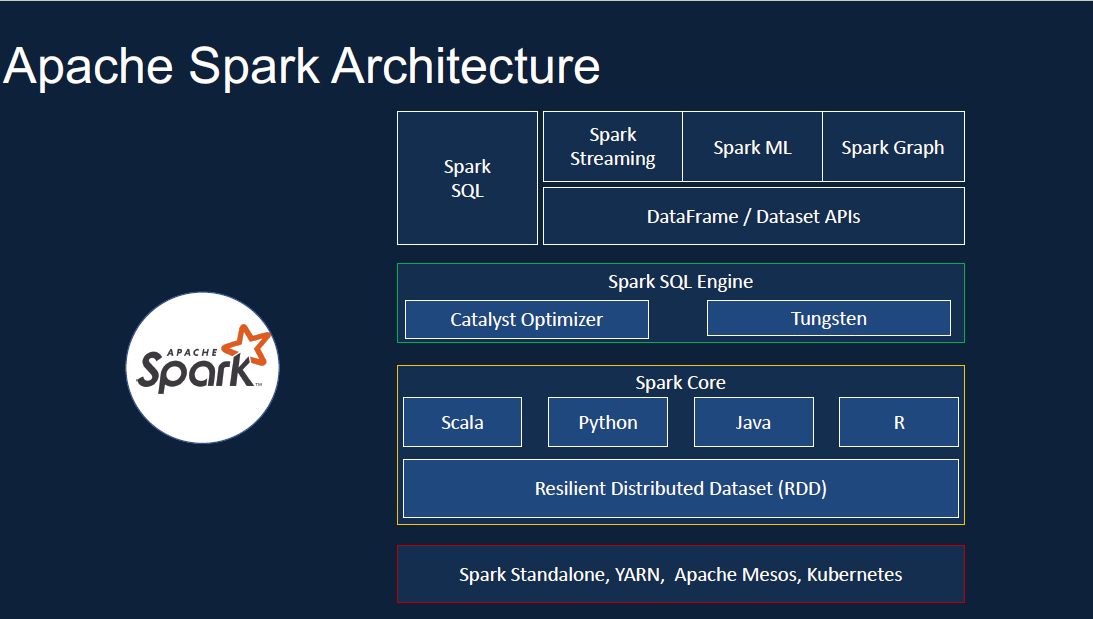
ADB is an open-source distributed compute processing engine called Apache Spark which is widely used in industry for developing big data projects. Data bricks is a company created by a founder of Apache Spark to make it easier to work with Spark by providing the necessary management layers.

Apache Spark is a unified analytics engine for big data processing and machine learning.

Hadoop was slow and inefficient for interactive and iterative computing jobs, and it was too complex to learn and develop. On the other hand, spark offers a much simpler, faster, and easier APIs to develop on. Spark can be 100X faster than Hadoop for large scale data processing by exploiting in memory computing and other optimizations. Spark runs on a distributed computing platform. Spark has a unified engine to support varying workloads. For e.g., it uses single engine for streaming and batch workloads. It doesn’t have separate one for each of those. It comes packaged with high level libraries including support for SQL queries, streaming data, Machine Learning and Graph processing. These standard libraries increase developer productivity and be seamlessly combined to create complex workflows.



At center of spark architecture is Spark Core. This contains basic functionality of spark. Spark Core takes care of scheduling tasks, memory management, fault recovery, communication with storage systems. It’s also home to spark main programming Abstraction API called RDD. RDD’s are a collection of items distributed across various compute nodes, in the cluster that can be processed in parallel. Spark Core provides the APIs to create and manipulate these RDD collections.

Early development in Spark was done using these APIs, but it had its drawbacks.

It was difficult to use for complex operations and it was difficult to optimize for Spark and mainly

down to the developer to write the optimized code.

To optimize the workload,

Spark introduced the SQL engine. It includes the Catalyst Optimizer, which takes care of converting a computational query to a highly efficient execution plan

and the Tungsten Project, which is responsible for memory management and CPU efficiency.

