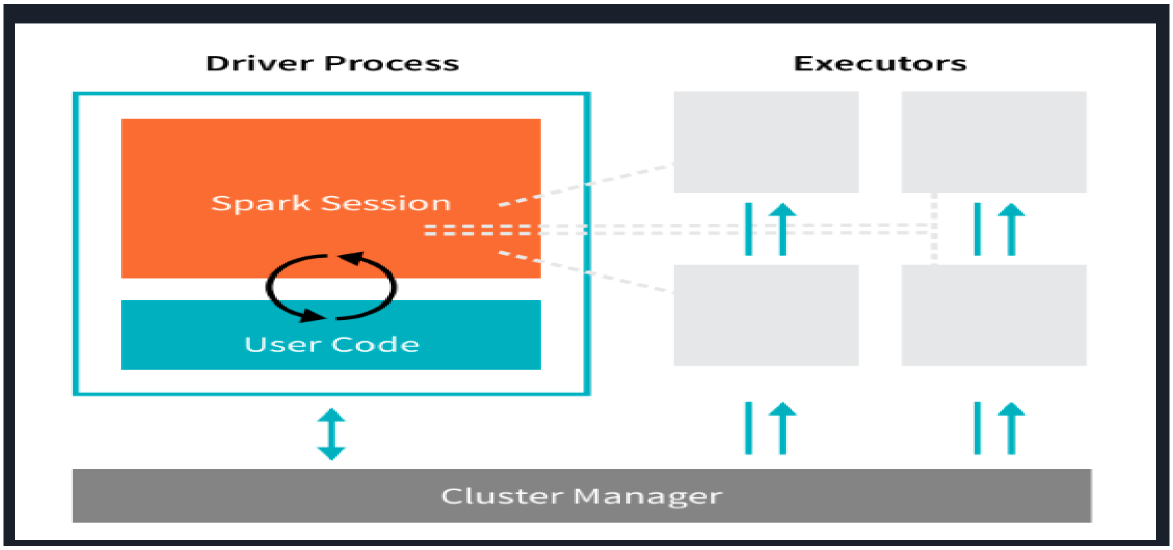
SPARK SUMMARY DOCUMENT

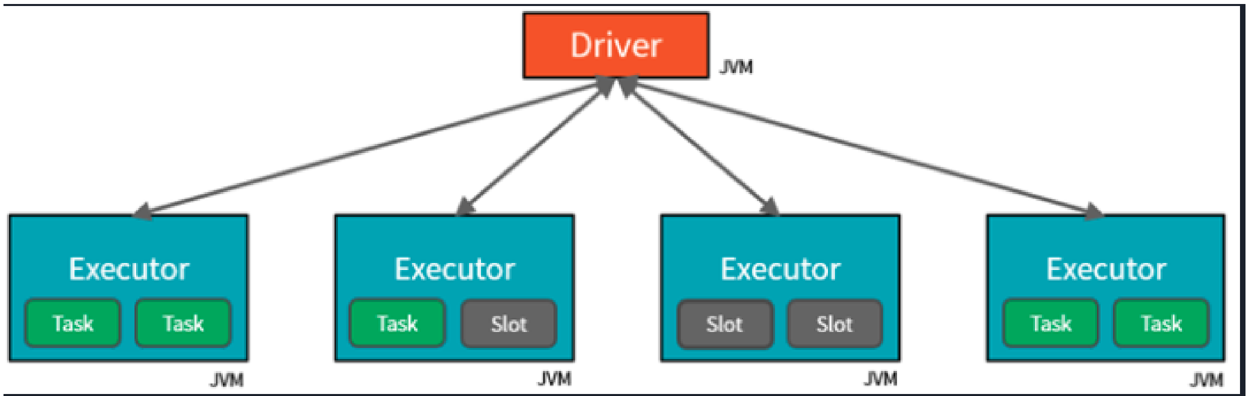
TABLE OF CONTENTS

SPARK CORE

Concept	Definition	Example/Code
Spark	Unified computing engine and a set of libraries for big data.	
Spark Components	• Driver • Cluster Manager • Executors • Client	



Client	• Responsible for starting the driver program. • Can be done via spark-submit or a spark-shell script or a custom application using spark api. • Prepares classpath and all application configuration for spark application. • Also passes required arguments	
Driver	• Orchestrates and monitors spark execution. • We have 1 driver per spark application. • Driver is responsible for requesting resources from the cluster manager. • Responsible for breaking application logic into stages and tasks. • Responsible for sending tasks to executors and Collecting results back. • Not responsible for scheduling tasks on executors	



Executors	• Accepts tasks from the driver, executes them and returns the results. • Each executor has several task slots or CPU cores(not physical). • Slots indicate the number of tasks that can be run in parallel. • We can set the task slots 2-3 times the number of actual physical CPU cores. • This is set by the spark.executor.cores.	
Worker	• Any node that can run application code in the cluster. • Has a finite or fixed number of Executors allocated at any point in time.	

Cluster Manager	• Keeps track of resources available.
	• Spark runs in local mode or inside a cluster.
	• Spark has below cluster managers:
	  ○ Standalone
	    ■ Most basic and default
	    ■ Can run only spark applications	

	○ Yarn ■ Hadoop's resource manager ■ Can run spark and other java applications ○ Mesos ■ Scalable and fault tolerant. ■ Can run spark as well several python,java and other applications ○ Kubernetes	
Slots	● Present inside an executor ● Slots indicate the number of tasks that can be run in parallel.	
Spark session	● Sparksession instance is the handle through which spark executes spark user defined manipulations across cluster ● There is 1-1 between spark session and spark application ● Gives access to low level configs and contexts ● subsume previous entry points to the Spark like the SparkContext, SQLContext, HiveContext, SparkConf, and StreamingContext ● Spark Session allows you to create JVM runtime parameters, define DataFrames and Datasets, read from data sources, access catalog metadata, and issue Spark SQL queries.	

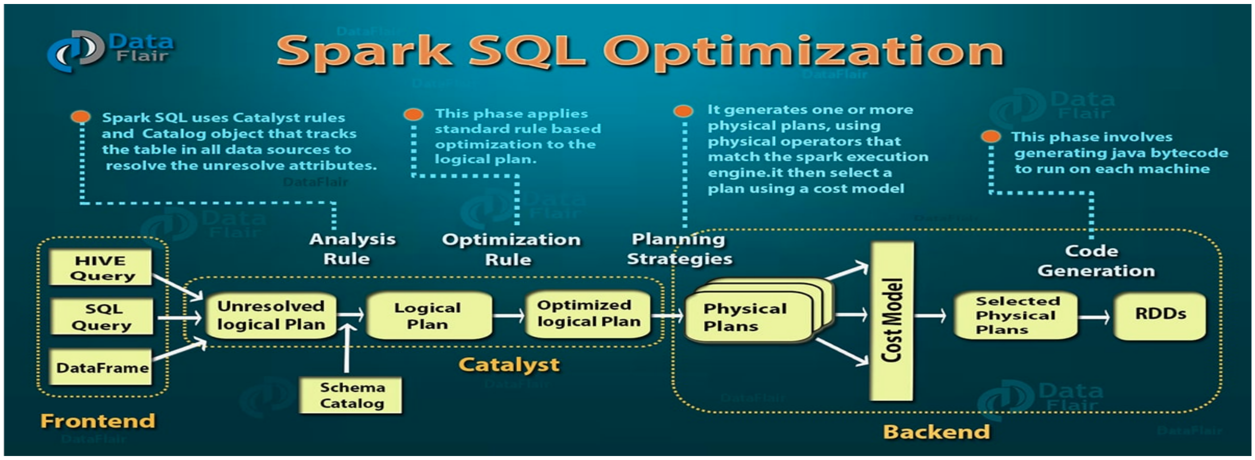
Dataframes	- Most common structured api in spark and represents a table of rows and columns in spark. - It can be considered as a spreadsheet with data across multiple computers	
Partitions	- A Partition is a logical chunk of your DataFrame - Data is split into Partitions so that each Executor core/thread can operate on a single part, enabling parallelization. - Spark by default creates 1 partition for every 128 MB of the file.	Eg: If you have 4 data partitions and you have 4 executor cores/threads, you can process everything in parallel, in a single pass. Syntax: rdd = sc.parallelize(range(1,11)) rdd.getNumPartitions() sc.defaultParallelism
Glom	returns an RDD having the elements within each partition in a separate list	rdd.glom().collect()
Transformations	- Dataframes are immutable. - Transformations are operations where we instruct Spark as to how to go about making changes to our dataframe. - Transformations are lazily evaluated. - They eventually return another DataFrame - There are two types of transformations: narrow and wide transformation	

