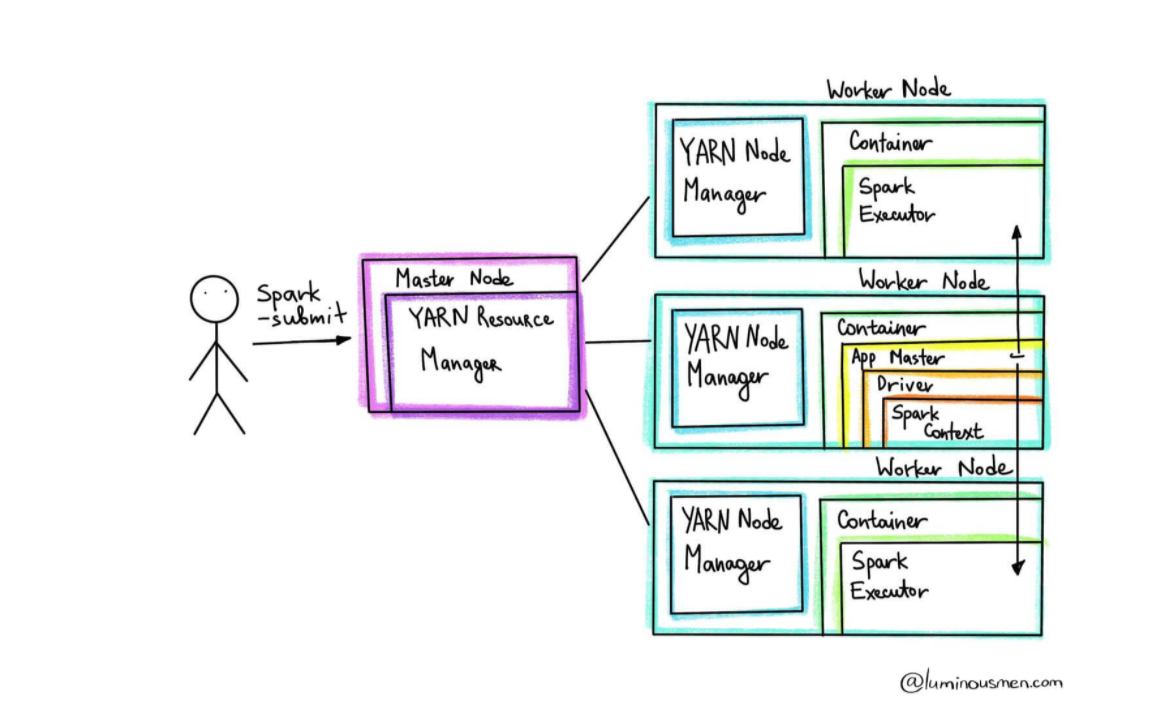
* A number of resources, expressed in megabytes of memory and processor shares
* Preferred container location, indicated by hostname, rackname, or \* to indicate no preference
* Priority within the application, not for multiple applications.

1. Resource Manager restarts application master if it crashes ,recreates the state of applications and restarts only tasks that have not been completed. This makes the application highly available
2. Application Master registers itself with RM via YarnRMClient. Registration allows the Customer program to request specific information from ResourceManager that allows it to directly interact with its ApplicationMaster.
3. In normal operation ApplicationMaster asks for suitable containers from ResourceManager for the application to run.
4. After successfully receiving the containers, ApplicationMaster launches them, providing NodeManager(s) their configurations.
5. Inside the containers, it runs the user application code. The NodeManager(s) then provides the information (execution phase, status) for ApplicationMaster.
6. During the runtime of the user application, the client interacts with ApplicationMaster to obtain the application status.
7. When the application completes and all necessary work is completed, ApplicationMaster deregisters from ResourceManager and terminates, releasing the container for other purposes.

**Extra Info:**

* **Uberization** is the ability to run all MapReduce tasks in ApplicationMaster's JVM if the tasks are small enough. This way, you avoid the overhead associated with requesting containers from ResourceManager and asking NodeManagers to run (presumably small) tasks.
* Simplified management and access to application log files. Application-generated logs do not remain on individual slave nodes (as in MRv1) but are moved to a central repository, such as HDFS. They can later be used for debugging or for historical analysis to detect performance problems.