The higher-level abstraction such as Spark SQL and the Dataset and the Data Frame APIs, make it easier

to develop applications and benefit from the optimizations from the SQL engine.

So, the recommended approach to develop applications in Spark, is to use these higher-level APIs rather

than the RDD API.

The Dataset and the Data Frame APIs can be invoked from any of the domain specific languages such as

Scala, Python, Java, or R. On top of this, we have the set of libraries such as Spark Structured

Streaming for streaming, ML Live for machine learning and Graphics for graph processing.

Also, Spark comes with its standalone resource manager, but you can choose other resource managers

such as YARN, Apache Mesos and Kubernetes.

Combining all of these, **Spark provides the unified platform for doing streaming, batch, machine learning**

**and graph processing workloads using a single execution engine and a standard set of APIs.**

Now we know Spark is a fast execution engine with an easy-to-use set of higher-level APIs.

But, to work with Spark, we must set up our own clusters, manage security, and use

third party products to write our programs.

That's where Databricks comes in.

Databricks is a company founded by the creators of Apache Spark to make it easier to work with Spark

on the Cloud.

For Spark to do its distributed computing, we need to spin up Clusters and install the software.

Databricks gives you the ability to spin up the Clusters with a few clicks.

You can choose the runtime, which is suitable for your needs.

For example, you can choose a runtime with ML libraries, support for GPU, etc. Also, you can choose

from a wide range of Clusters ranging from general purpose, memory optimized, compute optimized,

or GPU enabled.

**It provides a Jupiter Notebook style IDE with additional capabilities to create your application. Collaborate with your other colleagues and integrate with configuration management tools such as Git.**

It provides administration controls that you can use to restrict or provide access to your users, to

the workspace, Clusters, etc.

On top of this, Databricks provides the Spark runtime, which is highly optimized for the Databricks

platform and known to be up to 5x faster than the Vannila Apache Spark.

With the use of high meta store, Databricks also provides the ability to create databases and tables.

To provide ACID transaction capability,

Databricks also comes with the Open-Source project Delta Lake, and a recent addition to Databricks

is the SQL Analytics, which provides the data analyst a SQL based analytics environment.

This allows the analyst to explore data, create dashboards, schedule a regular refresh of the dashboard,

etc.

Also, it comes with managed ML flow on Databricks, which allows us to manage the machine learning

lifecycle, including experimentation, deployment, model registry, etc...

With the recent announcement from Google in February 2021, the Databricks Cloud platform is now available

on all three major Cloud platforms, such as Microsoft Azure, AWS, and Google Cloud.

But Azure's integration is deeper than others, Databricks is a first party service on Azure.

What that means is, on Azure you will be buying Databricks directly from Microsoft and all support requests

are handled by Microsoft.

As a result, it provides a unified Azure Portal for Databricks and a single unified bill for all your Azure

services, including Databricks.

Azure Databricks leverages, Azure security and seamlessly integrates with Azure Active Directory

and single sign on.

It provides seamless integration and high-speed connectors between various Azure data services such

as Azure Data Lake, Blob Storage, Cosmos DB, SQL DB and Synapse.

Messaging services such as Event Hub and IoT Hub, Power BI and Azure ML, you can seamlessly run

Databricks notebooks from Azure Data Factory and integrate with the rest of the data workflow in your

data project.

And finally, Databricks also connects with Azure Dev Ops to enable continuous integration and continuous

deployment.

So just to summarize, Azure Databricks is a spark based unified data analytics, platform as

a service offering, that's optimized for the Microsoft Azure Cloud.

Azure Data bricks: All resources🡪 search with ADB 🡪 create 🡪 we will enter ADB workspace🡪 select tier (premium/standard tier), premium tier offers extra features along with standard features.

Premium tier offers role-based access controls, audit logs, unity catalog, Azure AD etc., When ADB workspace is created by default storage account is created in ADB, if we want to enable infrastructure storage we can enable while creating ADB.

And we can pin created ADB workspace to dashboard.

