

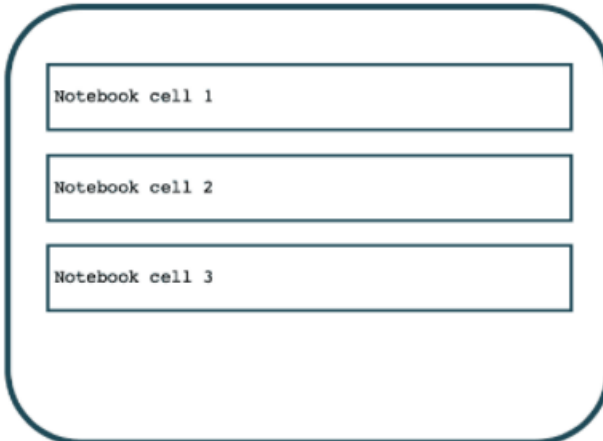
DATABRICKS WORKSPACE USER INTERFACE

Databricks Workspace Terminology

- ✚ The Databricks “Workspace” is the “Collaborative Online Environment” where Data Practitioners run their Data Engineering, Data Science, and, Data Analytics Workloads.
- ✚ In the “Workspace”, Data Practitioners work with “Assets” and “Folders”.
- ✚ **Workspace Assets** - “Workspace Assets” include “Notebooks”, “Clusters”, “Jobs”, “Libraries”, “Data” and “Experiments”.

- **Notebooks** – Most of the work done in the “Workspace” is done via Databricks “Notebooks”. Each “Notebook” is a “Web-Based Interface” composed of a “Group of Cells” that allow to execute coding commands. Databricks “Notebooks” are unique in that these can run code in a variety of Programming Languages, including - “Scala”, “Python”, “R” and “SQL”, and, can also contain “Markdown”.

Databricks “Notebooks” can be easily shared and imported among Databricks Users, using a variety of Formats. The image below shows a conceptual version of a Databricks “Notebook”, containing three cells -



Databricks support several “Notebook External Format”, all of which can be “Imported” and “Exported” using the Databricks “Workspace” -

- ✓ **Source File** - A “File” containing only “Source Code Statements” with the extensions “.scala”, “.py”, “.sql”, or, “.r”.
- ✓ **HTML** - A Databricks “Notebook” with the extension “.html”.
- ✓ **DBC Archive** - A Databricks “Archive” that can contain a “Folder of Notebooks”, or a single “Notebook”. A Databricks “Archive” is a “JAR File” with extra “Metadata” and has the extension “.dbc”. The “Notebooks” contained in the “Archive” are in a “Databricks Internal Format”.

✓ **IPython Notebook** - A “Jupyter Notebook” with the extension “ipynb”.

✓ **RMarkdown** - An “R Markdown Document” with the extension “.Rmd”.

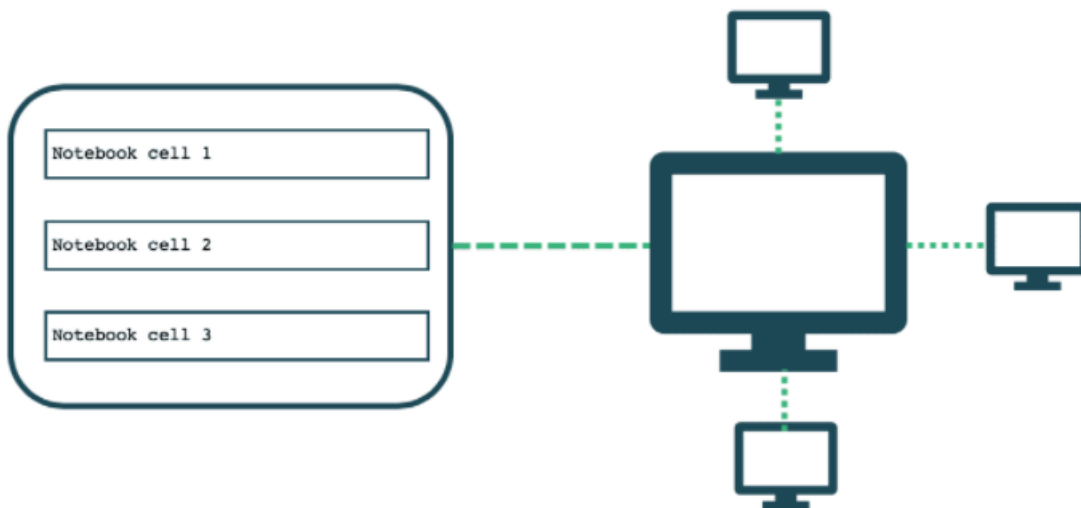
While “Exporting” a “Notebook” as “HTML”, “IPython”, or, “DBC Archive” without clearing “Notebook Results”, the “Results” of that “Notebook” will be included in the “Exported File”.

- **Clusters** - “Clusters” are “Sets of Computational Resources” that are configured in different ways to run an organization’s Engineering, Data Science and Data Analytics Workloads. Each Databricks “Notebook” must be connected to a “Cluster” in order to run.

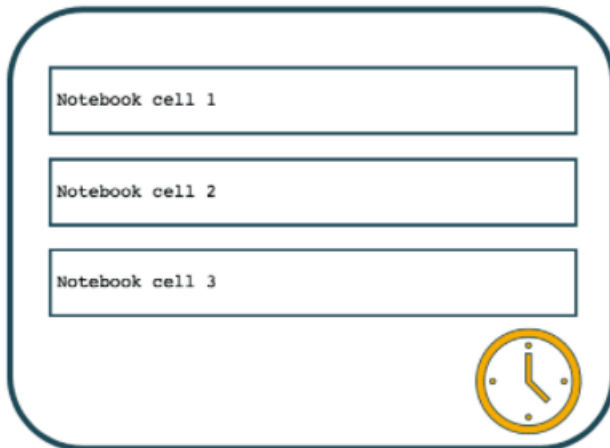
Databricks “Clusters” include “All-Purpose Clusters” and “Job Clusters”.

✓ **All-Purpose Clusters** - “All-Purpose Clusters” are created and started manually, and, used to analyze data collaboratively using “Notebooks”.

✓ **Job Clusters** - “Job Clusters” are initiated natively by Databricks when users “Pre-Schedule” “Notebooks” to run.



- **Jobs** - “Jobs” are ways to run “Notebooks” on a scheduled basis. “Jobs” are used to automate Data Engineering, Data Science and Data Analytics Workloads.