Narrow transformations	• In this transformation, we do not perform a shuffle on the parent rdd to get a new rdd. • The data required to compute the records in a single partition in child rdd resides at most one partition of the parent RDD. • narrow dependencies can be grouped into one stage. • If we partition buckets for a column on which we want to perform joins, we can avoid shuffle	• Map • Mapvalues • Filter • Union • Flatmap • Mappartitions • Join
Wide transformation	• In this transformation, we perform a shuffle on the parent rdd to get new rdd • The data required to compute the records in a single partition in child rdd resides at more than one partition of the parent RDD.	• groupBy • leftouterjoin • Groupbykey • Reducebykey • Distinct • Intersection • repartition

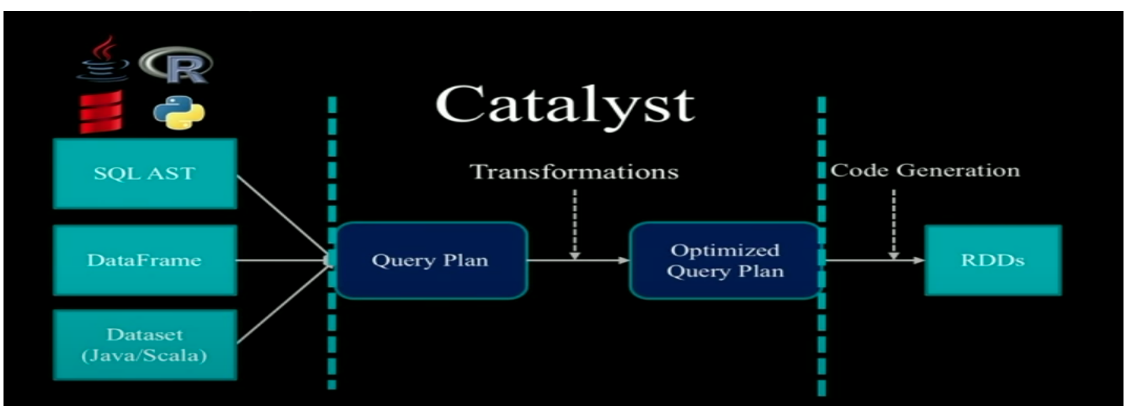
Action	• An action instructs spark to compute a result from a series of transformations.
	• 3 kinds of actions:
	  ○ Action to view data on console.
	  ○ Action to collect data to native objects.
	  ○ Action to write to an output data source.
		• `count`
		• `collect`
		• `take(n)`
		• `top()`
		• `countByValue()`
		• `reduce()`
		• `fold()`
		• `aggregate()`
		• `foreach()`

Lazy evaluation	• Spark pipelines all the transformations and performs the actual execution at the end. • This is called lazy evaluation. • Done to improve performance and	

	optimize execution	
Datasets	● Statistically typed code available in java and python.	
DAG	● DAG stands for directed acyclic graph. ● Has no directed circles. ● Used to construct execution plans for spark code, form different stages and optimize on the same.	
Spark Lifecycle (Explain Plan)	● User submits application to spark cluster. ● Driver takes spark code and identifies transformations and actions, ● Logical plan(consisting of tasks and stages) is constructed here which is passed through an optimizer and a physical plan is constructed. ● In physical plan stages are combined based on the nature of transformations. ● Wide transformations which require a shuffle to be performed , and any reads from external sources mark the start of a new stage.	Analysis->logical optimization->physical planning->code generation

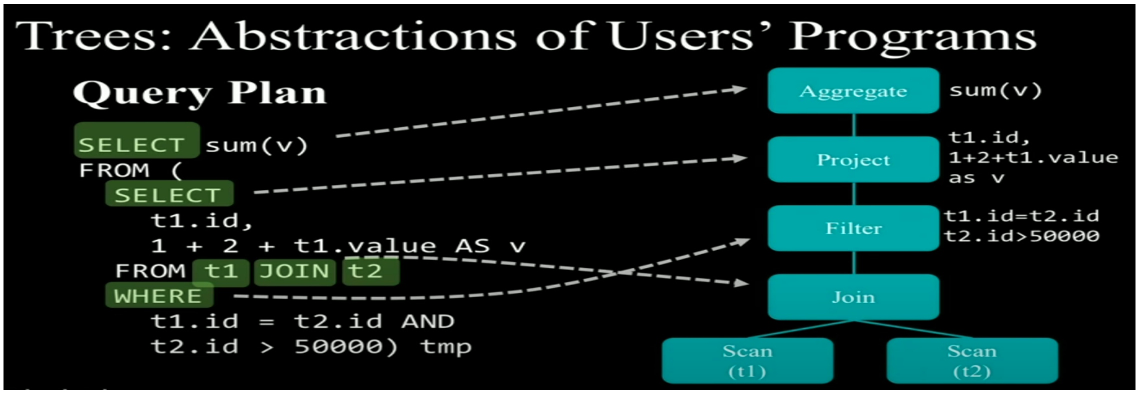
<table>
<tr><td colspan="3">
</td></tr>
<tr><td>Catalyst optimizer</td><td>

- Interface between high level structured API's and low level RDD's.
- Converts initial query plan to optimized version.
- The user programs are defined as trees and what catalyst does is convert one tree to another.
- There are 2 kinds of query plans:-
  - Logical plan
  - Physical plan.

</td><td></td></tr>
<tr><td colspan="3">
</td></tr>
<tr><td>Catalyst optimizer trees</td><td>

- Abstraction of a user program.

</td><td></td></tr>
</table>



Logical plan	• Abstraction of user programs without execution details. • Has list of output columns and property called constraints	
Physical plan	• Describes types of computation	
Types of transformations in optimizer	• Expression to expression: Techniques used are: ○ Predicate pushdown(pushing filter downstream so that less data is read) ○ Column pruning - get only required columns from DB. • Logical plan to logical plan • Logical plan to physical plan	
Dynamic partition pruning	• Intended to skip over data you do not need in the results of a query • Done at Logical planning level to find the dimensional filter and propagate across the join to the other side of the scan. • Done at Physical level to wire it together in a way that this filter executes only once on the dimension side.	spark.sql.optimizer. dynamicPartitionPruning .enabled

	• Then the results of the filter gets reused directly in the scan of the table. And with this two fold approach we can achieve significant speed ups in many queries in Spark. • Best use case is for joining fact and dimension tables	
Adaptive Query Execution	• Cost optimization is done based on initial statistics which is different from runtime statistics. • Aqe aims to optimize on the same. • This is done at stage boundaries. • Not applicable for streaming queries	spark.sql.adaptive.enabled=true needs to be set
AQE Scenarios	• Data Skew - partitions are uneven which leads to longer execution time, aqe has a skew partishioner which breaks down larger partitions. • Dynamically changing partitions - if aqe feels less partitions are needed than what is allocated it does a coalesce. • Introducing broadcast joins instead of sort merge joins - if we are joining 2 tables and 1 is smaller than 10mb we can convert the join to a broadcast thereby reducing shuffles.	• spark.sql.adaptive.coalescePartitions.enabled • spark.sql.adaptive.coalescePartitions.minPartitionNum • spark.sql.adaptive.coalescePartitions.initialPartitionNum • spark.sql.adaptive.advisoryPartitionSizeInBytes • spark.sql.adaptive.localSchuffReader.enabled - convert sort merge join to broadcast join
Spark submit	• Spark submit command is used to submit a job to a spark cluster once an application is packaged. • Arguments of spark submit are	Syntax: /bin/spark-submit \\ --class <main-class> \\

<table>
<tr>
<td></td>
<td>below.