Go to menu🡪My Dashboard 🡪 open dashboard which we created 🡪 open ADB workspace.

Some of the products such as Databricks SQL, Delta Live Tables are available on premium tier.

New menu in ADB used to create notebooks, clusters, jobs

Workspace menu: is basically a container for holding a set of folders, libraries, and files. By default, each user has its own workspace and there is shared workspace which is used to share assets among other users in this data bricks workspace.

Workflows menu contains Jobs, Job runs, Delta Live Tables. Data bricks jobs let you schedule notebooks periodically via scheduling system. We can create in jobs and monitor in job runs tab.

Delta Live Tables: This is an ETL Framework to build data pipelines with automated testing.

Compute menu used to create clusters, cluster pools and SQL Data warehouses.

When we create ADB workspace, by default data bricks itself will create Storage Account, Network Security Group, Virtual Network. Here Storage account (Azure blob storage) is the default storage /DBFS and its not recommended as a permanent data storage.

When user requesting cluster to be created, Databricks cluster manager will create required VM’s in our subscription via ARM.

**A Cluster is basically a collection of Virtual Machines**. In a Cluster, there is usually a Driver node,

which orchestrates the tasks performed by one or more worker nodes.

Clusters allow us to treat this group of computers, as a single compute engine via the Driver node.

Databricks Clusters enable us to run different types of workloads, such as ETL for Data Engineering,

Data Science and Machine Learning workloads.

**Databricks offers two types of Clusters.**

Let's look at them.

The first one is the **All-purpose Cluster, which is created manually**, via the Graphical User Interface,

the CLI or the API. Whereas, the **Job Clusters are created when job runs and destroyed/terminates as soon as it completes** , and the job has been configured to use a Job Cluster.

**All Purpose Clusters are persistent,**

**they can be terminated and restarted at any point in time**, whereas the **Job Clusters are terminated**

**at the end of the job.** They cannot be restarted. So, they're no longer usable once the job has been completed.

All Purpose Clusters are suitable for interactive and ad-hoc Analysis workloads.

On the other hand, Job Clusters are suitable for automated workloads, such as running an ETL pipeline

or Machine Learning workflow at a regular interval.

**All Purpose Clusters can be shared among many users, and they are good for collaborative analysis,**

whereas the Job Clusters are isolated just for the job being executed.

All Purpose Clusters are expensive to run compared to the Job Clusters. Job Clusters are cheaper.

In summary, **All Purpose Clusters are great for interactive analysis and ad-hoc work**, **whereas Job Clusters are great for repeated production workloads.**

**Cluster Pools:** When we create an cluster, it takes about 5-6 mins to spin up a cluster. In order to speed up that time, we can have a pool of resources waiting for you via cluster pools.

**Cluster Configuration:**

* **Multi Node vs Single Node:**

First, we have the option to choose whether we want to create a Single Node or a Multi Node Cluster.

Multi Node Cluster will have one Driver Node and one or more Worker Nodes.

When you run a Spark Job against a Multi Node Cluster, the Driver Node will distribute the tasks to

run on the Worker Nodes in parallel and returns the result.

They give us the ability to horizontally scale the Cluster depending on your workload.

We can basically keep adding Worker Nodes as we need.

These are the default type of Clusters used for Spark Jobs and suitable for large workloads.

On the other hand, Single Node Cluster will have only one node, which is the Driver Node and

there are no Worker Nodes.

Even though, there are no Worker Nodes, Single Node Clusters also supports Spark workloads.

When you run a Spark Job, the Driver Node acts as both the driver and the worker. As there are no Worker

Nodes,

the Single Node Clusters are not horizontally scalable, so they're not suitable for large ETL workloads.

They're mainly targeted for lightweight Machine Learning and Data Analysis workloads which don't require, any distributed compute capacity.

* **Access mode :**

There are four different types of Access Modes available at the moment for the Cluster.

As the name suggests, Single User access mode only allows a single user to access the Cluster.