- **Libraries** - “Libraries” are “Packages”, or, “Modules” that are used to extend Databricks Functionality. “Libraries” make “Third-Party”, or, “Locally-Built” code available to Data Practitioners using Databricks.
“Libraries” can be written locally and uploaded manually by Data Practitioners, or, can be installed directly into Databricks.
“Libraries” can be accessed on Databricks via “Shared Folders” in the “Workspace”, as part of the specially configured “Clusters” that have the “Libraries” installed, or, solely within the context of a “Notebook” into which “Libraries” are invoked.



- **Data** - Data Practitioners using Databricks, use “Traditional SQL Tables” that contain “Structured Data”. Currently, “Tables” can be stored on “Amazon S3”, or, “Azure Blob Storage”.
“Table Schema” is stored in the default “Databricks Internal Metastore”. Databricks users can also configure and use “External Metastores”.

Data Practitioners can import data into a “Distributed File System” by “mounting” data from External Data Source, or, a wide variety of Apache Spark Data Sources, into a Databricks “Workspace” and work with it in Databricks “Notebooks”, and, “Clusters”.

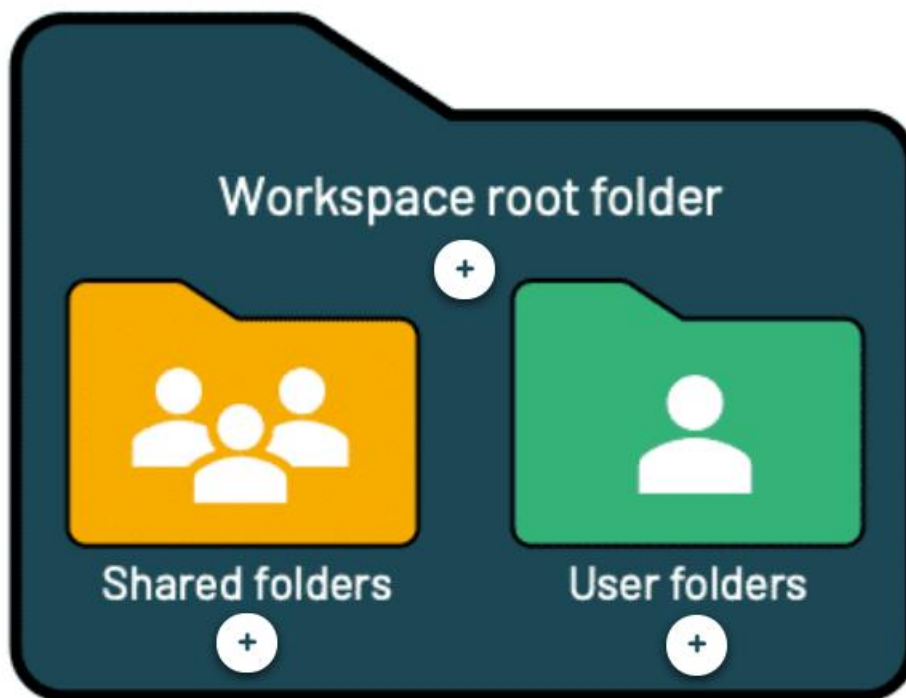


- **Experiments** - “Experiments” refer to the “Machine Learning Experiments” that Data Practitioners run inside of the “Workspace”. “Experiments” are managed, along with the Machine Learning Models, with Managed MLFlow, which is accessible from the “Workspace”.



- ✚ **Folders** - “Workspace Folders” help organize all of the “Assets” within a Databricks “Workspace”. “Folders” contain all “Static Assets” within a “Workspace” including “Notebooks”, “Libraries” and “Experiments”. “Folders” can also contain other “Folders”. Following are the different types of “Folders” in Databricks -

- **Workspace Root Folder** - The “Workspace Root Folder” is the “Container” for all of an organization’s Databricks “Static Assets”. It contains an organization’s “Shared Folders” and “User Folders”.
- **Shared Folder** - “Shared Folders” share “Assets”, or, Objects across an organization. All “Users” within a “Workspace” typically have “Full Permission” for all of the “Assets” and Objects in “Shared Folder”, unless set otherwise by enabling “Access Control” and setting “User Permissions”.
- **User Folder** - Each Databricks “Users” within a “Workspace” has their own “Folder”. This “Folder” contains all of the “Assets” belonging to that “User”.



What Happens When Notebooks are Attached and Detached to Clusters

- ✚ When a “Notebook” is attached to a “Cluster”, “Databricks” creates an “Execution Context”. An “Execution Context” contains the “State” for a “REPL Environment” for each supported Programming Language: “Python”, “R”, “Scala”, and, “SQL”. When a “Cell” is run in a “Notebook”, the command is dispatched to the appropriate Language “REPL Environment” and run.
- ✚ When a “Notebook” is detached from a “Cluster”, the “Execution Context” is removed, and, all “Computed Variable Values” are cleared from the “Notebook”.

Controlling Notebook Access

- By default, all “Users” can create, and, modify “Workspace Assets” - including “Notebooks”, “Experiments”, and, “Folders”, unless an “Administrator” enables “Workspace Object Access Control”. With “Workspace Object Access Control”, individual “Permissions” determine a “User’s abilities”.
- Before one can use “Workspace Object Access Control”, a Databricks “Workspace Administrator” must enable it.

Ability	No permissions	Read	Run	Edit	Manage
View cells		X	X	X	X
Comment		X	X	X	X
Run via % or notebook workflows		X	X	X	X
Attach/detach notebooks			X	X	X
Run commands			X	X	X
Edit cells				X	X
Change permissions					X

- To configure “Notebook Permissions”, in the “Notebook Context Bar”, click on “Permissions”.



- Then, in the “Add Users and Groups:” dropdown, select the “Users” to give access to the “Notebook”. It is possible to select all the “Users”, if desired.

Permission Settings for: **Loan Risk Analysis**

Who has access:

admins (group)	Can Manage	⌵ ⓘ
Demo Student1 (class+demo1@databricks.com)	Can Manage	⌵ ⓘ

Add Users and Groups:

Select User or Group... Can Read ⌵ ⓘ Add

Groups

- all users

Users

- Demo Student2 (class+demo2@databricks.com)
- Demo Student4 (class+demo4@databricks.com)
- Demo Student5 (class+demo5@databricks.com)

Done

After that, select the “Permission” to give the “Users”, or, “Groups”. To change the “Permission” of a “User”, or, “Group”, select the new “Permission” from the “Permission:” dropdown. Then, click on the “Save Changes” button to save the changes.