● --class: The entry point for your application (e.g. org.apache.spark.examples.SparkPi)
● --master: The master url for the cluster (e.g. spark://23.195.26.187:7077)
● --deploy-mode: Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client) †
● --conf: Arbitrary Spark configuration property in key=value format. For values that contain spaces, wrap “key=value” in quotes (as shown).
● application-jar: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes.
● application-arguments: Arguments passed to the main method of your main class, if any.</td>
<td>--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key>=<value> \

... # other options
<application-jar> \
[application-arguments]</td>
</tr>
<tr>
<td>Spark submit master Url</td>
<td>Spark master can have the below options:

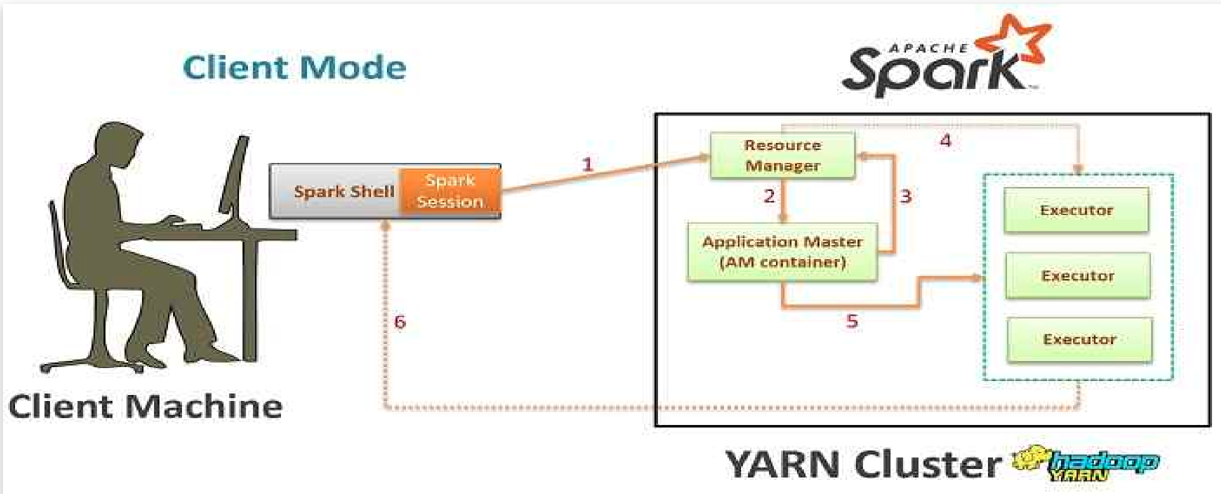
● Local - Run Spark locally with one worker thread (i.e. no parallelism at all)
● local[K]- Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).
● local[K,F] - Run Spark locally with K worker threads and F maxFailures</td>
<td></td>
</tr>
</table>

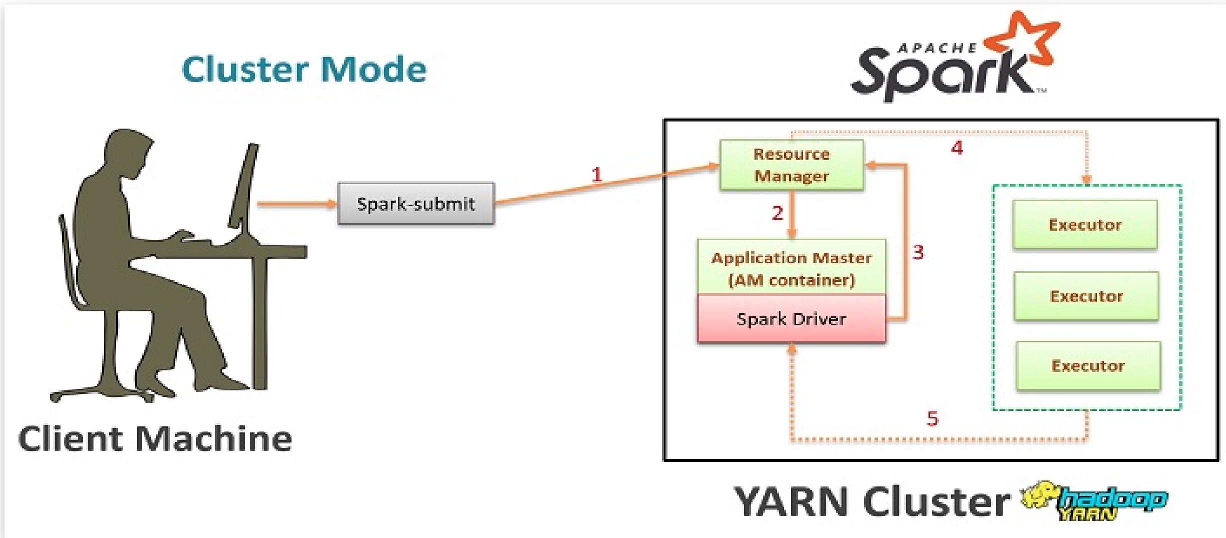
	(see spark.task.maxFailures for an explanation of this variable) ● **local[*]** - Run Spark locally with as many worker threads as logical cores on your machine. ● **local[*,F]** - Run Spark locally with as many worker threads as logical cores on your machine and F maxFailures. ● **spark://HOST:PORT** - Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default. ● **spark://HOST1:PORT1,HOST2:PORT2** - Connect to the given Spark standalone cluster with standby masters with Zookeeper. Port :7077 by default ● **mesos://HOST:PORT**- Connect to the given Mesos cluster. ● **Yarn** - Connect to a YARN cluster in client or cluster mode depending on the value of --deploy-mode.	
Dag scheduler	● DAGScheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling. ● It transforms a logical execution plan to a physical execution plan (using stages). ● Uses a fifo scheduler ● Fair scheduler runs jobs in round robin fashion ● To enable fair scheduler set spark.scheduler.mode= Fair	
Task scheduler	● These schedulers get sets of tasks submitted to them from the DAGScheduler for each stage, and are responsible for sending the tasks to the cluster, running them,	

	retrying if there are failures, and mitigating stragglers. • They return events to the DAGScheduler.	
Jobs	• A Job is a sequence of stages, triggered by an action such as count(), collect(), read() or write(). • Each parallelized action is referred to as a Job. • The results of each Job (parallelized/distributed action) are returned to the Driver from the Executor. • Depending on the work required, multiple Jobs will be required.	
Stages	• Each job that gets divided into smaller sets of tasks is a stage. • Stage is a step in a physical execution plan • A stage is a set of parallel tasks - one task per partition • Each stage contains a sequence of narrow transformations	
Tasks	• A task is a unit of work that is sent to the executor. • Each stage has some tasks, one task per partition. • The same task is done over different partitions of the RDD.	
Spark config	• spark.executor.instances: Number of executors for the spark application. • spark.executor.memory: Amount of memory to use for each executor	

	that runs the task. • spark.executor.cores: Number of concurrent tasks an executor can run. • spark.driver.memory: Amount of memory to use for the driver. • spark.driver.cores: Number of virtual cores to use for the driver process. • spark.sql.shuffle.partitions: Number of partitions to use when shuffling data for joins or aggregations. • spark.default.parallelism: Default number of partitions in resilient distributed datasets (RDDs) returned by transformations like join and aggregations. • Yarn has different configuration settings which can be looked at as well.	
Additional config properties	• spark.speculation - relaunch one or more tasks if they are running slowly in a stage. • spark.memory.fraction - ○ percentage of memory used for computation in shuffles,joins,sort and aggregations ○ If gc is invoked multiple times before task completed, decrease this so that amount of memory used for caching is reduced	
Dynamic allocation	• scales the number of executors registered with this application up and down based on the workload.	• spark.dynamicAllocation.enabled • spark.dynamicAllocation.minExecutors

		• spark.dynamicAllocation.maxExecutors • spark.dynamicAllocation.initialExecutors
Execution modes	Spark execution can be in one of 3 modes: • Local mode. • Client Mode • Cluster mode	
Local Mode	• This is like running a program on someone's laptop or desktop using a single JVM. • You should have defined & used spark context objects, imported spark libraries and processed data from your local system files. • Everything runs LOCALLY and there is no concept of NODE involved and nothing runs in DISTRIBUTED mode. • The driver & the executor are created inside a single JVM process.	
Client mode	• In Client mode, the Driver is started in the Local machine i.e. Driver is outside of the Cluster. • The Executors will be running inside the Cluster. • Entire application is dependent on the Local machine since the Driver resides in here. • In case of any issue with the local machine , the driver will go off . Subsequently the entire application will go off.	