It supports all four languages Python, SQL, Scala, and R. Shared access mode allows the Cluster

to be shared amongst more than one user, but it provides process isolation.

Each process gets its environment, so one process can't see the data or the credential used by the

other one.

It's only available on premium workspaces.

Also, it only supports Python and SQL workloads.

No Isolation Shared also allows the Cluster to be shared amongst more than one user.

It's available on both standard and premium workspaces.

Also, it supports all four languages Python, Scala, SQL and R.

The main difference between this and the Shard access mode is that, No Isolation Shared access mode doesn't

provide any process isolation.

So failure in one user's process may affect the others.

Also, they don't offer any task preemption, so one running process may use all the resources and the

others may fail.

And most importantly, as everything is shared, it's considered less secure.

Custom access mode is not an option,

* **DataBricks Run time:**

 Databricks offers four types of

runtimes.

Databricks Runtime, Databricks Runtime ML, Photon Runtime and Databricks Runtime Light.

Databricks Runtime includes an optimized version of Apache Spark Library. Java, Scala, Python and

R Libraries, Ubuntu, and its accompanying system Libraries,

GPU Libraries for GPU enabled Clusters, Delta Lake Libraries, and other Libraries for Databricks services

that integrate with other components of the platform such as Notebooks, Jobs, and Cluster Manager.

Databricks Runtime ML includes all the libraries from the Databricks Runtime, plus the popular ML Libraries

such as PyTorch, Keras, TensorFlow, XGBoost, etc..

Photon Runtime also includes all the libraries from the Databricks runtime, plus the Photon Engine,

which is the Databricks native vectorized query engine, that runs SQL workloads faster and reduces your

cost per workload.

Databricks Runtime Light is the runtime option for only jobs not requiring advanced features such as

auto scaling, reliability, and improved performance.

Also, it's only suitable for Automated Workloads.

You can't use it for Interactive Workloads or Notebook Jobs.

**Auto Termination:**

Auto Termination is a nice feature that will avoid unnecessary costs on idle Clusters.

It's especially useful on Ad-hoc clusters for preventing them, running during evenings and weekends when

they're not in use.

You can specify when to terminate your Databricks Cluster,

if the cluster has not been in use. It will be terminated after the number of minutes specified.

Default value for Auto Termination is 120 minutes, but you can change the value.

The accepted values range from 10 to 10000 minutes.

**Auto Scaling:**

When you create a Multi Node Cluster, you can specify the minimum and the maximum number of Worker

Nodes.

Auto Scaling will automatically add or remove nodes from the Cluster depending on your workload.

This can result in optimum utilization of the Cluster.

This is especially useful if you're unsure about the workload upfront or your workload changes throughout

the process. They're not recommended for streaming workloads, even if specified Databricks defaults to the maximum number of worker nodes.

**Cluster VM Type/Size:**

There is a wide array of Azure VM types available for us to use.

Databricks groups them into small number of easy-to-understand groups.

Memory Optimized instance types are recommended for memory intensive applications.

For example, a Machine Learning workload that caches a lot of data in memory.

Compute Optimized instance types can be useful for structured streaming applications, where you need

to make sure that the processing rate is above the input rate at peak times of the day.

These can also be used for Distributed Analytics and Data Science Applications.

Storage Optimized instance types are recommended for use cases requiring high disk throughput and I/O.

General Purpose instance types are recommended for Enterprise Grade applications and analytics with

In-memory caching.

GPU Accelerated instance types are recommended for Deep Learning Models, that are data and compute intensive.

**Cluster Policy:**

The final configuration option is Cluster Policy.

As you have seen, there are a lot of options to choose from when you are configuring a Cluster.

This could easily overwhelm a Data Engineer or a Machine Learning Engineer, and creating Clusters

become the sole responsibility of the administrator. Because it's too difficult to configure for a Standard

Data Engineer or a Machine Learning Engineer.

Also, without careful consideration, users could accidentally create Clusters which are oversized

and too expensive to run.

Cluster Policies help us avoid these common issues.

Administrators can create Cluster policies with restrictions and assign them to users or groups.