**Anatomy of Spark application**

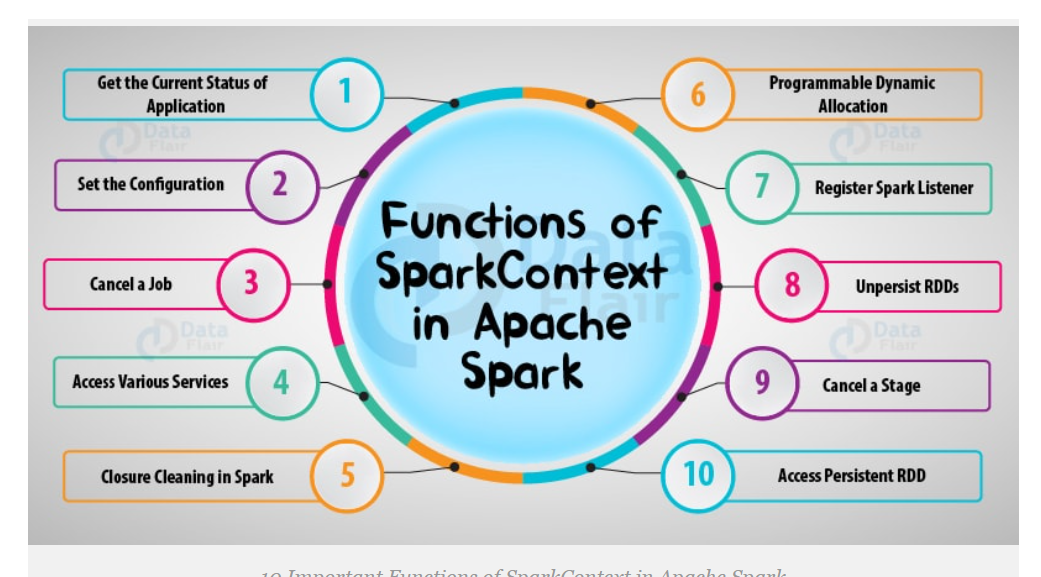


Components:

* Driver
* Application Master
* Spark Context
* Cluster Resource Manager(aka Cluster Manager)
* Executors

1. **The Driver(aka driver program)** is responsible for converting a user application to smaller execution units called **tasks** and then schedules them to run with a cluster manager on executors. The driver is also responsible for executing the Spark application and returning the status/results to the user.
   1. Run in its own JVM. If in cluster mode , it runs along with YARN’s AM in same container
   2. In client mode it run as an independent process
   3. optimizes logical DAG transformations and, if possible, combines them in stages and determines the best location for execution of this DAG
   4. creates Spark WebUI with detailed information about the application
2. **Spark Master** (AM)is created simultaneously with Driver on the same node (in case of cluster mode) when a user submits the Spark application using spark-submit.
3. The Driver informs the Application Master of the executor's needs for the application, and the Application Master negotiates the resources with the Resource Manager to host these executors.
4. **Spark Context** allows driver to access the cluster through its Cluster Resource Manager and can be used to create RDDs, accumulators and broadcast variables on the cluster. Spark Context also tracks executors in real-time by sending regular heartbeat messages. **we can only have one**

**active SparkContext per JVM**



**Note:**

This behavior can change though when we set the **spark.driver.allowMultipleContexts** configuration flag to true. However having multiple SparkContexts in the same JVM is not considered as a good practice. It's useful for tests execution and it's, by the way, its primary use in Apache Spark library. Outside the test scope, it's not guaranteed that the pipeline will work correctly with multiple active contexts. Besides, it makes the whole data pipeline more difficult to manage. The workflow is not isolated - a potential failure of one context can impact another and even it can break down the whole JVM. It also brings some extra hardware pressure on everything the driver does. Even though we collect a part of data with toLocalIterator (read more in the post Collecting a part of data to the driver with RDD toLocalIterator), it's always multiple times more data to process than with isolated processes

1. One of the drawbacks of SparkContext was its specific character regarding the processing context. In order to work with Hive we needed to use HiveContext. If we wanted to deal with structured data the SQLContext instance had to be used. StreamingContext was devoted to streaming applications. That problem was addressed with **SparkSession** which appears as one common entry point for all different pipelines. The instance of SparkSession is constructed with a builder common for all processing types except Hive which requires a call to **enableHiveSupport()** method.

If you set enableHiveSupport(), then spark.sql.catalogImplementation is set to hive, otherwise to in-memory. After this only , spark can read and write to hive meta store . Also this enables spark to get access to hive UDFs , SerDe etc.

1. **Cluster Manager** in a distributed Spark application is a process that controls, governs, and reserves computing resources in the form of containers on the cluster. These containers are reserved by request of Application Master and are allocated to Application Master when they are released or available. SparkContext can connect to different types of Cluster Managers. Now the most popular types are YARN, Mesos, Kubernetes or even Nomad. There is also Spark's own standalone cluster manager.
2. **Executors** are the processes at the worker's nodes, whose job is to complete the assigned tasks. These tasks are executed on the worker nodes and then return the result to the Spark Driver.