<table>
<tr><td></td><td>Good for Debugging or Testing.</td><td></td></tr>
<tr><td colspan="3">
</td></tr>
<tr><td>Cluster mode</td><td>In Cluster Mode, the Driver & Executor both run inside the Cluster.
You submit the spark job from your local machine to a Cluster machine inside the Cluster
Such machines are usually called Edge Nodes.
This is the approach used in Production use cases.</td><td></td></tr>
</table>



Serialization	• There are 2 types of serialization supported in spark:
	  ○ Java serialization: default serialization, easy but slow.
	  ○ Kryo Serialization -
	    ■ 10 times faster than default serialization
	    ■ Requires custom registration of classes.
	    ■ Best in case of network intensive applications.
	• All data saved to disk in spark is serialized by default
		Kryo:
		
		conf.set("Spark.serializer"," org.apache.spark serializer.kryoserializer")

Shuffle	• A Shuffle refers to an operation where data is re-partitioned across a Cluster - i.e. when data needs to move between executors(comparison done). • Join and any operation that ends with ByKey(wide transformation)	

	will trigger a Shuffle.
	• It is a costly operation because a lot of data can be sent via the network.
	• Types of shuffle implementations in spark are:
	  ○ Bypassmergesortshufflewriter
	  ○ Sortshufflewriter
	  ○ Unshuffle writer	

Caching	• Shuffle files are by definition temporary files and will eventually be removed. • However, we can cache data explicitly to accomplish the same thing that happens inadvertently with shuffle files. • This is used for optimizing the spark query. • LRU cache is used by default. • Cache is eager by default, we can change this by using keyword lazy.	spark.catalog.cacheTable("tableName") dataFrame.cache() Unpersist() - used to forcibly remove rdd from head `spark.sql("cache lazy table table_name")`

Uncache Table	• Removes the entries and associated data from the in-memory and/or on-disk cache for a given table or view.
	• The underlying entries should already have been brought to cache by the previous CACHE TABLE operation.
	• The UNCACHE TABLE on a non-existent table throws an Exception if IF EXISTS is not specified.
		Syntax
		
		`UNCACHE TABLE [ IF EXISTS ] table_name`
		
		Parameters
		
		• `var: table_name`
		• `def: The name of the table or view to be uncached.`
		Example:
		
		`UNCACHE TABLE t1;`

Persist Options	• MEMORY_ONLY-	

	◦ rdd stored in memory • MEMORY_ONLY_SER - ◦ rdd stored in memory ◦ Cost of serialization/deserialization incurred • MEMORY_AND_ DISK ◦ Default storage option for cache ◦ First we store in memory, if space is not available we push the oldest partition to disk • MEMORY_AND_DISK_SER ◦ Above with data serialized • MEMORY_AND_DISK_DESER ◦ Default storage level of spark since 3.1.1 • MEMORY_ONLY_2 ◦ 2 replicas created of the rdd ◦ Do only if rdd is critical • DISK_ONLY ◦ Keep replication factor to 1	
RDD	• Rdd represents an immutable partitioned collection of records that can be operated on in parallel. • RDDS are java,python or scala objects. • Dataframes and datasets are internally converted into rdd’s	
Types of RDD	• You have rdd interfaces and multiple implementations of the	

	same.
	• Things that we specify in an rdd implementation are
	  ○ Set of partitions
	  ○ List of dependencies on parent RDD's
	  ○ Function to compute a partition given to a parent.
	  ○ Optional preferred locations.
	  ○ Optional partition information.
	  ○ Broadly there are two types of rdd's:-
	    ■ Generic RDD -
	    ■ Key Value RDD custom partitions by keys are done.	

Broadcast Variables	• Shared immutable variable cached on every machine on the cluster • Present in the driver and sent across clusters. • Broadcastjoin is used when we are joining 2 tables and want to broadcast a complete table(size limit 10mb) • No shuffle incurred	
Accumulators	• Accumulators are used to write data across a cluster. • Eg. We are reading a file in chunks and we want to count the number of blank lines. • We can use an accumulator to do this.	

<table>
<tr><td></td><td>Executors just update the value.</td><td></td></tr>
<tr><td>Repartition</td><td>Repartition is a method in spark which is used to perform a full shuffle on the data present and create partitions based on the user’s input.
Method 1 : Repartition using Column Name
Method 2 : Repartition using integer value
By default, 200 partitions are created if the number is not specified in the repartition clause.
Use Repartition only when you want to increase the number of partitions (or) if you want to perform a full shuffle on the data.</td><td>Repartition using Column Name
<code>df = df.repartition('_1')</code>
Repartition using integer value
<code>df = df.repartition(3)</code></td></tr>
<tr><td>Coalesce</td><td>Used to reduce the number of partitions in a dataframe.
We cannot increase the number of partitions using coalesce
Unlike repartition, coalesce doesn’t perform a shuffle to create the partitions.
Suppose there are 100 partitions with 10 records in each partition, and if the partition size is reduced to 50, it would retain 50 partitions and append the other values to these existing partitions thereby having 20 records in each partition.</td><td><code>df3 = df.coalesce(2)</code></td></tr>
</table>

Repartition vs Coalesce	- Repartition can be used under these scenarios: - when you want your output partitions to be of equally distributed chunks. - increase the number of partitions. - perform a shuffle of the data and create partitions. - Coalesce can be used under the following scenarios, - when you want to decrease the number of partitions. - in order to avoid shuffle when partitions are decreasing.	
Catalog	- Catalog is the interface for managing a metastore (aka metadata catalog) of relational entities (e.g. database(s), tables, functions, table columns and temporary views). - Catalog is available using SparkSession.catalog property.	

SPARK READ/WRITE

Concept	Definition	Example/Code
Generic File options	• Ignore Corrupt Files • Ignore Missing Files • pathGlobFilter- include files with file names matching the pattern. • recursiveFileLookup - used to recursively load files and it disables partition inferring. Its default value is false	spark.sql ("set spark.sql.files.ignoreCorruptFiles=true") spark.sql.files.ignoreMissingFiles spark.read. load("examples/src/main/resources/dir1", format="parquet", pathGlobFilter="*.parquet") recursive_loaded_df = spark.read.format("parquet") .option("recursiveFileLookup", "true") .load("examples/src/main/resources/dir1")

Read	• The read modes are:
	  ○ permissive — All fields are set to null and corrupted records are placed in a string column called _corrupt_record
	  ○ dropMalformed — Drops all rows containing corrupt records.
		Syntax:
		
		DataFrameReader.format(…).option(“key”, “value”).schema(…).load()
		
		FOSL

	○ failFast — Fails when corrupt records are encountered. ● If you want to control the maximum partition size while reading files. ○ spark.files.maxpartitionBytes	
Write	● DataFrame writer default format is the parquet file Save modes are: ● append — appends output data to files that already exist ● overwrite — completely overwrites any data present at the destination ● errorIfExists — Spark throws an error if data already exists at the destination ● ignore — if data exists do nothing with the dataFrame	Syntax: DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save() FOPBSS
CSV		**Read:** csvFile = spark.read.format("json") .option("mode", "FAILFAST") .option("inferSchema","true") .load("/data/flight-data/json/2010-summary.json") .show(5) **Write:** csvFile.write.format("csv")

		.mode("overwrite") .option("sep", "\t"). .save("/tmp/my-tsv-file.tsv")
Json	• Spark SQL can automatically infer the schema of a JSON dataset and load it as a Dataset[Row]. • Each line must contain a separate, self-contained valid JSON object • For a regular multi-line JSON file, set the multiLine option to true. • Default date format in json is yyyy-MM-dd	**Read:** spark.read.format("json") .option("mode", "FAILFAST") .option("inferSchema", "true") .load("/data/flight-data/json/2010-summary.json").show(5) **Write:** csvFile.write.format("json") .mode("overwrite") .save("/tmp/my-json-file.json")
Text	• When you write a text file, you need to be sure to have only one string column; otherwise, the write will fail	**Read:** lines = sc.textFile("/home/deepak/test1.txt") **Write:** lines.coalesce(1).write.format("text") .option("header","false") .mode("append").save("output.txt")
Parquet	• Parquet is a columnar format that is supported by many other data processing systems • When reading Parquet files, all columns are automatically converted to be nullable for compatibility reasons.	**Read:** spark.read.format("parquet") .load("/data/flight-data/parquet/summary.parquet") .show(5) **Write:** csvFile.write.format("parquet") .mode("overwrite") .save("/tmp/my-parquet-file.parquet")

Orc		**Read:** spark.read.format("orc") .load("/data/flight-data/orc/2010-summary.orc") .show(5) **Write:** csvFile.write.format("orc") .mode("overwrite") .save("/tmp/my-json-file.orc")
Database		driver = "org.sqlite.JDBC" path = "/data/flight-data/jdbc/my-sqlite.db" url = "jdbc:sqlite:" + path tablename = "flight_info" **Read:** dbDataFrame = spark.read.format("jdbc") .option("url", url) .option("dbtable", tablename) .option("driver", driver).load() **Write:** newPath = "jdbc:sqlite://tmp/my-sqlite.db" csvFile.write .jdbc(newPath, tablename, mode="overwrite", properties=props)

TABLES AND VIEWS

Global Managed Table	• A managed table is a Spark SQL table for which Spark manages both the data and the metadata. • A global managed table is available across all clusters. • When you drop the table both data and metadata get dropped.	`dataframe.write.saveAsTable("my_table")`
Global Unmanaged/External Table	• Spark manages the metadata, while you control the data location. As soon as you add the 'path' option in the dataframe writer it will be treated as a global external/unmanaged table. • When you drop a table only metadata gets dropped. • A global unmanaged/external table is available across all clusters.	`Dataframe.write.option('path', "<your-storage-path>").saveAsTable("my_table")`
Local Table (a.k.a) Temporary Table (a.k.a) Temporary View	• Spark session scoped. • A local table is not accessible from other clusters (or if using a databricks notebook not in other notebooks as well). • It is not registered in	`Dataframe.createOrReplaceTempView()`