In this example, we haven't selected a Cluster Policy and it's left as unrestricted.

As you can see, when a Cluster Policy is selected, the configuration becomes much more simplified.

In this example, a Personal Compute cluster policy has taken up the option of Multi Node, and the user

can only create a Single Node Cluster.

Also, it defaulted the runtime version to ML Runtime, limited the node types and Auto Termination

set to 20 minutes.

We can assign this to ML Engineers, and they'll be able to create only these kinds of small Clusters.

In summary, Cluster policies simplify the user interface, thus enabling standard users to create Clusters

and take away the need for administrators to be involved in every decision.

And most importantly, it achieves cost control by limiting the maximum size of the Clusters.

But please note that this is only available on premium tier.

Spot instances: we can request to use unused azure capacity via spot instances by checking tic box here. This will save your cost of running your application.

Photon Acceleration: Cluster with photon engines are normally more expensive but for larger workloads, we are likely to save cost as the query is finished quicker and can terminate the cluster soon.

Azure Data bricks pricing calculation:

**How to know cluster pricing?**

Cluster pricing depends on Workloads (All-purpose/Job), Tier (Premium/Standard) , VM Type, purchase plan(prepurchase/pay as you go)

Notebook: is a collection of cells that we can run commands on a data brick cluster.

We can move through languages from python to SQL to Scala through magic commands

%sql

Select “Hello”

%python

Message = ‘Welcome to the ADB’

%md is a mark down language used to document our notebook which make it usable for everyone.

%md

# Notebook Introduction (for header)

## Magic Commands (for subheader)

## UI Introduction ((for subheader)

* %sql (for bullet points)
* %python (for bullet points)
* %scala (for bullet points)

%fs : If we want a list of files in default storage , we need to run command ls here and that list all files for us.

%fs ls 🡪 This command basically listed folders with in the data bricks root folder.

%sh : shell command If we want to see all the processes that are running in the cluster, we will use this shell command.

%sh ps

**Databrick utilities:**

Databricks Utilities make it easier to combine **different types of tasks in a Single notebook**. For example, they allow us to **combine file operations with ETL tasks**. These utilities can only be run from Python, Scala, or R cells in a Notebook. They cannot be run from a SQL cell.

**File System Utilities:**

It allows us to access data bricks file system from a notebook and you can use various file system level

operations. File System Utilities **to mount containers from Azure Data Lake Storage into Databricks.**

**Secrets Utilities:**

Secrets Utilities allow us to get secret values from secrets, which are stored in secret scopes backed by Databricks or Azure Key Vault.

**Widget Utilities:**

Widget Utilities allows us to parameterized notebooks so that a calling notebook or another application, for example, a Data Factory Pipeline can pass a parameter value to the notebook at runtime.

This is really useful to make a notebook reusable.

**Notebook Workflow Utilities:**

Notebook Workflow Utilities allow us to invoke one notebook from another and chain them together.

dbutils.fs dbutils.fs package offers a number of methods to perform file system operations and ls is one

of them.

dbutils.fs.ls(‘/’) this command will useful to access root folder , if we keep / it will list all folders and files with in the root folder

dbutils.fs.ls(‘/databricks-datasets’) this command used to access files within the folder, if we run this command we can see files and sub folders and folders .

dbutils.fs.ls(‘/databricks-datasets/COVID’) if we want to see covid information we can go to covid folder and we will get no of files used in our projects.

%fs ls

why we would want to use the dbutils package, instead of the Magic Command %fs. The answer is dbutils package provides a greater flexibility, as it can be combined with other native languages like Python, Scala or R.

we can use for loop to iterate through the list and get list of all files as shown below code.

for files in dbutils.fs.ls(‘/databricks-datasets/COVID’) :

print(files)

--------------------------

for files in dbutils.fs.ls(‘/databricks-datasets/COVID’) :

if files.name.endswith(‘/’):

print(files)

When to use magic command and db utils ? We can use %fs magic command for doing some ad hoc queries and dbutils.fs package if we are doing programmatically like above for if commands .

dbutils.help() describes various utilities present in dbutils

dbutils.fs.help() provides different methods in fs

dbutils.fs.help(‘ls’) if we want to get help on one of the method ls we will get those details.