Executors are started once at the beginning of Spark Application and then work during all life of the application, this phenomenon is known as "Static Allocation of Executors". However, users can also choose to dynamically allocate executors where they can add or remove executors to Spark dynamically to match the overall workload

**Spark application running steps**

1. When we send the Spark application in cluster mode, the spark-submit utility communicates with the Cluster Resource Manager to start the Application Master.
2. In Cluster mode application master and driver will be started in the same container. Container is launched by specific node manager after getting the instruction from cluster resource manager
3. Application master registers with Resource Manager via YarnRM client. Registration allows the client program to request information from the Resource Manager, that information allows the client program to communicate directly with its own Application Master.
4. The Spark Driver then runs on the Application Master container (in case of cluster mode).
5. The driver implicitly converts user code containing transformations and actions into one or more Spark jobs and then each job in to a logical plan called a DAG. Each DAG can have multiple stages and each stage can have multiple tasks. All RDDs are created in the driver and do nothing until the action is called. At this stage, the driver also performs optimizations such as pipelining narrow transformations.
6. The driver process scans through the user application. Based on the RDD actions and transformations in the program, Spark creates an operator graph.
7. When an action (such as collect) is called, the graph is submitted to a DAG scheduler. The DAG scheduler divides the operator graph into stages.
8. A stage comprises tasks based on partitions of the input data. The DAG scheduler pipelines operators together to optimize the graph. For instance, many map operators can be scheduled in a single stage. This optimization is the key to Spark's performance. The final result of a DAG scheduler is a set of stages.
9. It then converts the DAG into a physical execution plan. After conversion to a physical execution plan, the driver creates physical execution units called tasks at each stage.
10. The stages are passed on to the task scheduler. The task scheduler launches tasks via cluster manager. (Spark Standalone/Yarn/Mesos). The task scheduler doesn't know about dependencies among stages.
11. The Application Master now communicates with the Cluster Manager and negotiates resources. Cluster Manager allocates containers and asks the appropriate NodeManagers to run the executors on all selected containers. When executors run, they register with the Driver. This way, the Driver has a complete view of the artists.
12. At this point, the Driver will send tasks to executors via Cluster Manager based on the data placement.
13. The code of the user application is launched inside the container. It provides information (stage of execution, status) to the Application Master.
14. During the execution of the user application, the client communicates with the Application Master to obtain the application status.
15. When the application finishes executing and all of the necessary work is done, the Application Master disconnects itself from the Resource Manager and stops, freeing up its container for other purpose

Factors identifying number of partitions :

(a)spark.default.parallelism (default: Total No. of CPU cores)

2

(b)spark.sql.files.maxPartitionBytes (default: 128 MB)

3

(c)spark.sql.files.openCostInBytes (default: 4 MB)

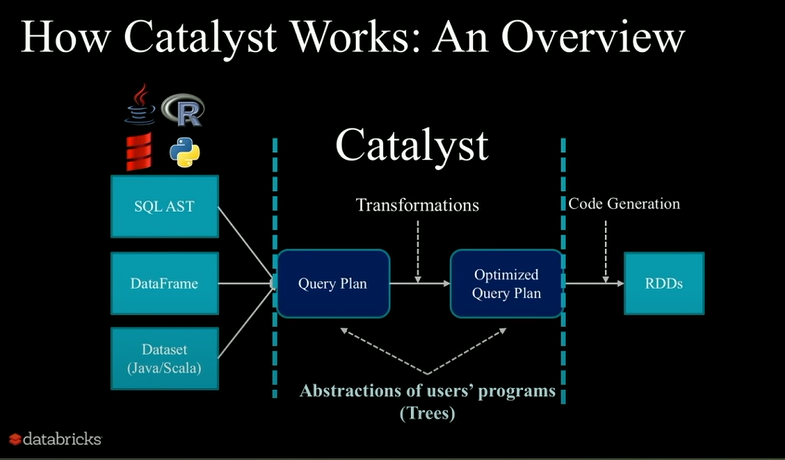
1

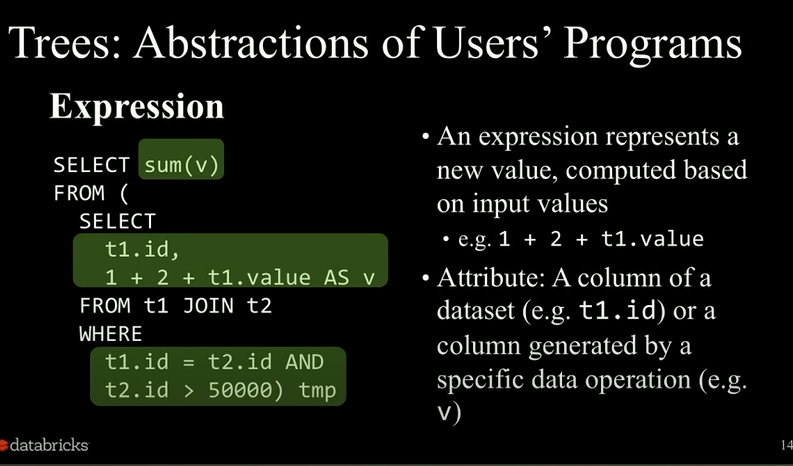
maxSplitBytes = Minimum(maxPartitionBytes, bytesPerCore)

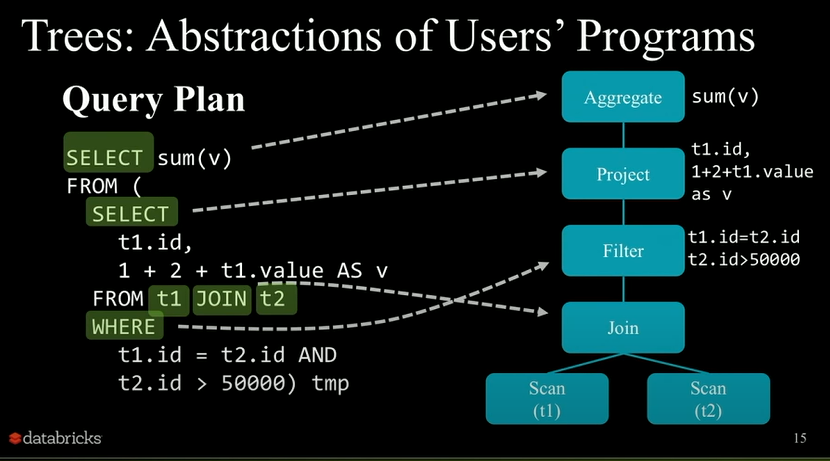
bytesPerCore =

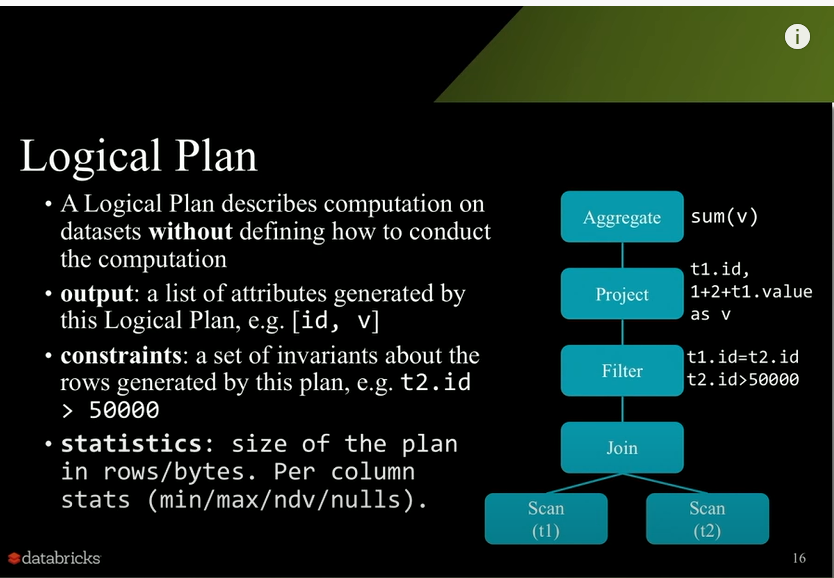
2

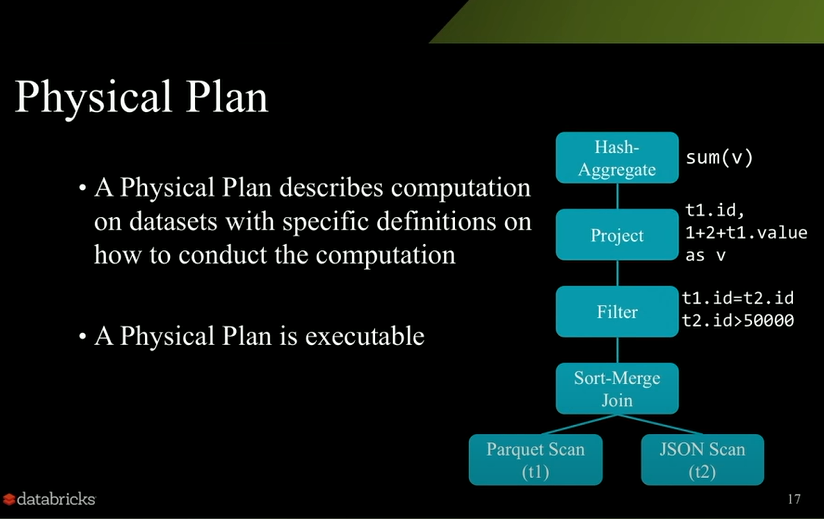
(Sum of sizes of all data files + No. of files \* openCostInBytes) / default.parallelism

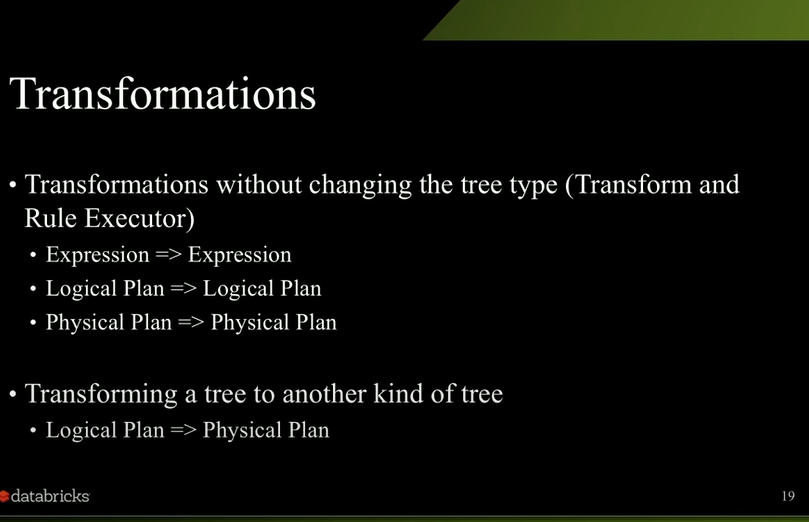




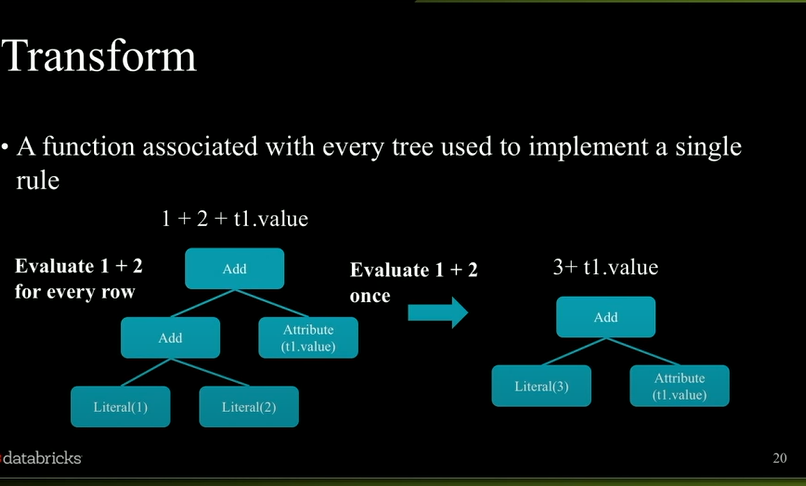


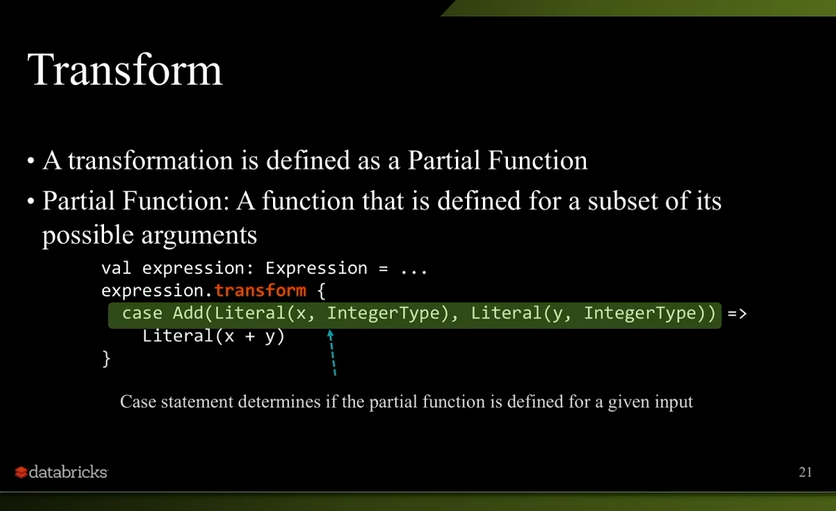


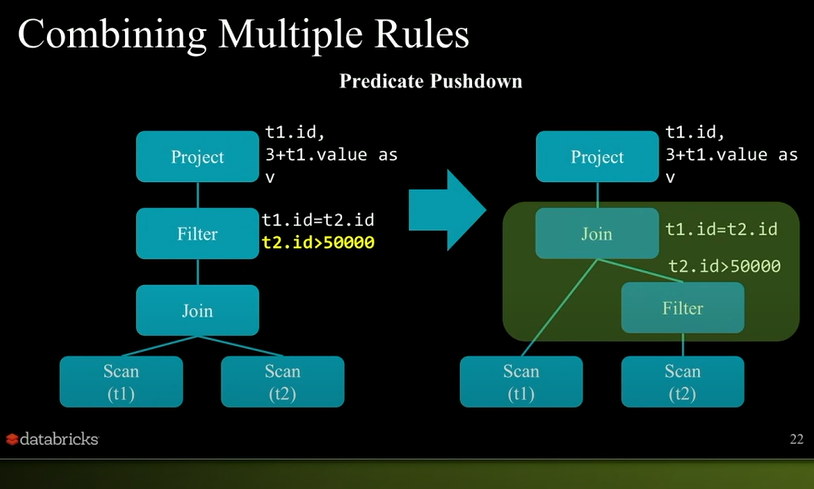


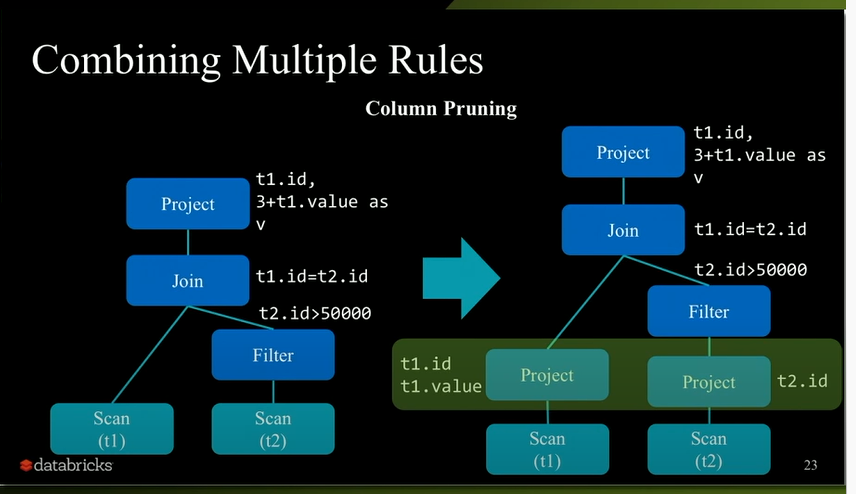


**CONSTANT FOLDING:**

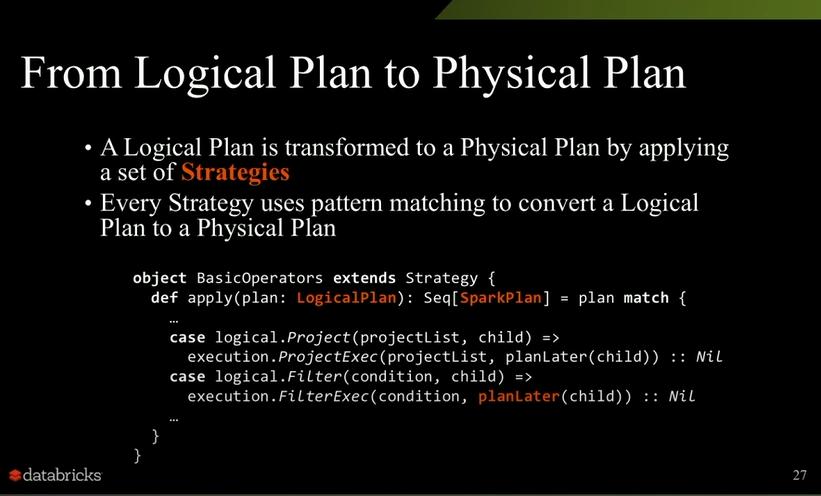




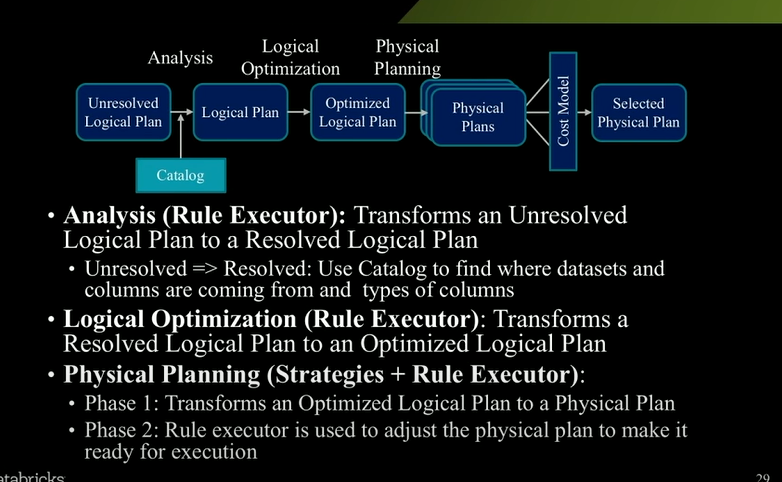


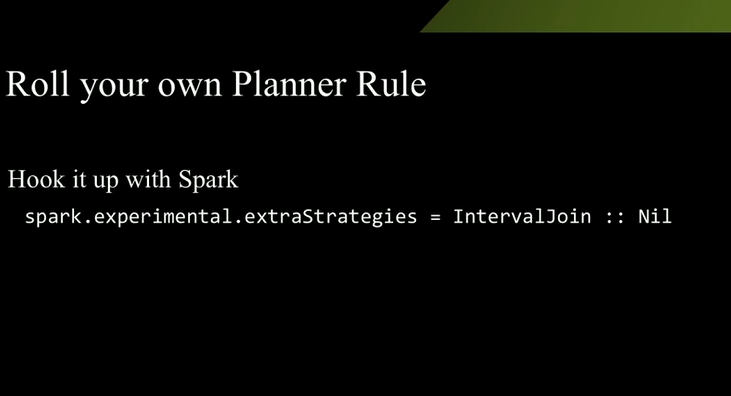


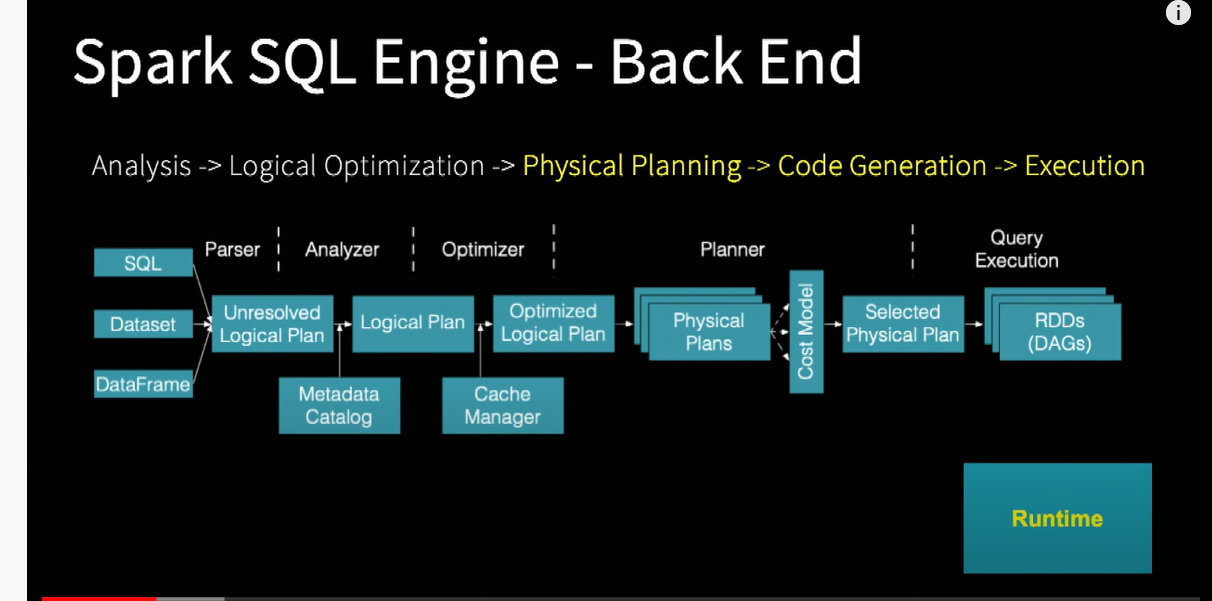


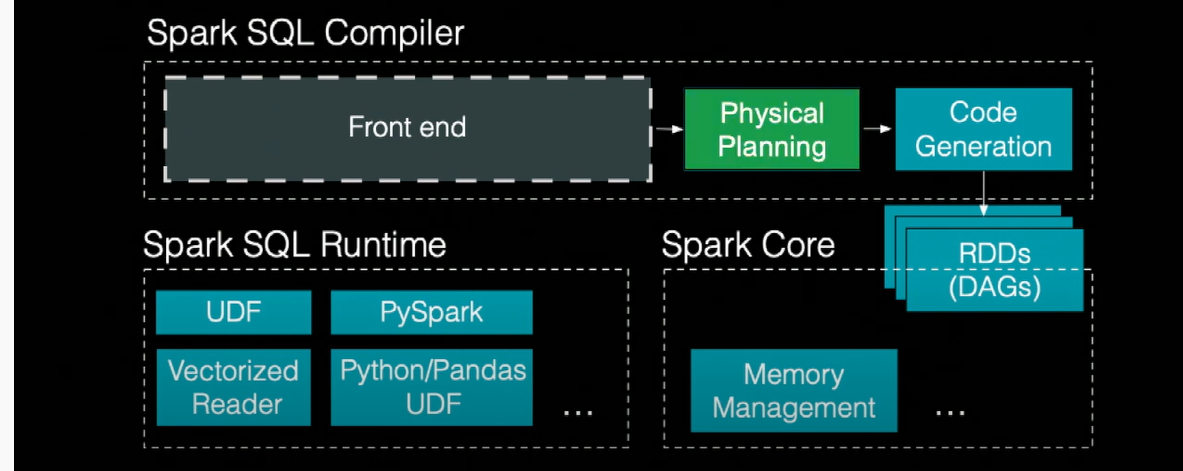


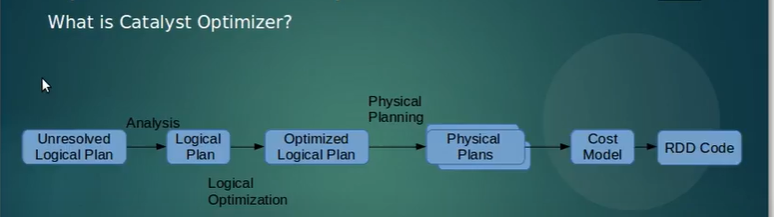












* 1. Utils class launch the sparkDriver process
  2. sparkEnv registers MapOutputTarcker and BlockManagerMaster
     + **Spark Runtime Environment** (SparkEnv) is the runtime environment with Spark’s public services that interact with each other to establish a distributed computing platform for a Spark application.

Spark Runtime Environment is represented by a [SparkEnv](https://mallikarjuna_g.gitbooks.io/spark/content/spark-sparkenv.html" \l "SparkEnv) object that holds all the required runtime services for a running Spark application with separate environments for the [driver](https://mallikarjuna_g.gitbooks.io/spark/content/spark-sparkenv.html#createDriverEnv) and [executors](https://mallikarjuna_g.gitbooks.io/spark/content/spark-sparkenv.html#createExecutorEnv).

* + - MapOutputTracker is a Spark service that runs on the driver and executors that [tracks the locations of the shuffle map outputs](https://mallikarjuna_g.gitbooks.io/spark/content/spark-service-mapoutputtracker.html#mapStatuses) (i.e. MapStatuses per shuffle).

There are two concrete MapOutputTrackers, i.e. one for the driver and another for executors:

* [MapOutputTrackerMaster](https://mallikarjuna_g.gitbooks.io/spark/content/spark-service-MapOutputTrackerMaster.html) for the driver
* [MapOutputTrackerWorker](https://mallikarjuna_g.gitbooks.io/spark/content/spark-service-MapOutputTrackerWorker.html) for executors

Given different runtime environments of the driver and executors, accessing the current MapOutputTracker is possible using [SparkEnv](https://mallikarjuna_g.gitbooks.io/spark/content/spark-sparkenv.html" \l "get).

* + - **Block Manager** (BlockManager) is a key-value store for blocks of data (simply blocks) in Spark. BlockManager acts as a local cache that runs on every "node" in a Spark application, i.e. the [driver](https://mallikarjuna_g.gitbooks.io/spark/content/spark-driver.html) and [executors](https://mallikarjuna_g.gitbooks.io/spark/content/spark-executor.html) (and is created right when [SparkEnv is](https://mallikarjuna_g.gitbooks.io/spark/content/spark-sparkenv.html" \l "NettyBlockTransferService)).

BlockManager provides interface for uploading and fetching blocks both locally and remotely using various stores, i.e. [memory, disk, and off-heap](https://mallikarjuna_g.gitbooks.io/spark/content/spark-blockmanager.html#stores).

When [BlockManager is created](https://mallikarjuna_g.gitbooks.io/spark/content/spark-blockmanager.html" \l "creating-instance), it creates its own private instances of [DiskBlockManager](https://mallikarjuna_g.gitbooks.io/spark/content/spark-DiskBlockManager.html), [BlockInfoManager](https://mallikarjuna_g.gitbooks.io/spark/content/spark-BlockInfoManager.html), [MemoryStore](https://mallikarjuna_g.gitbooks.io/spark/content/spark-MemoryStore.html) and [DiskStore](https://mallikarjuna_g.gitbooks.io/spark/content/spark-DiskStore.html) (that it immediately wires together, i.e. BlockInfoManager with MemoryStore and DiskStore with DiskBlockManager).

The common idiom in Spark to access a BlockManager regardless of a location, i.e. the driver or executors, is through [SparkEnv](https://mallikarjuna_g.gitbooks.io/spark/content/spark-sparkenv.html" \l "get):

SparkEnv.get.blockManager

A BlockManager is a [BlockDataManager](https://mallikarjuna_g.gitbooks.io/spark/content/spark-blockdatamanager.html), i.e. manages the storage for blocks that can represent cached RDD partitions, intermediate shuffle outputs, broadcasts, etc. It is also a [BlockEvictionHandler](https://mallikarjuna_g.gitbooks.io/spark/content/spark-blockmanager.html" \l "BlockEvictionHandler) that drops a block from memory and storing it on a disk if applicable.

* + - ShuffleManager is the [pluggable mechanism](https://mallikarjuna_g.gitbooks.io/spark/content/spark-shuffle-manager.html#contract) for **shuffle systems** that track [shuffle dependencies for ShuffleMapStage](https://mallikarjuna_g.gitbooks.io/spark/content/spark-dagscheduler-ShuffleMapStage.html) on the driver and executors. The setting [spark.shuffle.manager](https://mallikarjuna_g.gitbooks.io/spark/content/spark-shuffle-manager.html" \l "spark_shuffle_manager) sets up the default shuffle manager.The driver and executor access the current ShuffleManager using [SparkEnv](https://mallikarjuna_g.gitbooks.io/spark/content/spark-sparkenv.html" \l "shuffleManager).The driver registers shuffles with a shuffle manager, and executors (or tasks running locally in the driver) can ask to read and write data.It is network-addressable, i.e. it is available on a host and port.There can be many shuffle services running simultaneously and a driver registers with all of them when [CoarseGrainedSchedulerBackend](https://mallikarjuna_g.gitbooks.io/spark/content/spark-scheduler-backends.html) is used.