<table>
<tr><td></td><td>the metastore.</td><td></td></tr>
<tr><td>Global Temporary View</td><td>Spark application scoped global temporary views are tied to a system preserved temporary database global_temp.
This view can be shared across different spark sessions (or if using databricks notebooks, then shared across notebooks).</td><td><code>dataframe.createOrReplaceGlobalTempView("my_global_view")</code>

<code>can be accessed as,</code>
<code>spark.read.table("global_temp.my_global_view")</code></td></tr>
<tr><td>Global Permanent View</td><td>Persist a data frame as a permanent view.
The view definition is recorded in the underlying metastore.
You can only create a permanent view on a global managed table or global unmanaged table. Not allowed to create a permanent view on top of any temporary views or dataframe.
Note: Permanent views are only available in SQL API — not available in dataframe API</td><td><code>spark.sql("CREATE VIEW permanent_view AS SELECT * FROM table")</code></td></tr>
</table>

DATAFRAMES

Function	Use	Example

CreateDataFrame	• Creates a DataFrame from an RDD, a list or a pandas.DataFrame
	• Package:
	  ○ pyspark.sql.sparkSession
		Syntax:
		
		`createDataFrame(data, schema=None,samplingRatio=None, verifySchema=True)`
		
		• data:
		  ○ type:dataRDD or iterable
		  ○ def: an RDD of any kind of SQL data representation
		
		• schema:
		  ○ type: pyspark.sql.types.DataType, str or list, optional
		  ○ def:
		
		• samplingRatio:
		  ○ type:float, optional
		  ○ def:the sample ratio of rows used for inferring
		
		• verifySchema
		  ○ type:bool, optional
		  ○ def:verify data types of every row against schema. Enabled by default.
		
		Example:
		
		`l = [('Alice', 1)]`
		`spark.createDataFrame(l).collect()`

printSchema	• Prints out the schema in the tree format.
	• Package:
	  ○ pyspark.sql.DataFrame
		Syntax:
		`DataFrame.printSchema()`
		Example:
		`df.printSchema()`

createOrReplaceTempView	• Creates or replaces a local temporary view with this DataFrame.
	• Package:
	  ○ pyspark.sql.DataFrame
		Syntax:
		`DataFrame.createOrReplaceTempView(name)`
		Example:
		`df.createOrReplaceTempView("people")`

Schema	• Returns the schema of this DataFrame as a pyspark.sql.types.StructType.
	• Package:
	  ○ pyspark.sql.DataFrame
		Syntax:
		`DataFrame.schema`
		Example:
		`df.schema`

col	• A column in a DataFrame.
	• Package:
	  ○ pyspark.sql.functions
		`Syntax:`
		`Example:`
		• `Select a column out of a DataFrame`
		`df.colName`
		`df["colName"]`
		• `Create from an expression`
		`df.colName + 1`

		`1 / df.colName`

row	- A row in DataFrame. - The fields in it can be accessed: - like attributes (row.key) - like dictionary values (row[key]) - Package: - pyspark.sql	`Syntax:`
		
		`Example:`
		`row = Row(name="Alice", age=11)`
		`row.name, row.age`

Null in spark	Use None	

expr	- Parses the expression string into the column that it represents - Package: - pyspark.sql.functions	**`Syntax:`**
		
		`expr(str)`
		
		`Example:`
		
		`df.select(expr("length(name)")).collect()`

select	- Projects a set of expressions and returns a new DataFrame. - Package: - pyspark.sql.	**`Syntax:`**
		
		`DataFrame.select(*cols)`
		    `cols` `type:str, Column, or list` `def:`

	DataFrame	▪ `column names (string) or expressions (Column).`
		▪ `If one of the column names is '*', that column is expanded to include all columns in the current DataFrame.`
		
		`Example:`
		
		`df.select('name', 'age').collect()`
		
		`df.select(df.name, (df.age + 10).alias('age')).collect()`

selectExpr	• Projects a set of SQL expressions and returns a new DataFrame.
	
	• This is a variant of select() that accepts SQL expressions.
	
	• Package:
	  ○ pyspark.sql.DataFrame
		`Syntax:`
		
		`DataFrame.selectExpr(*expr)`
		
		`Example:`
		
		`df.selectExpr("age * 2", "abs(age)").collect()`

withColumn	• Returns a new DataFrame by adding a column or replacing the existing column that has the same name.
	
	• The column expression must be an expression over
		`Syntax:`
		
		`DataFrame.withColumn(colName, col)`
		
		• `colName`
		  ○ `Type:str`
		  ○ `def: string, name of the new column.`
		
		• `col`
		  ○ `type:Column`

	this DataFrame; attempting to add a column from some other DataFrame will raise an error. • Package: ○ pyspark.sql.DataFrame	○ `def: a Column expression for the new column.` **Example:** `df.withColumn('age2', df.age + 2).collect()`

withColumnRenamed	• Returns a new DataFrame by renaming an existing column. This is a no-op if the schema doesn't contain the given column name.
	• Package:
	  ○ pyspark.sql.DataFrame
		Syntax:
		
		`DataFrame.withColumnRenamed(existing, new)`
		
		• `existing:`
		  ○ `Type:str`
		  ○ `def:string, name of the existing column to rename.`
		• `new:`
		  ○ `Type:str`
		  ○ `def:string, new name of the column.`
		
		Example:
		
		`df.withColumnRenamed('age', 'age2').collect()`

alias	• Returns this column aliased with a new name or names.
	• In the case of expressions that return more than one column, such as explode).
	• package:
	  ○ pyspark.sql.
		Syntax:
		
		`Column.alias(*alias, **kwargs)`
		
		`alias`
		`type: str`
		`def: desired column names (collects all positional arguments passed)`
		
		Example:

<table>
<tr><td></td><td>column</td><td><code>df.select(df.age.alias("age2")).collect()</code></td></tr>
<tr><td>cast</td><td>Convert the column into a different dataType.
Package:
pyspark.sql.Column</td><td>Syntax:
<code>Column.cast(dataType)</code>
Example:
<code>df.select(df.age.cast("string").alias('ages')).collect()</code></td></tr>
<tr><td>drop</td><td>Returns a new DataFrame that drops the specified column.
This is a no-op if the schema doesn't contain the given column name(s).
package:
pyspark.sql.DataFrame</td><td>Syntax:
<code>DataFrame.drop(*cols)</code>
cols:
type:str or :class:`Column`
def: a name of the column, or the Column to drop
Example:
<code>df.drop('age').collect()</code>
<code>df.drop(df.age).collect()</code></td></tr>
<tr><td>dropDuplicates</td><td>Return a new DataFrame with duplicate rows removed, optionally only considering certain columns.
For a static batch DataFrame, it just drops duplicate rows.
For a streaming DataFrame, it will keep all data</td><td>Syntax:
<code>DataFrame.dropDuplicates(subset=None)</code>
Example:
<code>df.dropDuplicates(['name', 'height']).show()</code></td></tr>
</table>

	across triggers as an intermediate state to drop duplicate rows. ● You can use withWatermark() to limit how late the duplicate data can be and the system will accordingly limit the state. ● drop_duplicates() is an alias for dropDuplicates() ● Package: ○ pyspark.sql.DataFrame	

dropna	● Returns a new DataFrame omitting rows with null values.
	
	● DataFrame.dropna() and DataFrameNaFunctions.drop() are aliases of each other.
	
	● Package:
	    ○ pyspark.sql.DataFrame
		Syntax:
		
		`DataFrame.dropna(how='any', thresh=None, subset=None)`
		
		● `how`
		    ○ `type: str, optional`
		    ○ `def: 'any' or 'all'. If 'any', drop a row if it contains any nulls. If 'all', drop a row only if all its values are null.`
		
		● `thresh:`
		    ○ `type: int, optional`
		    ○ `def: default None If specified, drop rows that have less than thresh non-null values. This overwrites the how parameter.`
		
		● `subset`
		    ○ `type:str, tuple or list, optional`

		○ `def: optional list of column names to consider.` **Example:** `df4.na.drop().show()`

where/filter	● Filters rows using the given condition.
	● where() is an alias for filter().
	● Package:
	○ pyspark.sql.DataFrame
		Syntax:
		
		DataFrame.filter(condition)
		
		● `Condition:`
		○ `type:Column or str`
		○ `def: a Column of types.BooleanType or a string of SQL expression.`
		
		Example:
		
		`df.filter(df.age > 3).collect()`
		
		`df.where(df.age == 2).collect()`
		
		`df.filter("age > 3").collect()`
		
		`df.where("age = 2").collect()`

distinct	● Returns a new DataFrame containing the distinct rows in this DataFrame.
	● Package:
	○ pyspark.sql.DataFrame
		Syntax:
		
		`DataFrame.distinct()`
		
		Example:
		
		`df.distinct().count()`

sample	● Returns a sampled subset of this DataFrame. ● Package: ○ pyspark.sql.DataFrame	**Syntax:** `DataFrame.sample(withReplacement=None, fraction=None, seed=None)` ● `withReplacement` ○ `type:bool, optional` ○ `def: Sample with`

<table>
<tr>
<td></td>
<td></td>
<td>
replacement or not (default False).

fraction
type:float, optional
def: Fraction of rows to generate, range [0.0, 1.0].
seed
type:int, optional
def: Seed for sampling (default a random seed).