In order to access the data in a data lake , we need to set a spark configuration

Spark.config.set(“fs.azure.account.key.<storage account name>.dfs.core.windows.net”, “<accesskey>”)

To access data stored in a storage account, we use below command abfs(azure blob file system) driver

dbutils.fs.ls(“abfss://containername@storageaccountname.dfs.core.windows.net/”)

How to access Access Azure data lake using access keys from ADB:

1 Set the spark config fs.azure.account.key

spark.conf.set( "fs.azure.account.key.formula1dl.dfs.core.windows.net", "access key")

Here access key will get from Storage account name 🡪 access keys 🡪 key 1/key 2 anything we can select and paste at top conf command

2 List files from containername

display(dbutils.fs.ls("abfss://demo@formula1dl.dfs.core.windows.net"))Read data from csv file

3 Read data from circuits.csv file

display(spark.read.csv("abfss://demo@formula1dl.dfs.core.windows.net/circuits.csv"))

Shared Access Signature/SAS Token:

Control access more at a granular level. Restrict access to specific resource types/services

**Allow specific permissions, restrict access to specific time period.**

**We can also limit access to specific IP address avoiding public access.**

**Service Principals:**  are recommended method to be used in automated tools such as data bricks jobs as well as CI/CD pipelines. This is because they provide better security and traceability.

Cluster scoped authentication: Authentication provided only for cluster (so all notebooks inside cluster is accessible), when cluster is detached, again authentication needs to be provided. So go to created cluster open edit ->go to advanced and there we will add token and fs.azure.account.com command.

Session Scoped Authentication: Authentication obtained only w.r.t to session in execution of notebook.

Pass through Authentication: Authentication done by Storage account🡪 Access Control (IAM) 🡪 Add role assignment🡪 Storage BLOB data contributor access 🡪 to member and go to cluster 🡪 edit 🡪 enable pass through credential

Without spark config directly, we can access data lake account

**Azure Key Vault:** used to securely access secrets. It reduces chances secrets can be accidentally leaked.

We will go to Azure Key Vault 🡪 Create Key Vault 🡪 create secret 🡪 there we will specify name and key (key is SAS, access key, service principal client tenant and secret id’s etc.)

We can also use secret utilities in cluster🡪Advanced 🡪 and add that command.

**Databricks mount benefits:**

We can access data without requiring credentials

Access files using file semantics rather than storage URL’s (eg: /mnt/storage1)

Stores files to object Storage (Azure blob), so u get all benefits from azure

**How to mount ADLS Gen2 to DBFS?**

In order to create a mount, first we need to create **service principal**. A Service Principal is nothing but an Azure Active Directory credential, similar to your user account. You can consider this as a service account.

Once we have created the Service Principal, we need to grant access for the Data Lake storage to the

Service Principal. We can then create the mount points in DBFS using these credentials.

The mount points we create provide access to the storage without requiring credentials.

Also, we will be able to use the Data Lake using file system semantics such as /mnt/storage1

1 Get client\_id, tenant\_id and client\_secret from key vault

client\_id = dbutils.secrets.get(scope = 'formula1-scope', key = 'formula1-app-client-id')

tenant\_id = dbutils.secrets.get(scope = 'formula1-scope', key = 'formula1-app-tenant-id')

client\_secret = dbutils.secrets.get(scope = 'formula1-scope', key = 'formula1-app-client-secret')

2 Set Spark Config with App/ Client Id, Directory/ Tenant Id & Secret

configs = {"fs.azure.account.auth.type": "OAuth",

"fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",

"fs.azure.account.oauth2.client.id": client\_id,

"fs.azure.account.oauth2.client.secret": client\_secret,

"fs.azure.account.oauth2.client.endpoint": f"https://login.microsoftonline.com/{tenant\_id}/oauth2/token"}