Permission Settings for: **Loan Risk Analysis**

You have made changes that you need to save

Who has access:

admins (group)	Can Manage	⌵ ⓘ
all users (group)	Can Run	⌵ ⓘ ✕
Demo Student1 (class+demo1@databricks.com)	Can Manage	⌵ ⓘ

Add Users and Groups:

Select User or Group... Can Run ⌵ ⓘ Add

Cancel Save Changes

RELATIONAL ENTITIES ON DATABRICKS

Relational Entities on Databricks

- While the Syntax for creating, and, working with “Databases”, “Tables”, and, “Views” will be familiar to most SQL Users, some default behaviors may be different in Databricks.

What is Hive Metastore

- There is already a “Hive Metastore” on Databricks, which is configured and run by the Databricks for the “Users”. Optionally, the Users can bring their own “Hive Metastore” on Databricks. The specific behavior of configuring the “Hive Metastore” that the Users are bringing will vary.
- “Hive Metastore” stores the information, i.e., the “Metadata” necessary to query the various “Relational Entities” -
 - **Schemas** - The “Schema” of each of the Databricks “Tables” in the “Workspace”.
 - **Locations** - The “Location” of the data that is underlying those Databricks “Tables”.

The data of “Hive Metastore” is stored and managed in its own “Relational Database”, which is run as a separate Service that is managed by Databricks. That Service is not tied to “Lifetime of any Cluster”.

What is Spark Catalog

- The “Spark Catalog” is used within each “Spark Session”, i.e., within a “Notebook”, or, a “Job”, and, provides an “Interface” to interact with or manage the “Hive Metastore”.
- Each “Spark Session” initializes a “Metadata Catalog”.

Databases on Databricks

- In Apache Spark, “Database” is a “Logical Construct” that organizes “Tables”. Apache Spark interchangeable uses the Keyword “Schema” with “Database” when DDL is used to create the Databricks “Relational Entities”.
- The reason for using “Databases” in Databricks is that the “Databases” simplify the “Discovery and Management” of the data stored in diverse “Locations”.
- Each “Database” is created against a “Physical Location” on an “Object Store”. The “Database” itself does not exist in the “Database Directory”. The “Database Directory”

becomes the “Directory”, where the data of the “Managed Tables”, created inside the concerned “Database”, are written.

- ✚ The default location of any “Database” in Databricks is determined by “Spark Config” using the command - `spark.sql.warehouse.dir`, which can be changed.

Find the Default Location of a "Database" in Databricks Using "spark.config"

```
1 print(spark.conf.get('spark.sql.warehouse.dir'))  
  
/user/hive/warehouse
```

- ✚ The “Location” Keyword allows to specify a “Directory Path” during the “Database” creation, and, inside that “Directory Path”, data of all the “Managed Tables” created inside that “Database” will end up. By default, a “Database” in the Databricks is created in the location - `dbfs/user/hive/warehouse/<database_name>.db`.

Find the Location of "retailer_db"

```
1 %sql  
2  
3 describe database retailer_db
```

	database_description_item ▲	database_description_value ▲
1	Database Name	retailer_db
2	Comment	
3	Location	dbfs:/user/hive/warehouse/retailer_db.db
4	Owner	root

Showing all 4 rows.

Most customers would want to avoid writing sensitive information to the default “Directory Path”. Instead, customers would want to define either -

- A default “Directory Path” for all the “Databases” in the “Workspace”, or,
- Different default “Directory Path” for the different “Databases” of the different Data Teams.

In both the cases, the approaches provide a “Secured Access” to the data, where the “Secured Access” can be configured at an “Account Level”.

Tables on Databricks

- ✚ In Databricks, a “Table” is a “Directory of Files” that is registered as a “Table Relation” within a “Database”. The “Underlying Files” can be of any type. The default “File Type” in Databricks is “Parquet”.
- ✚ The idea, or, concept of a “Table” being a “Directory of Files” is so strongly supported in Apache Spark that “Spark SQL” actually supports an alternate Syntax of querying, where a “Directory of File” can be referred to as a “Table”, and, the query can be performed on “<the_file_format>.<path_to_the_files>”. This way a “Directory of Files”, or, an individual file can be queried upon.
- ✚ When a “Table” is created in Databricks, the “Data Location”, “File Type”, and, the “Schema” are registered in the “Hive Metastore”, along with some additional properties that are configured based on the “File Type” used to create the “Table”.
- ✚ The “Tables” created in the Databricks will persist at the “Workspace” level within a “Database”, and, not at the “Job” level, or, “Notebook” level.
- ✚ A “Table” is analogous to persisting a properly configured “DataFrameReader” for a “Dataset”. A “Table” can be thought of as a “Definition” for how a “File” would be loaded, which is stored at a given “Location”, of a given “Type” with a given “Schema”, back into Apache Spark so that the Data Practitioners can execute “Queries” upon the data of the “File”.

Unmanaged Tables on Databricks

- ✚ In Databricks, an “Unmanaged Table”, is synonymous with “External Table”. So, the “External” Keyword is referred to as “Unmanaged Table” in Databricks.
- ✚ There are two ways to create an “Unmanaged Table” -
 - Specifying the “Location” to save the data when creating a new “Table”.
 - Provide the “Location” of existing data when registering a “Table”.
- ✚ In both the cases, the “Location” of the “Underlying Data” is mentioned explicitly when the “Table” is created. This specificity of the “Location” of the “Underlying Data” is what creates an “Unmanaged Table”.
- ✚ The “life time” of the “Files”, used to create an “Unmanaged Table”, is not tied to the “life time” of the “Unmanaged Table” existing in the “Hive Metastore”.
 - Dropping an “Unmanaged Table” removes the “Metadata” information from the “Hive Metastore”, but, the “Underlying Data” is not dropped.

- Copying, or, cloning an “Unmanaged Table” from one “Database” to another “Database”, or, changing the name of an “Unmanaged Table” can be done without moving the “Underlying Data”, because, the “Underlying Data” is still in the same “Location”, and, only the registrations are changed in “Hive Metastore”.