Example:

df.sample(withReplacement=True, fraction=0.5, seed=3).count()

df.sample(False, fraction=1.0)
</td>
</tr>
<tr>
<td>split</td>
<td>

Splits str around matches of the given pattern.
Package:
pyspark.sql.functions

</td>
<td>
Syntax:

split(str, pattern, limit=- 1)

str
type: Column or str
def:a string expression to split
pattern
Type:str
def: a string representing a regular expression. The regex string should be a Java regular expression.
limit
type:int, optional
def: an integer which controls the number of times a pattern is applied.
limit > 0:The resulting array’s length will not be more than limit, and the resulting array’s

</td>
</tr>
</table>

		last entry will contain all input beyond the last matched pattern. ○ limit <= 0: pattern will be applied as many times as possible, and the resulting ○ array can be of any size. **Example:** `df.select(split(df.s, '[ABC]', -1).alias('s')).collect()`

sort/orderBy	● Returns a new DataFrame sorted by the specified column(s).
	● Package:
	○ pyspark.sql.DataFrame
		Syntax:
		`DataFrame.sort(*cols, **kwargs)[source]`
		
		● cols
		○ type: str, list, or Column, optional
		○ def: list of Column or column names to sort by.
		
		● ascending
		○ type:bool or list, optional
		○ def: boolean or list of boolean (default True). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, the length of the list must equal the length of the cols.
		
		Example:
		
		`df.sort(df.age.desc()).collect()`
		
		`df.sort("age", ascending=False).collect()`
		
		`df.sort(asc("age")).collect()`

<table>
<tr><td></td><td></td><td><code>df.orderBy(desc("age"), "name").collect()</code>

<code>df.orderBy(["age", "name"], ascending=[0, 1]).collect()</code></td></tr>
<tr><td>sortwithinpartitions</td><td>Returns a new DataFrame with each partition sorted by the specified column(s).
package:
pyspark.sql.DataFrame</td><td>Syntax:

<code>DataFrame.sortWithinPartitions(*cols, **kwargs)[source]</code>
cols
type: str, list, or Column, optional
def: list of Column or column names to sort by.
ascending
type:bool or list, optional
def: boolean or list of boolean (default True). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, the length of the list must equal the length of the cols.
Example:

<code>df.sortWithinPartitions("age", ascending=False).show()</code></td></tr>
<tr><td>asc_nulls_first</td><td>Returns a sort expression based on the ascending order of the column, and null values return before non-null values.
Package:
pyspark.sql.Column</td><td>Syntax:

<code>Column.asc_nulls_first()</code>

Example:

<code>df.select(df.name).orderBy(df.name.asc_nulls_first()).collect()</code></td></tr>
</table>

desc_nulls_first	- Returns a sort expression based on the descending order of the column, and null values appear before non-null values. - Package: - pyspark.sql.Column	**Syntax:**
		`Column.desc_nulls_first()`
		
		Example:
		
		`df.select(df.name).orderBy(df.name.desc_nulls_first()).collect()`

asc_nulls_last	- Returns a sort expression based on the ascending order of the column, and null values appear after non-null values. - Package: - pyspark.sql.Column	**Syntax:**
		`Column.asc_nulls_last()`
		
		Example:
		
		`df.select(df.name).orderBy(df.name.asc_nulls_last()).collect()`

desc_nulls_last	- Returns a sort expression based on the descending order of the column, and null values appear after non-null values. - Package: - pyspark.sql.Column	**Syntax:**
		
		`Column.desc_nulls_last()`
		
		Example:
		
		`df.select(df.name).orderBy(df.name.desc_nulls_last()).collect()`

limit	- Limits the result count to the number specified. - Package:	**Syntax:** `DataFrame.limit(num)` **Example:**

	○ pyspark.sql.DataFrame	`df.limit(1).collect()`
first		
take		
collect		

show	• Prints the first n rows to the console.
	
	• Package:
	
	○ pyspark.sql.DataFrame
		Syntax:
		
		`DataFrame.show(n=20, truncate=True, vertical=False)[source]`
		
		• `N`
		○ `type: int, optional`
		○ `def:Number of rows to show.`
		
		• `truncate`
		○ `type:bool, optional`
		○ `def:If set to True, truncate strings longer than 20 chars by default. If set to a number greater than one, truncates long strings to length, and aligns cells right.`
		
		• `vertical`
		○ `type:bool, optional`
		○ `def:If set to True, print output rows vertically (one line per column value).`
		
		Example:
		
		`df.show(truncate=3)`
		`df.show(vertical=True)`

from_unixtime	- Converts the number of seconds from the unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the given format. - Package: - pyspark.sql.functions	**Syntax:**
		
		`from_unixtime(timestamp, format='yyyy-MM-dd HH:mm:ss')[source]`
		
		Example:
		
		`time_df = spark.createDataFrame([(1428476400,)], ['unix_time'])`
		`time_df.select(from_unixtime('unix_time').alias('ts')).collect()`

to_date		

datediff	- Returns number of data from start to end - Package: - pyspark.sql.functions	**Syntax:**
		
		`datediff(end, start)`
		
		Example:
		
		`df = spark.createDataFrame([('2015-04-08','2015-05-10')], ['d1', 'd2'])`
		`df.select(datediff(df.d2, df.d1).alias('diff')).collect()`

explode	- Returns a new row for each element in the given array or map. - Uses the default column name for elements in the array and key and value for elements in the map unless specified otherwise. - package:	**Syntax:**
		
		`explode(col)`
		
		Example:
		
		`eDF = spark.createDataFrame([Row(a=1, intlist=[1,2,3], mapfield={"a": "b"})])`
		
		`eDF.select(explode(eDF.intlist).alias("anInt")).collect()`

	○ pyspark.sql.functions	

agg	• Aggregate on the entire DataFrame without groups
	• package:pyspark.sql.DataFrame
		Syntax:
		`DataFrame.agg(*exprs)`
		
		Example:
		`df.groupBy().agg()).`
		`df.agg({"age": "max"}).collect()`
		`df.agg(F.min(df.age)).collect()`

avg	• Aggregate function: returns the average of the values in a group. • Package: ○ pyspark.sql.functions	**Syntax:** `avg(col)` **Example:**

isin	• A boolean expression that is evaluated to be true if the value of this expression is contained by the evaluated values of the arguments.
	• Package:
	○ pyspark.sql.Column
		Syntax:
		`Column.isin(*cols)`
		
		Example:
		`df[df.name.isin("Bob", "Mike")].collect()`
		`df[df.age.isin([1, 2, 3])].collect()`

range	• Create a new RDD of int containing	**Syntax:** `range(start, end=None, step=1,`

	elements from start to end (exclusive), increased by every step.
	
	● Can be called the same way as python's built-in range() function.
	
	● If called with a single argument, the argument is interpreted as end, and start is set to 0.
	