3 Call file system utlity mount to mount the storage

dbutils.fs.mount(

source = "abfss://demo@formula1dl.dfs.core.windows.net/",

mount\_point = "/mnt/formula1dl/demo",

extra\_configs = configs)

display(dbutils.fs.ls(“/mnt/formula1dl/demo”))

display(spark.read.csv(“/mnt/formula1dl/demo/circuits.csv”))

# Unmount the mount point if it already exists

if any(mount.mountPoint == f"/mnt/{storage\_account\_name}/{container\_name}" for mount in dbutils.fs.mounts()):

dbutils.fs.unmount(f"/mnt/{storage\_account\_name}/{container\_name}")

# Mount the storage account container

dbutils.fs.mount(

source = f"abfss://{container\_name}@{storage\_account\_name}.dfs.core.windows.net/",

mount\_point = f"/mnt/{storage\_account\_name}/{container\_name}",

extra\_configs = configs)

display(dbutils.fs.mounts())

Data frame API’s Commands:

dbutils.fs.mounts()

display(dbutils.fs.mounts())

dbutils.fs.ls(‘/mnt/formula1dl/raw’)

circuits\_df =spark.read.csv(“dbfs:/mnt/formula1dl/raw/circuits.csv”)

type(circuits\_df) it will show type data frame

circuits\_df.show() ---to see contents in csv file it will display first 20 rows by default here data can be truncated to 20 characters so can use display command to see full data.

display(circuits\_df)

circuits\_df =spark.read.option(“header”, True). csv(“dbfs:/mnt/formula1dl/raw/circuits.csv”)

circuits\_df.printSchema() if we want to shows col names schemas and whether it is nullable or not we will use printSchema method

circuits\_df.describe().show() this will show min max mean count stddev of all cols .

circuits\_df =spark.read.option(“header”, True). option(“inferSchema”,True).csv(“dbfs:/mnt/formula1dl/raw/circuits.csv”)

or

circuits\_df =spark.read \

.option(“header”, True) \

. option(“inferSchema”,True) \

.csv(“dbfs:/mnt/formula1dl/raw/circuits.csv”) inferSchema will internally identify column types.

Struct Type represents row where as StructField represents individual fields/cols. List of struct fields is struct type.

### Selecting only required columns

circuits\_selected\_df = circuits\_df.select(“circuitId”,”circuitRef”, “name” ,”location”,”country”,”lat”,”lng”,”alt”) using this select we cant perform any operations, if we want to perform operations then ,

from pyspark.sql.functions import col

circuits\_selected\_df = circuits\_df.select(col(“circuitId”),col(”circuitRef”), col(“name” ),col(”location”),col(”country”).alias(“race\_country”),col(”lat”),col(”lng”),col(”alt”))

display(circuits\_selected\_df)

rename columns as required

circuits\_renamed\_df = circuits\_selected\_df.**withColumnRenamed**(“circuitId”,”circuit\_id”) \

.withColumnRenamed(“circuitRef”,”circuit\_ref”) \

.withColumnRenamed(“lat”,”latitude”) \

.withColumnRenamed(“long”,”longitude”) \

.withColumnRenamed(“alt”,”altitude”)

display(circuits\_renamed\_df) new renamed col is circuit\_id, circuit\_ref,latitude,longitude,altitude

Adding ingestion date to data frame

circuits\_final\_df = circuits\_renamed\_df.withColumn(“date column”,current\_timestamp())

from pyspark.sql.funtions import current\_timestamp

circuits\_final\_df = circuits\_renamed\_df.withColumn(“ingestion\_date”,current\_timestamp())

display(circuits\_final\_df)

from pyspark.sql.funtions import current\_timestamp,lit

circuits\_final\_df = circuits\_renamed\_df.withColumn(“ingestion\_date”,current\_timestamp()) \

.withColumn(“env”, lit(“Production”))

display(circuits\_final\_df) -- this will create new column env with literal value as production for all rows.

Write data to data lake as parquet

circuits\_final\_df.write.parquet(“/mnt/formula1dl/Processed/circuits”) in this folder we are creating file

df=spark.read.parquet(“/mnt/formula1dl/Processed/circuits”)

display(df)

If we run again circuits\_final\_df.write.parquet(“/mnt/formula1dl/Processed/circuits”) this will throw error as file is already there, so we will write below command

circuits\_final\_df.write.mode(“overwrite”).parquet(“/mnt/formula1dl/Processed/circuits”) when it is mounted no need to use dbfs:/mnt/….. command before mnt

display(spark.read.parquet(“/mnt/formula1dl/Processed/circuits”))

using %run command we can call one notebook (child notebook) in to another notebook

%run “..folder\_name/notebook\_name”

Passing Parameters to Notebook: we can reuse notebook. For e.g.: if u process data from 2 different sources with same characteristics, instead of writing 2 different notebooks, u can write 1 notebook and send the data source name as a parameter and store that against data.

Dbutils.widgets.help()

dbutils.widgets.text(“p\_datasource”,””)

v\_datasource = dbutils.widgets.get(“p\_datasource”)

dbutils.notebook.help()

v\_result = dbutils.notebook.run(“notebook\_name”, 0,{“p\_datasource”: “Ergast API”})

dbutils.notebook.exit(“success”)

We can run multiple notebooks concurrently using threads

Databricks jobs: we can schedule a notebook to run at specified time at regular intervals.

Unity Catalog: used for implementing data governance in data lake house. Data governance means managing data security, availability, usability, and integrity.

Unity catalog is useful for

data access control --- Providing data access to users

Data Audit : Once user is given the access , we should be able to see how user is using the data and how often and when its being accessed, so audit information of data access to be logged and made available.

Data Lineage: If we face any issues in pipelines, we can trace back the root cause of any potential issues with data.

ADF is fully managed serverless data integration, data transformation and data orchestration service. ADF is loading and transforming data periodically and process data in to ASA, ADLS, cosmos db etc . it will not store any data.

ADF Components:

Linked Service: to get connection from one of ip sources.

DataSet: used to get structure of file, name of file

Activity: If we wan tot execute ADB notebooks, using activity we can do.

Copy activity: which copies data from one form of storage to another form.

Pipeline: in order to execute activity. A pipeline may have 1/more activities. You can change them. You can create dependencies between them .

**Managed vs external table in azure:**

When you drop managed table,you will drop data as well,

Data as well as table dropped in managed tables.

In external tables , data remains same, only meta data will get dropped.

How to identify duplicate values in python:

df =spark.read.csv(“/mnt/SN/CN/filename.csv”)

duplicate\_count =df.groupBy(df.col\_name).count()

duplicate\_value = duplicate\_count.filter(col(“count”)>1)

duplicate\_value.show()

%run “..fn/notebook\_name”

df = spark.read.parquet(f”{processed\_folder\_path}/races”)

df\_filtered\_year = df.filter(“col\_name=2019”)

df\_filtered\_year = df.filter([df.col\_name] ==2019)

race\_filter\_yr = race\_results\_df.filter(“race\_year==2020”)

Full Load vs Incremental Load:

When we have small amounts of data, call as Full Load. It won’t be suitable for large data pipelines mainly due to amount of processing will have to perform in order to process all of the data every time.

Incremental load: load and process the data that has changed between current run and previous run.

Eg: If we receive all of the data every time, it is a good candiadate for full load.

But if we instead receive only the data that’s changed since last load , then that is a good candiadate for incremental load.

Simply Full load is load and process the data that hasn’t changed between current run and previous run.

Incremental Load is load and process the data that has changed between current run and previous run.

HOW DBFS FILE SYSTEM IS ENABLED:

Go to Settings (mail id) at RHS 🡪 Go to Admin Console 🡪 Go to workspace settings 🡪 enable DBFS file system .

Now go to Data 🡪 DBFS tab 🡪 double click FileStore🡪 right click and create folder Data and upload files of local files.