Following is a code of creating “Unmanaged Table” using “Spark SQL” -

```
spark.sql("""
    create table if not exists retailer_db.customerTbl (
        c_customer_sk long comment \"This is the Primary Key\",
        c_customer_id string,
        c_current_demo_sk long,
        c_current_hdemo_sk long,
        c_current_addr_sk long,
        c_first_ship_to_date_sk long,
        c_first_sales_date_sk long,
        c_salutation string,
        c_first_name string,
        c_last_name string,
        c_preferred_cust_flag string,
        c_birth_day int,
        c_birth_month int,
        c_birth_year int,
        c_birth_country string,
        c_login string,
        c_email_address string,
        c_last_review_date long
    )

    using csv
    options (
        path 'dbfs:/FileStore/tables/retailer/data/customer.dat',
        sep '|',
        header true
    )
""")
```

Verifying from “Hive Metastore” if the created “Unmanaged Table”, i.e., “retailer_db.customerTbl” is in fact “Unmanaged” using “Spark SQL” command -

```
1 display(spark.sql('describe extended retailer_db.customerTbl'))
```

	col_name	data_type	comment
25	Last Access	UNKNOWN	
26	Created By	Spark 3.0.1	
27	Type	EXTERNAL	
28	Provider	csv	
29	Location	dbfs:/FileStore/tables/retailer/data/customer.dat	
30	Serde Library	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe	
31	InputFormat	org.apache.hadoop.mapred.SequenceFileInputFormat	

Showing all 33 rows.

Following is a code of creating “Unmanaged Table” using “DataFrame” in “Python” -

```
ddl = """c_customer_sk long comment \"This is the Primary Key\",
        c_customer_id string,
        c_current_cdemo_sk long,
        c_current_hdemo_sk long,
        c_current_addr_sk long,
        c_first_shipto_date_sk long,
        c_first_sales_date_sk long,
        c_salutation string,
        c_first_name string,
        c_last_name string,
        c_preferred_cust_flag string,
        c_birth_day int,
        c_birth_month int,
        c_birth_year int,
        c_birth_country string,
        c_login string,
        c_email_address string,
        c_last_review_date long"""

customerDf.write\
    .options(
        path = "/tmp/tables/retailer_db.db/customerDfPtTbl",\
        schema = ddl\
    )\
    .saveAsTable("retailer_db.customerDfPtTbl")
```

Verifying from “Hive Metastore” if the created “Unmanaged Tables”, i.e., “retailer_db.customeTbl”, and, “retailer_db.customerDfPtTbl” are in fact “Unmanaged” using “Spark Catalog” command in “Python” -

```
1 spark.catalog.listTables(dbName='retailer_db')

▶ (8) Spark Jobs

Out[29]: [Table(name='cataloguesalestbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='customeraddresstbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='customerdfpttbl', database='retailer_db', description=None, tableType='EXTERNAL', isTemporary=False),
Table(name='customertbl', database='retailer_db', description=None, tableType='EXTERNAL', isTemporary=False),
Table(name='customerwithaddresstbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='datedimtbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='itemtbl', database='retailer_db', description=None, tableType='EXTERNAL', isTemporary=False),
Table(name='websalestbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='v_ak_addresses', database=None, description=None, tableType='TEMPORARY', isTemporary=True),
Table(name='v_customeraddress', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]

Command took 1.24 seconds -- by oindrila.chakraborty2988@gmail.com at 6/21/2021, 1:20:43 AM on cluster-oindrila
```

Managed Tables on Databricks

- ✚ “Managed Tables” are the default behavior when creating, or, saving “Tables” with either “Spark SQL”, or, “DataFrame” API.
- ✚ There are two ways to create an “Managed Table” -
 - Create a “Non-Empty Managed Table” by saving results from a “Spark SQL” Query, or, result from a “DataFrame” API Query.
Creating a “Non-Empty Managed Table”, will write the “Underlying Data” to a new “Directory” inside the “Database Directory” The default “Location” of a “Managed Table” is -
`dbfs/user/hive/warehouse/<database_name>.db/<new_table_name>/.`
 - It is also possible to create an “Empty Managed Table” using Spark SQL DDL, and, then load data into the “Directory” of the created “Empty Managed Table”, and, run “`MSCK REPAIR TABLE`” command.
This is a non-standard workflow.
- ✚ The Data Lifecycle of a “Managed Table” is managed by “Hive Metastore”.
 - Dropping a “Managed Table” deletes the “Underlying Data” permanently.
 - Changing the “Relational Structure” of a “Managed Table”, or, changing the name of a “Managed Table”, would require the “Underlying Data” to be physically moved and re-written to a new “Location”.
In this case, the “Files” are not only copied from one “Location” to another, the following operations are also performed -
 - ✓ The “Underlying Data” is “De-Serialized”.
 - ✓ The “Underlying Data” is loaded into Apache Spark.
 - ✓ The “Underlying Data” is written back to a new “Location”.
For very large “Datasets”, this can be extremely cost-prohibitive and time-consuming.
- ✚ Following is a code of creating an “Empty Managed Table” using “Spark SQL” -

```
spark.sql("""create table if not exists retailer_db.customerWithAddressTbl(
    ca_address_sk long comment \"This is the Primary Key\",
    ca_address_id string,
    ca_street_number long,
    ca_street_name string,
    ca_street_type string,
    ca_suite_number string,
    ca_city string,
    ca_county string,
    ca_state string,
    ca_zip long,
    ca_country string,
    ca_gmt_offset int,
    ca_location_type string)""")
```

Load data into the created “Empty Managed Table”, i.e., “retailer_db.customerWithAddressTbl”.

```
spark.sql("""insert into retailer_db.customerWithAddressTbl
    select  ca_address_sk,
            ca_address_id,
            ca_street_number,
            ca_street_name,
            ca_street_type,
            ca_suite_number,
            ca_city,
            ca_county,
            ca_state,
            ca_zip,
            ca_country,
            ca_gmt_offset,
            ca_location_type
    from retailer_db.customerAddressTbl cat
    inner join retailer_db.customerTbl ct
    on cat.ca_address_sk = ct.c_current_addr_sk
""")
```

Verifying from “Hive Metastore” if the created “Managed Table”, i.e., “retailer_db.customerWithAddressTbl” is in fact “Managed” using “Spark SQL” command -

```
1 display(spark.sql("describe formatted retailer_db.customerWithAddressTbl"))
```

	col_name	data_type	comment
21	Created By	Spark 3.0.1	
22	Type	MANAGED	
23	Provider	hive	
24	Table Properties	[transient_lastDdlTime=1624221212]	
25	Location	dbfs:/user/hive/warehouse/retailer_db.db/customerwithaddresstbl	
26	Serde Library	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe	
27	InputFormat	org.apache.hadoop.mapred.TextInputFormat	

Showing all 30 rows.

Following is a code of creating “Managed Table” using “DataFrame” in “Python” -

```
ddl = """ca_address_sk long comment \"This is the Primary Key\",
        ca_address_id string,
        ca_street_number long,
        ca_street_name string,
        ca_street_type string,
        ca_suite_number string,
        ca_city string,
        ca_county string,
        ca_state string,
        ca_zip long,
        ca_country string,
        ca_gmt_offset int,
        ca_location_type string"""

customerAddressDf = spark.read.options(\
    header = "true",\
    sep = "|",\
    schema = ddl\
)\
.csv("dbfs:/FileStore/tables/retailer/data/customer_address.dat")

customerAddressDf.write.saveAsTable("retailer_db.customerAddressTbl")
```

Verifying from “Hive Metastore” if the created “Managed Tables”, i.e., “retailer_db.customerWithAddressTbl”, and, “retailer_db. customerAddressTbl” are in fact “Managed” using “Spark Catalog” command in “Python” -


```

1 spark.catalog.listTables(dbName='retailer_db')

▶ (8) Spark Jobs

Out[40]: [Table(name='cataloguesalestbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='customeraddresstbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='customerdfpttbl', database='retailer_db', description=None, tableType='EXTERNAL', isTemporary=False),
Table(name='customertbl', database='retailer_db', description=None, tableType='EXTERNAL', isTemporary=False),
Table(name='customerwithaddresstbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='datedimtbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='itemtbl', database='retailer_db', description=None, tableType='EXTERNAL', isTemporary=False),
Table(name='websalestbl', database='retailer_db', description=None, tableType='MANAGED', isTemporary=False),
Table(name='v_ak_addresses', database=None, description=None, tableType='TEMPORARY', isTemporary=True),
Table(name='v_customeraddress', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]

Command took 1.23 seconds -- by oindrila.chakraborty2988@gmail.com at 6/21/2021, 2:27:27 AM on cluster-oindrila

```

Recommendation from Databricks about Usage of Managed Tables on Databricks

- ✚ It is recommended by Databricks to use “Unmanaged Tables” from the beginning for “Production Data”, or, “Mission-Critical Data”.
- ✚ For “Managed Tables”, use the “Tables” sparingly, and, in an informed way. Also make sure to have proper lock set to those “Managed Table”, so that any User does not accidentally drops any “Managed Table” with “Mission-Critical Data”.

Views on Databricks

- ✚ A “View” registers “Logical Queries” using the “Physical Plan” generated by the “Catalyst Optimizer”.
- ✚ “View” is functionally equivalent to a Spark “DataFrame” in the sense that a write-back “Query” is defined against a Data Source, or, several Data Sources.
- ✚ Each time a “View” is materialized, the results are “Re-Calculated”, i.e., the “Underlying Physical Plan” is executed against the Data Source.
- ✚ There are three types of “Views” in Databricks. All the different types of “Views” have different “Scoping”, or, “Persistence” -
 - **View** - The default “View” saves the “Logical Query”, i.e., the “Underlying Physical Plan” against the Data Source, to the “Hive Metastore” associated with a “Database”. The default “View” will be available across the “Workspace” and between the “Spark Sessions”.
 - **Temp View** - A “Temp View” is “Ephemeral”, i.e., lasts for a very short time. A “Temp View” is also an isolated entity to “Spark Session”, i.e., only exists within one “Notebook”, or, one “Job”.
In Databricks, “Local Table” is synonymous with “Temp View”.


```
customerAddressDf = spark.read\
    .option("header", "true")\
    .option("sep", "|")\
    .option("inferSchema", "true")\
    .csv("dbfs:/FileStore/tables/retailer/data/customer_address.dat")

customerAddressDf.createTempView("v_customerAddress")
```

The entries with “TEMPORARY” as “tableType” are the “Temp Views” created.

```
1 spark.catalog.listTables()

▶ (3) Spark Jobs

Out[41]: [Table(name='demotbl', database='default', description=None, tableType='MANAGED', isTemporary=False),
Table(name='v_ak_addresses', database=None, description=None, tableType='TEMPORARY', isTemporary=True),
Table(name='v_customeraddress', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]

Command took 1.34 seconds -- by oindrila.chakraborty2988@gmail.com at 6/21/2021, 2:31:28 AM on cluster-oindrila
```

- **Global Temp View** - A “Global Temp View” is also “Ephemeral”, i.e., lasts for a very short time, but, tied to the life time of the “Cluster”, and, shared across a “Spark Context”.

“Global Temp Views” are registered to a special kind of “Database”, i.e., “global_temp”, which is created by default for each “Spark Context”.

```
from pyspark.sql.functions import col

customerAddressDf.\
    select("ca_address_sk", "ca_country", "ca_state", "ca_city", "ca_street_name").\
    where(col("ca_state").contains("AK")).\
    createTempView("v_AK_Addresses")

display(spark.sql("select * from v_AK_Addresses"))

customerAddressDf.createGlobalTempView("gv_customerAddress")
```

To find all the “Global Temp Views”, pass the special kind of “Database” name, i.e., “global_temp” to the command “`spark.catalog.listTables()`” as an Argument to “dbName”.

```
1 spark.catalog.listTables(dbName='global_temp')

▶ (3) Spark Jobs


Out[45]: [Table(name='gv_customeraddress', database='global_temp', description=None, tableType='TEMPORARY', isTemporary=True),
Table(name='v_ak_addresses', database=None, description=None, tableType='TEMPORARY', isTemporary=True),
Table(name='v_customeraddress', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]

Command took 0.25 seconds -- by oindrila.chakraborty2988@gmail.com at 6/21/2021, 2:34:34 AM on cluster-oindrila
```

BASIC ARCHITECTURE OF APACHE SPARK

Components of Apache Spark Architecture

- ✚ Apache Spark is a “Sophisticated Distributed Computation Framework” for executing code in Parallel across many different machines. While the Abstraction and Interfaces are simple, managing “Clusters of Computers” and ensuring “Production-Level Stability” is not. Databricks makes Big Data simple by providing Apache Spark as a “Hosted Solution”, where much of the Spark setup will be managed by Databricks for the Users.
- ✚ Apache Spark uses “Clusters of Computers” to process Big Data by breaking a large Task into smaller ones and distributing the work among several Machines. Following are the Components that Apache Spark uses to co-ordinate work across a “Clusters of Computers”.
 - **Cluster and Nodes** - A “Cluster” is a group of “Nodes”. In conceptual level, a “Cluster” is comprised of a “Driver” and multiple “Executors”.
“Nodes” are individual Machines within a “Cluster”. “Nodes” are mostly “Virtual Machines”, like the “VM” in Microsoft Azure.
 - ✓ Databricks runs a very specific configuration. In that configuration, the “Driver” runs on one “Node” on a single “JVM”, and, each of the “Executors” runs on a single “Node” on a single “JVM”. This way “Cross-Contamination” can be avoided and between “Executors”, and, it makes overall management of the System much simpler.
 - **Driver** - The “Driver” is at the heart of the “Spark Architecture. The “Driver” is the Machine in which the “Spark Application” runs. It is responsible for the following things -
 - ✓ Running and maintaining information about the “Spark Application”.
 - ✓ Analyzing, Distributing, and, scheduling work across the “Executors”, i.e., while processing the “Spark Application” the “Driver” divides it into “Jobs”, “Stages”, and, “Tasks”. Since, “Tasks” are, in turn, assigned to specific “Slots” in an “Executor”, it is the “Driver’s” job to keep track of -
 - Which “Executor” is running which “Task”.
 - Whether the assigned “Tasks” have succeeded or failed.
 - Whether an “Executor” is free to be assigned another “Task”, if there are more “Tasks” in “Queue” etc.
 - ✓ If a “Driver” is to produce a result, like finding out how many records are there in a “Dataset” using a “count” operation, then “Driver” receives the result and make further decision based on the received result.
 - ✓ Responding to user’s program.



In a single “Databricks Cluster”, there will be only one “Driver”, regardless of the number of “Executors”.

- **Executor** - In the simplest term, an “Executor” is an instance of a “Java Virtual Machine”, which is running a “Spark Application”, and, is designed to receive instruction from “Driver”.

An “Executor” also provides an “Environment” in which “Tasks” can be run. Each “Executor” has a number of “Slots”. Each “Slot” can be assigned a “Task”. So, an “Executor” leverages the natural capabilities of a “Java Virtual Machine” to execute many “Threads”. Example - if an “Executor” has “4 Cores”, then there will be “4 Threads”, or, “4 Slots” for the “Driver” to assign “Tasks” to that “Executor”.

The “Executors” are responsible for carrying out the work assigned by the “Driver”. Each “Executor” is responsible for two things -

- ✓ Execute the codes assigned by the “Driver”.
 - ✓ Report the “State of the Computation” back to the “Driver”.
- **Partition** - A “Partition” is a single slice of a larger “Dataset”. Apache Spark divides a large “Dataset” into no more than 128MB chunks of data.
 - ✓ As Apache Spark processes those “Partitions”, it assigns “Tasks” to those “Partitions”. One “Task” will process one and only one “Partition”. This “One-to-One Relationship” between the “Task” and “Partition” is one of the means by which Apache Spark maintains its “Stability”, and, “Repeatability”.
 - ✓ The actual size of each “Partition” and where the data gets split are decided by the “Driver”.
 - ✓ The size and limitation of a “Partition” is controlled by a handful of “Configuration Option”. The initial size of “Partition” is partially adjustable with various “Configuration Options”. For all practical purposes, tweaking “Configuration Options” is not going to achieve any real significant performance gain in vast majority of scenarios.
 - ✓ Each “Executor” will hold a chunk of the data to be processed. Each of these chunks is called a “Spark Partition”. A “Partition” is a “Collection of Rows” that sits on one Physical Machine in the “Cluster of Computers”.
 - ✓ One thing to note is that “Spark Partition” is completely separated from “Hard Disk Partitions”, which have to do with the Storage Space on a Hard Drive.

- **Jobs** - In the Databricks Environment, when a “Spark Application” is submitted to the “Driver”, it is going to sub-divide the “Spark Application” into its Natural Hierarchy of - “Jobs”, “Stages” and “Tasks” respectively, where one “Action” may create one, or, more “Jobs”.

The secret to Apache Spark’s performance is “Parallelism”. Each “Parallelized Action” is referred to as a “Job”.

- ✓ **Stage** - Each “Job” is broken down into “Stages”, which is a “Set of Ordered Steps” that together accomplishes a “Job”.

A “Stage” cannot be completed until all the “Tasks” of that “Stage” are completed. If any of the “Tasks” inside a “Stage” fails, the entire “Stage” fails.

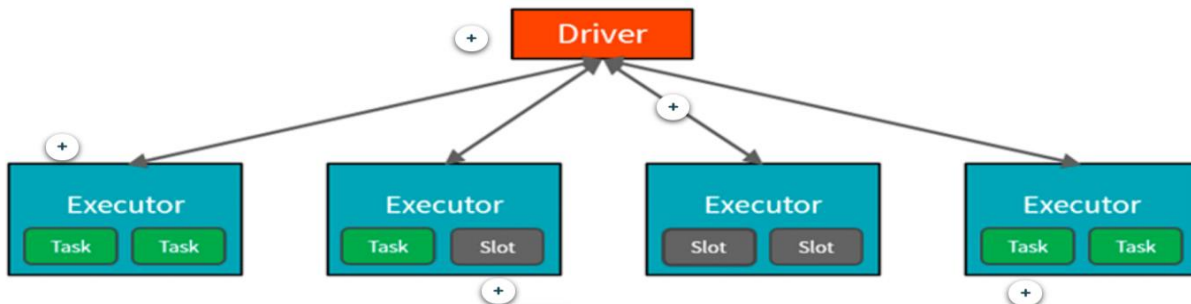
One “Long-Running Task” can delay an entire “Stage” from completing. The “Shuffle” between two “Stages” is one of the most expensive operations in Apache Spark.

- ✓ **Task** - “Tasks” are the “Smallest Unit of Work”. “Tasks” are created by the “Driver” and assigned a “Partition” of data to process. Then “Tasks” are assigned to “Slots” for Parallel Execution. Once started, each “Task” will fetch its assigned “Partition” from the original Data Source.

- **Slots** - “Slot” represents the “Lowest Unit of Parallelization”. Example - if an “Executor” has “4 Cores”, then that “Executor” can only do “4 Tasks” at a time. A “Slot” executes a “Set of Transformations” against a single “Partition” as directed by the “Driver”. A “Driver” picks a single “Partition”. For that selected “Partition”, the “Driver” creates a “Task”, and, assigns that “Task” to a “Slot” in an “Executor”. That “Slot” executes the operations, dictated to the “Task” by the “Driver”.

Apache Spark “Parallelizes” at two levels -

- ✓ One is “Splitting the work among “Executors”.
- ✓ The other is the “Slot”. “Slots” are also interchangeably called as “Threads”, or, “Cores” in Databricks.

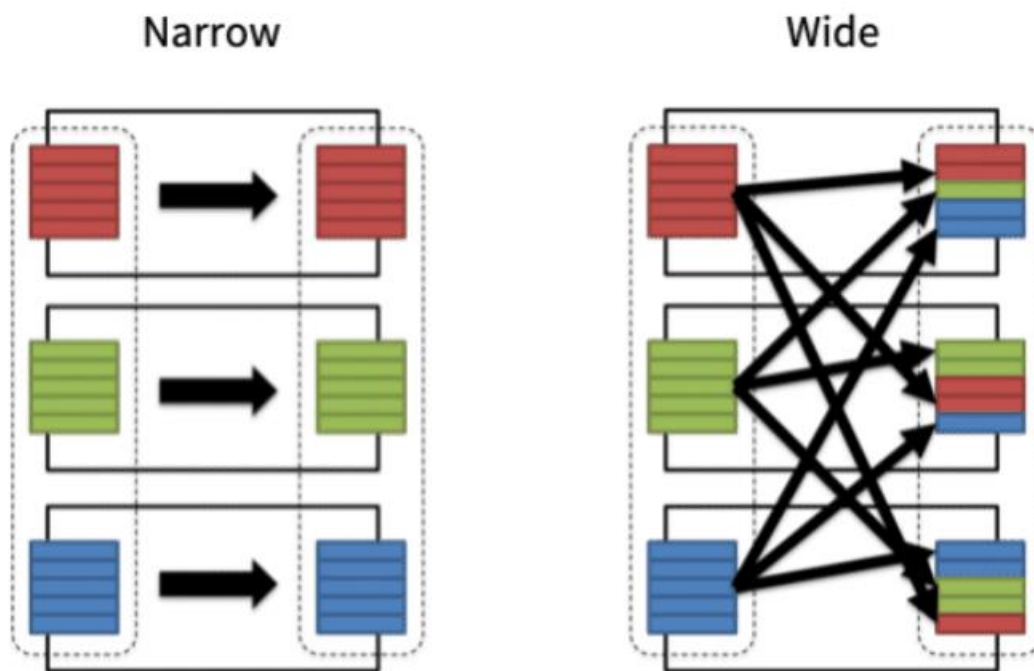


Introduction to DataFrames

- ✚ In the Databricks environment, the Data Team will access and manipulate data in “Tables”. “Databricks Tables” are “Structured Data” that the Data Team will use for Analysis. “Databricks Tables” are equivalent to Apache Spark “DataFrames”.
A “DataFrame” is an Immutable, “Distributed Collection of Data” organized into “Named Columns”. In concept, a “DataFrame” is equivalent to a “Table” in a “Relational Database”, except that, “DataFrames” carry important “Metadata” that allows Apache Spark to optimize queries.

Operations on DataFrames

- ✚ **Lazy Evaluation** - “Lazy Evaluation” refers to the idea that Apache Spark waits until the last moment to execute a “Series of Operations”. Instead of modifying the data immediately when some Operation is expressed, Apache Spark builds up a “Plan of Transformations” that will be applied on the Source Data. This “Plan” is executed when an “Action” is called.
“Lazy Evaluation” makes it easier to “Parallelize Operations” and allow Apache Spark to apply various “Optimizations”.
 - ✚ **Transformations** - “Transformations” are at the core of how to express “Business Logic” in Apache Spark. “Transformations” are the “Instructions” used, to modify a “DataFrame” to get the desired results.
 - **Why Transformations are Called “Lazy Operations”** - “Transformations” are called “Lazy Operations”, because, the “Transformations” will not be completed at the time the Data Team members write and execute the code in a “Cell” - these will only get executed once an “Action” is called.
- There are two types of “Transformations” -
- **Narrow** - For “Narrow Transformations”, the data required to compute the records in a “Single Partition” resides in at most “One Partition” of the “Parent Dataset”.
“Narrow Transformations” mean that work can be computed and reported back to the “Executor” without changing the way the data is “Partitioned” over the System.
 - **Wide** - For “Wide Transformations”, the data required to compute the records in a “Single Partition” may reside in “Many Partition” of the “Parent Dataset”.
“Wide Transformations” requires that the data be “Re-Distributed” over the System. This is called a “Shuffle”. “Shuffles” are Triggered when data needs to move between “Executors”.



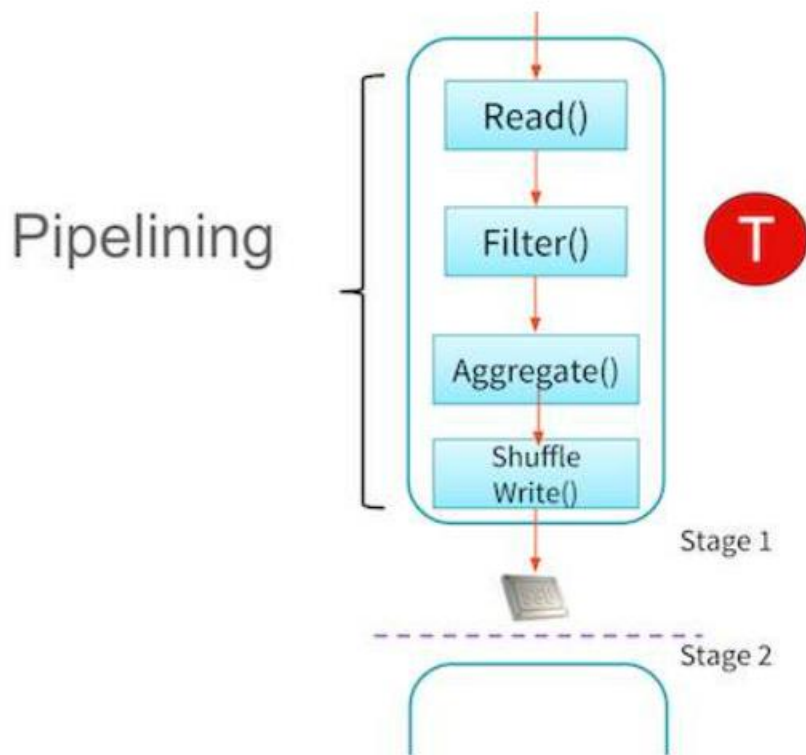
✚ **Actions** - “Actions” are “Statements” that are computed and executed when encountered in the Developer’s code. “Actions” are not postponed or waited for other code constructs.

While “Transformations” are “Lazy”, “Actions” are “Eager”.

Pipeline of Computations

✚ “Lazy Evaluation” allows Apache Spark to “Optimize” the entire “Pipeline of Computations” as opposed to the individual pieces. This makes it exceptionally fast for certain types of computation, because, it can perform all relevant computations at once.

Technically speaking, Apache Spark “Pipelines” the computation in the below image -



From the above image, it can be understood that certain computations can all be performed at once (like a Map and a Filter), rather than having to do one operation for all pieces of data and then the following operation.

- ✚ Apache Spark can keep results “in-memory” as opposed to other Frameworks that immediately write to Disk after each “Task”.

Catalyst Optimizer

- ✚ The “Catalyst Optimizer” is at the core of “Spark SQL’s” power and speed. It automatically finds the most efficient “Plan” for applying the “Transformations” and “Actions”. The following stages occur as the “Input Query” travels through the “Optimization Process”, i.e., “Queries” are processed and “Optimized” in Spark 2.x -
 - **User Input** - Users can specify the queries using the “User-Facing APIs”, like “SQL Query”, “Dataset”, “DataFrame” etc.
 - **Unresolved Logical Plan** - This is the “Logical Plan” to transform the data. It is called “Unresolved” at this stage because, some Elements, like “Table Names”, “Column Names”, for example, have not been resolved yet. At this stage, those Elements might not exist.

For any “Query”, the “Catalyst Optimizer” accepts the “Unresolved Logical Plan” submitted by the User, and, checks against the “Metadata Catalog” to be sure that there won’t be any “Naming”, or, “Typing” errors.

- **Analysis** - At this stage, the the “Table Names” and “Column Names” are validated against the “Metadata Catalog”. The “Metadata Catalog” is used to refer to the “Metadata”, about the data, stored in the “Table”. At this stage, the “Unresolved Logical Plan” becomes the “Logical Plan”.

The “Unresolved Logical Plan” is checked against the “Metadata Catalog”, and, then turned into a “Logical Plan”.

- **Optimized Logical Plan** - This is the stage where the first set of “Optimizations” take place. At this stage, the “Query” may be optimized by “Re-Ordering” the sequence of commands.

The “Logical Plan” is adjusted to recognize the Query Order, and, make it as efficient as possible.

- **Physical Planning** - At this stage, the “Catalyst Optimizer” generated one or more “Physical Plans” that can be used to execute the “Input Query”. Each “Physical Plan” represents what the “Query Engine” will actually do after all “Optimizations” have been applied.

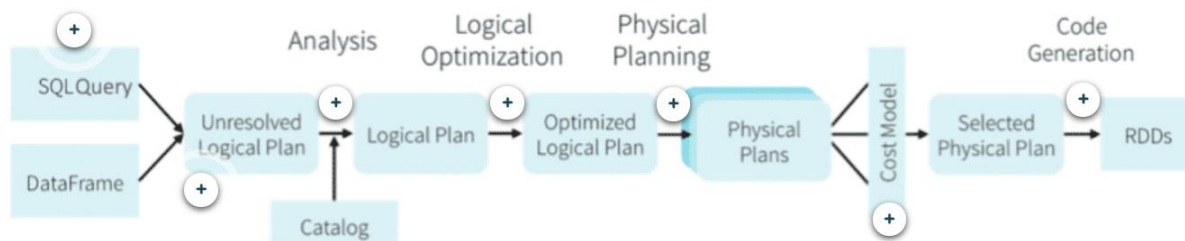
Apache Spark determines how the “Logical Plan” will physically execute across the “Cluster”. At this stage, Apache Spark generates several “Physical Plans”.

- **Cost Model** - At this stage, each “Physical Plan” is evaluated according to its “Cost Model”. The best performing model is selected. This provides the “Selected Physical Plan”.

“Physical Plans” are compared through a “Cost Model”, and, the best one is selected.

- **Code Generation** - The “Selected Physical Plan” is “Compiled” to “Java Byte Code” and executed.

Then the “Catalyst Optimizer” converts the “Selected Physical Plan” to “RDDs”, and, then generates “Bytecodes” for the “Java Virtual Machine”.



- ✚ **Limitation of Catalyst Optimizer** - The “Catalyst Optimizer” in Spark 2.x applies “Optimizations” throughout “Logical” and “Physical Planning Stages”. It generates a selection of “Physical Plans” and selects the most efficient one. These are “Rule-Based Optimizations”, and, while these generally improve the “Query Performances”, these are all based on “Estimates” and “Statistics” that are generated before runtime. There may be unanticipated problems, or, tuning opportunities that appear as the “Query” runs.

Caching

- ✚ Moving data around is expensive, both in Time and Money. In practice, the work will be connected to a “Central Data Store”, and, the “Queries” that runs will pull data from there. In general, each time a “Query” is run, the work is going all the way back to the original “Central Data Store” to get the data. All of this overhead can be avoided by “Caching”.
One of the significant parts of Apache Spark is its ability to store data “in-memory” during computation. This is a neat trick that can be used as a way to speed up access to commonly Queried Tables”, or, pieces of data. This is also great for “Iterative Algorithms” that works over and over again on the same data.
However, other concepts like “Data Partitioning”, “Clustering”, and, “Bucketing” can end up having a much greater effect on the execution of the “Job” than “Caching”.
- ✚ **When to Cache** - In applications that reuse the same “Datasets” over and over again, one of the most useful “Optimizations” is “Caching”. “Caching” will place a “DataFrame”, or, a “Table” into “Temporary Storage” across the “Executors” in the “Cluster of Computers”, and, makes subsequent reads faster.
Once data is “Cached”, the “Catalyst Optimizer” will only reach back to the location where the data is “Cached”.
- ✚ **When Not to Cache** - Since, “Caching” is an expensive operation itself, if a “Dataset” is used only once for example, the cost of “Pulling” and “Caching” the data is greater than the cost of original “Data Pull”.

Shuffling in Apache Spark

- ✚ “Shuffling” is the process of “Re-Arranging Data” within a “Cluster of Computers” between “Stages”. This is a very expensive process. One of the most significant, yet often avoidable, cause of “Performance Degradation” are the “Shuffle Operations”.
- ✚ “Shuffling” is “Triggered” by the “Wide Transformation” operations, like the following -
 - Re-Partitioning
 - By-Key Operations (except “Counting”)

- Joins, the worse being “Cross Join”
- Sorting
- Distinct
- GroupBy

✚ To assist the “Catalyst Optimizer” for the best automatic “Optimization”, multiple “Wide Transformation” operations can be grouped together, and, executed back-to-back.

- **Aim for**: Narrow, Narrow, Wide, Wide, Wide, Narrow.
- **Avoid**: Narrow, Wide, Narrow, Wide, Narrow, Wide.