	● Package:
	○ pyspark.SparkContext
		`numSlices=None)`
		
		● `start`
		○ `Type:int`
		○ `def:the start value`
		
		● `end`
		○ `type:int, optional`
		○ `def:the end value (exclusive)`
		
		● `step`
		○ `type:int, optional`
		○ `def:the incremental step (default: 1)`
		
		● `numSlices`
		○ `type:int, optional`
		○ `def:the number of partitions of the new RDD`
		
		Example:
		
		`sc.range(5).collect()`
		`sc.range(2, 4).collect()`
		`sc.range(1, 7, 2).collect()`

Actions	**Wide transformation**	**Narrow transformations**
Count	Repartition	Coalesce
foreach	groupby	Filter
head	Join	map
Collect	distinct	sample
take		union
top		flatMap
reduce		drop
agg		selectexpr
printschema		withcolumnrenamed
show		
take		

AGGREGATE FUNCTIONS:

Function	Use	Example

agg	- Compute aggregates and returns the result as a DataFrame. - The available aggregate functions can be: - built-in aggregation functions, such as avg, max, min, sum, count - group aggregate pandas UDFs, created with pyspark.sql.functions.pandas_udf() - package: pyspark.sql. GroupedData	**Syntax:**
		
		`GroupedData.agg(*exprs`
		
		    `exprs:` `type: dict` `def: a dict mapping from column name (string) to aggregate functions (string), or a list of columns.`
		Example:
		
		`gdf = df.groupBy(df.name)`
		`sorted(gdf.agg({"*": "count"}).collect())`
		
		`sorted(gdf.agg(F.min(df.age)).collect())`

avg/mean	- Computes average values for each numeric column for each group. - mean() is an alias for avg(). - package: pyspark.sql. GroupedData	**Syntax:** `GroupedData.avg(*cols)` `cols` `type: str` `def: column names. Non-numeric columns are ignored.` **Example:**

		`df.groupBy().avg('age').collect()`
		
		`df3.groupBy().avg('age', 'height').collect()`

count	• Counts the number of records for each group
	• package: pyspark.sql. GroupedData.
		Syntax:
		
		`GroupedData.count()`
		
		Example:
		
		`sorted(df.groupBy(df.age).count().collect())`

max	• Computes the max value for each numeric column for each group.
	• package: pyspark.sql. GroupedData
		Syntax:
		
		`GroupedData.max(*cols)[source]`
		
		Example:
		
		`df.groupBy().max('age').collect()`
		
		`df3.groupBy().max('age', 'height').collect()`

min	• Computes the min value for each numeric column for each group.
	• package: pyspark.sql. GroupedData
		Syntax
		
		`GroupedData.min(*cols)`
		
		• cols
		  ○ type:str
		  ○ def: column names. Non-numeric columns are ignored.
		
		Example:
		
		`df.groupBy().min('age').collect()`

pivot	• Pivots a column of the current DataFrame and performs the	**Syntax:** `GroupedData.pivot(pivot_col, values=None)`

	specified aggregation.
	● There are two versions of the pivot function:
	○ one that requires the caller to specify the list of distinct values to pivot on
	○ one that does not.
	
	● package: pyspark.sql.GroupedData
		● `pivot_col`
		○ `type: str`
		○ `def: Name of the column to pivot.`
		
		● `values :`
		○ `def: List of values that will be translated to columns in the output DataFrame.`
		
		Examples:
		
		● `Compute the sum of earnings for each year by course with each course as a separate column`
		
		`df4.groupBy("year").pivot("course", ["dotNET", "Java"]).sum("earnings").collect()`
		
		● `Or without specifying column values (less efficient)`
		
		`df4.groupBy("year").pivot("course").sum("earnings").collect()`

sum	● Compute the sum for each numeric column for each group.
	● package: pyspark.sql.GroupedData
		Syntax:
		
		`GroupedData.sum(*cols)`
		
		● `cols`
		○ `type :str`
		○ `def: column names. Non-numeric columns are ignored.`
		
		Example:
		
		`df.groupBy().sum('age').collect()`
		
		`df3.groupBy().sum('age', 'height').collect()`

JOINS:

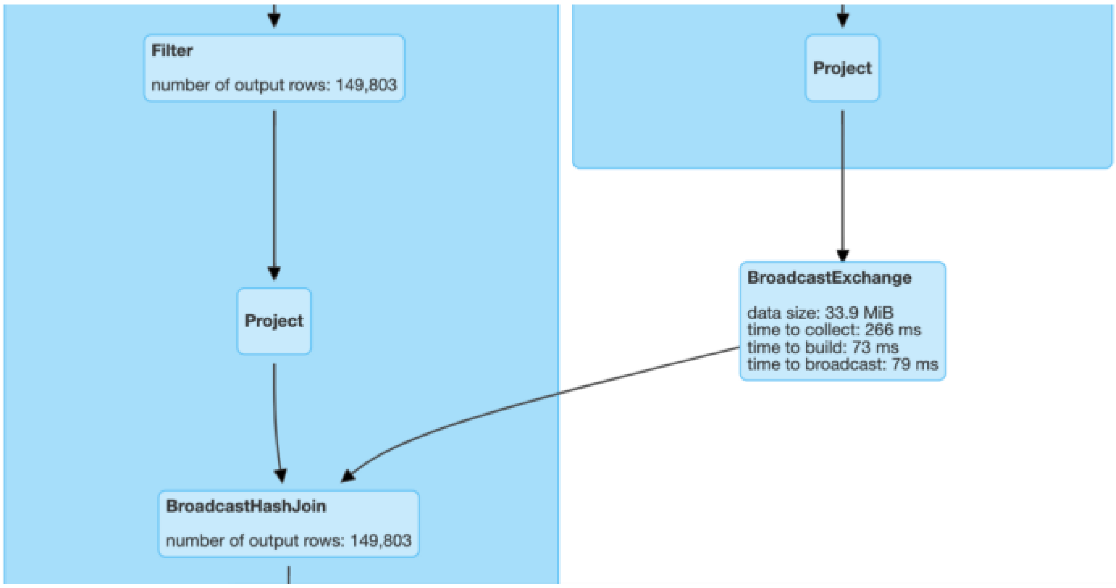
Function	Usage	Example
join	- Joins with another DataFrame, using the given join expression. - package: pyspark.sql.DataFrame.join	**Syntax:** `DataFrame.join(other, on=None, how=None)` - other - type: DataFrame - def: Right side of the join - On - type:str, list or Column, optional - def: a string for the join column name, a list of column names, a join expression (Column), or a list of Columns. If there is a string or a list of strings indicating the name of the join column(s), the column(s) must exist on both sides, and this performs an equi-join.

		• `how`
		  ○ `type: str, optional`
		  ○ `def:`
		    ■ `default inner.`
		    ■ `Must be one of: inner, cross, outer, full, fullouter, full_outer, left, leftouter, left_outer, right, rightouter, right_outer, semi, leftsemi, left_semi, anti, leftanti and left_anti.`

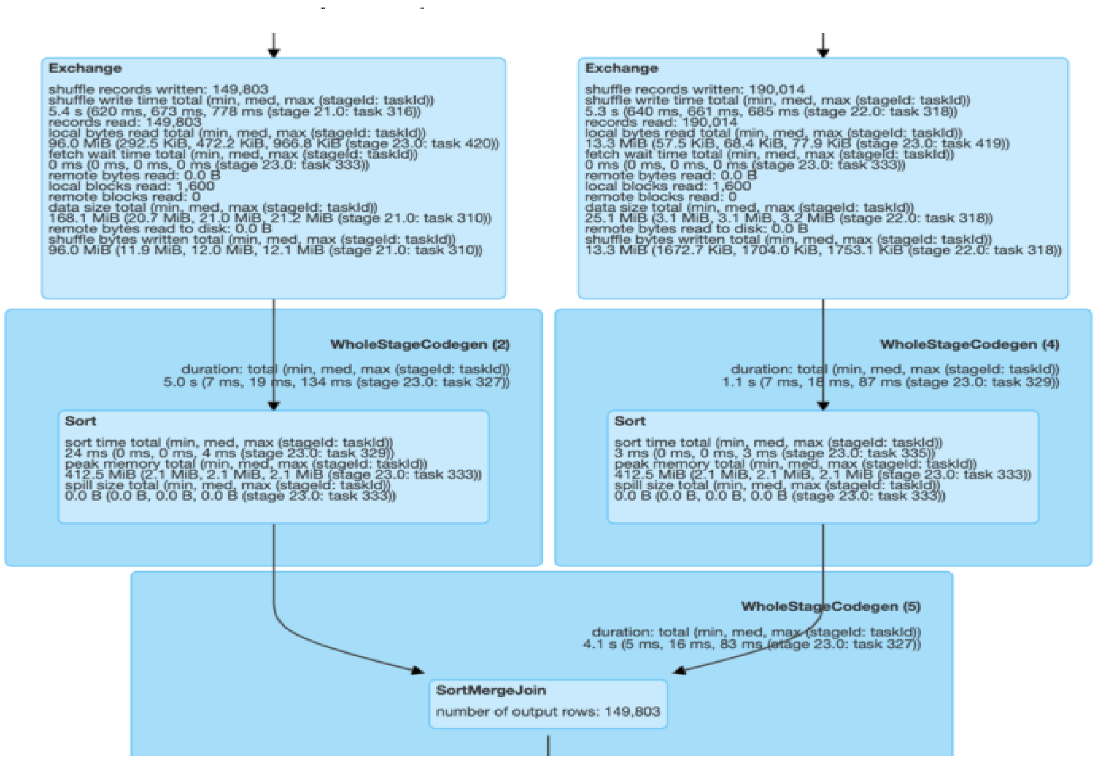
Inner Join	• Spark Inner join is the default join and it's mostly used. • It is used to join two DataFrames/Datasets on key columns, and where keys don't match the rows get dropped from both datasets	`df.join(df2, df.name == df2.name, 'outer') .select(df.name, df2.height)`
Full Outer Join	• fullouter join returns all rows from both Spark DataFrame/Datasets,	

	where join expression doesn't match it returns null on respective record columns.	
Right Outer join	• is the opposite of left join. • here it returns all rows from the right DataFrame/Dataset regardless of match found on the left dataset, when join expression doesn't match, it assigns null for that record and drops records from left where match not found.	
Left Semi Join	• The Spark Left Semi join is similar to the inner join. • The difference being leftsemi join returns all columns from the left DataFrame/Dataset and ignores all columns from the right dataset. • In other words, this join returns columns from the only left dataset for the records matched in the right dataset on join expression; records not matched on join expression are ignored from both left and right datasets. • The same result can be achieved using select on the result of the inner join; however, using this join would be efficient.	

Left Anti Join	• Left Anti join does the exact opposite of the Spark leftsemi join, leftanti join returns only columns from the left DataFrame/Dataset for non-matched records.	
Self Join	• Though there is no self-join type available, we can use any of the above-explained join types to join DataFrame to itself. below example use inner self join	
Join Algorithms in Spark	• The algorithm chosen depends on the size of the tables joined and the hints provided in join • The commonly used joins in spark are: ○ Broadcast Hash join ○ Sort Merge Join ○ Shuffle Hash Join	
Broadcast Hash join	• Used when dataframe size if less than spark.sql.autoBroadcastJoinThreshold. • Does not perform shuffle while doing a join. • Broadcasting huge datasets results in a	

	timeout	
		

Sort Merge Join	• If dataframes cannot be broadcasted, spark used sort merge join.
	• Uses node-to node communication strategy.
	• Two steps:
	  ○ First step exchanges and sorts datasets.
	  ○ Second step merges the dataset.
	• Better to use bucketing for optimization.
	• Default since spark 2.3
		`spark.sql.join.preferSortMergejoin=true`

<table>
<tr><td colspan="3">
</td></tr>
<tr><td>Shuffle Hash join</td><td>

- Works on the concept of map reduce.
- Maps through a dataframe and uses values of join as output key.
- Used when one side is at least 3 times smaller than the other.
- Dataframe size is less than the broadcast threshold.
- Prefer Sort Merge Join is set to false.

</td><td><code>spark.sql.join.preferSortMergejoin=false</code></td></tr>
</table>

PERFORMANCE TUNING CONFIG OPTIONS

Property	Definition	Default
Performance Tuning considerations	• Overhead of garbage collection • Amount of memory used by objects • Cost of accessing these objects	
Improve spark performance	• Increase spark.default.parallelism and spark.sql.shuffle.partitions to improve spark performance	
spark.sql.inMemoryColumnarStorage.compressed	• When set to true, Spark SQL will automatically select a compression codec for each column based on statistics of the data.	true
spark.sql.inMemoryColumnarStorage.batchSize	• Controls the size of batches for columnar caching. • Larger batch sizes can improve memory utilization and compression, but risk OOMs when caching data.	10000
spark.sql.files.maxPartitionBytes	• The maximum number of bytes to pack into a single partition when reading files. • This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.	134217728 (128 MB)
spark.sql.files.openCostInBytes	• The estimated cost to open a file, measured by the number of bytes, could be scanned at the	4194304 (4 MB)

	same time. • This is used when putting multiple files into a partition. • It is better to over-estimated, then the partitions with small files will be faster than partitions with bigger files (which is scheduled first). • This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.	
spark.sql.files.minPartitionNum	• The suggested (not guaranteed) minimum number of split file partitions. • If not set, the default value is `spark.default.parallelism`. • This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.	Default Parallelism
spark.sql.broadcastTimeout	• Timeout in seconds for the broadcast wait time in broadcast joins	300
spark.sql.autoBroadcastJoinThreshold	• Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. • By setting this value to -1 broadcasting can be disabled. • Note that currently statistics are only supported for Hive Metastore tables where the command ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan has been run.	10485760 (10 MB)
spark.sql.shuffle.partitions	• Configures the number of partitions to use when shuffling	200

	data for joins or aggregations.	
spark.sql.sources.parallelPartitionDiscovery.threshold	• Configures the threshold to enable parallel listing for job input paths. • If the number of input paths is larger than this threshold, Spark will list the files by using a Spark distributed job. • Otherwise, it will fallback to sequential listing. • This configuration is only effective when using file-based data sources such as Parquet, ORC and JSON.	32
spark.sql.sources.parallelPartitionDiscovery.parallelism	• Configures the maximum listing parallelism for job input paths. • In case the number of input paths is larger than this value, it will be throttled down to use this value. • Effective when using file-based data sources such as Parquet, ORC and JSON.	10000

TUNING IN SPARK:

Topic	Definition
Need of Tuning	• Since spark is of in-memory nature we can have bottlenecks due to resource, memory or network bandwidth.
Areas of tuning	• Data serialization-crucial for good network performance and reduce memory usage. • Memory tuning
Data serialization	Covered earlier
Memory tuning	• Java objects occupy a lot of memory so we need to mind memory usage and also use appropriate objects

Memory management in spark.	• Spark has two categories of memory usage:
	   ○ Storage
	   ○ Execution
	• Execution refers to memory used for computation in shuffle joins etc.
	• Storage is memory used for caching and propagating internal data structures.
	• When no execution is being done, storage can occupy all the memory.
	• There is a subregion in memory which is always reserved for storage.
	• `spark.memory.fraction` refers to the size of memory M
	• `spark.memory.storageFraction` refers to size of storage fraction as

	a fraction of M
Determining memory usage	• Put rdd in the cache and look at the storage tab in spark UI. • To estimate the memory of a particular object, use sizeEstimator's estimate method.
Measuring impact of GC	• This can be done by adding `-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps` to the Java options
Components of memory in JVM	• Java Heap space is divided into two regions: Young and Old. The Young generation is meant to hold short-lived objects while the Old generation is intended for objects with longer lifetimes. • The Young generation is further divided into three regions [Eden, Survivor1, Survivor2]. • A simplified description of the garbage collection procedure: ○ When Eden is full, a minor GC is run on Eden and objects that are alive from Eden and Survivor1 are copied to Survivor2. ○ The Survivor regions are swapped. If an object is old enough or Survivor2 is full, it is moved to Old. Finally, when Old is close to full, a full GC is invoked.
Goal Of GC	• The goal of GC tuning in Spark is to ensure that only long-lived RDDs are stored in the Old generation and that the Young generation is sufficiently

	sized to store short-lived objects.
Steps to avoid Full GC	- Check if there are too many garbage collections by collecting GC stats. If a full GC is invoked multiple times before a task completes, it means that there isn't enough memory available for executing tasks. - If there are too many minor collections but not many major GCs, allocating more memory for Eden would help. - In the GC stats that are printed, if the OldGen is close to being full, reduce the amount of memory used for caching by lowering spark.memory.fraction;
Data locality	- Data locality is how close data is to the processing it. - This affects speed of execution. - The levels of data locality are: - PROCESS_LOCAL data is in the same JVM as the running code. This is the best locality possible - NODE_LOCAL data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS_LOCAL because the data has to travel between processes - NO_PREF data is accessed equally quickly from anywhere and has no locality preference - RACK_LOCAL data is on the same rack of servers. Data is on a different server on the

	same rack so needs to be sent over the network, typically through a single switch ○ ANY data is elsewhere on the network and not in the same rack

References And Important Links for Spark:

Advanced Apache Spark Training - Sameer Farooqui (Databricks)

- https://www.youtube.com/watch?v=7ooZ4S7Ay6Y

A Deep Dive into Spark SQL's Catalyst Optimizer with Yin Huai

- https://www.youtube.com/watch?v=RmUn5vHlevc&t=671s

Preparation material :

- Spark the definitive guide
- http://spark.apache.org/docs/latest/index.html

Useful Links:

- https://towardsdatascience.com/spark-essentials-how-to-read-and-write-data-with-pyspark-5c45e29227cd?gi=9c79a01f751a
- https://sparkbyexamples.com/pyspark/pyspark-sql-types-datatype-with-examples/

Interesting White papers:

- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
  - http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf
- Spark SQL: Relational Data Processing in Spark
  - http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

- Discretized Streams: Fault-Tolerant Streaming Computation at Scale
  - http://people.csail.mit.edu/matei/papers/2013/sosp_spark_streaming.pdf

- Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters
  - http://people.csail.mit.edu/matei/papers/2012/hotcloud_spark_streaming.pdf
- Shark: Fast Data Analysis Using Coarse-grained Distributed Memory
  - http://people.csail.mit.edu/matei/papers/2012/sigmod_shark_demo.pdf
- Spark: Cluster Computing with Working Sets
  - http://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf