Program: **MSc of Data Science**

Module: **Big Data Tools and Techniques**

Week 5

Lakehouse in the Databricks Platform

&

Querying Tables and Views with Apache Spark SQL

2025

Ground Rules

1. Choose a quiet place to attend the class and please concentrate during the lecture

2. Put your questions in Padlet (not Teams' chat box) and I will review them in the due time (Padlet link is in Bb, week 5, Lecture folder)

3. We will have 5 mins break after the first hour of the lecture (please remind me)

4. Jisc code will be shared during the break time

Learning Outcomes

1. To learn what is the Lakehouse concept and its relationship with the Databricks platform
2. To learn databases, tables and views in Databricks
3. To learn how SQL queries can be written and executed in Spark SQL

Section One:
What is Lakehouse

Modern Data Platforms

Modern data platforms serve to fulfil a range of essential functions including data storage, delivery, governance, and protection. They function as a comprehensive system for:

- Data Repository
- Data Management
- Data Analytics

To accomplish these objectives, various architectures such as data warehouses, data lakes, and lakehouses have been developed. These architectures are founded on distinct designs, structures, and functionalities.

Lakehouse vs other technologies

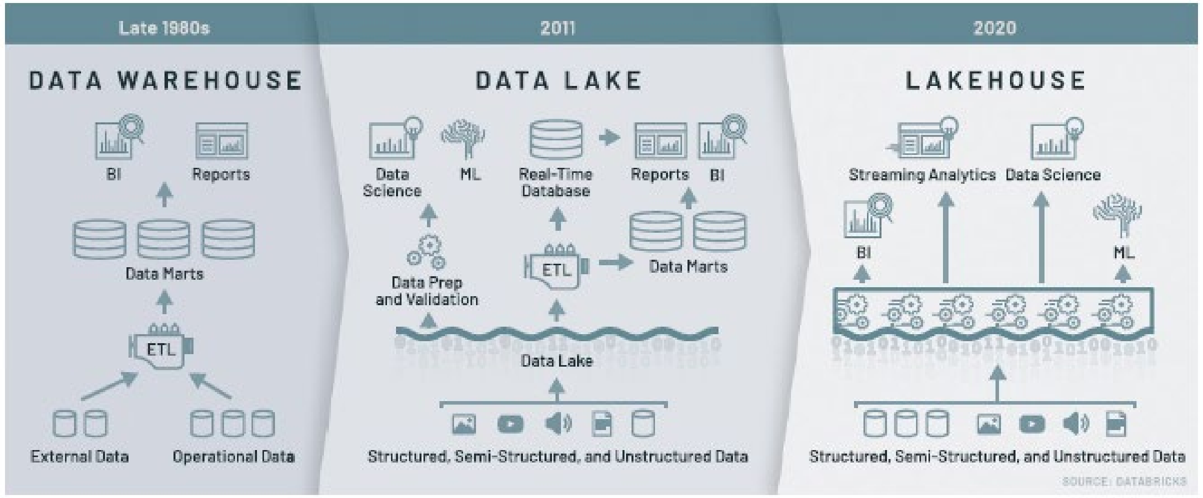


FIGURE 2-1: The differences between a data warehouse, a data lake, and a lakehouse.

This diagram provides a broad overview of three distinct technologies for modern data systems.

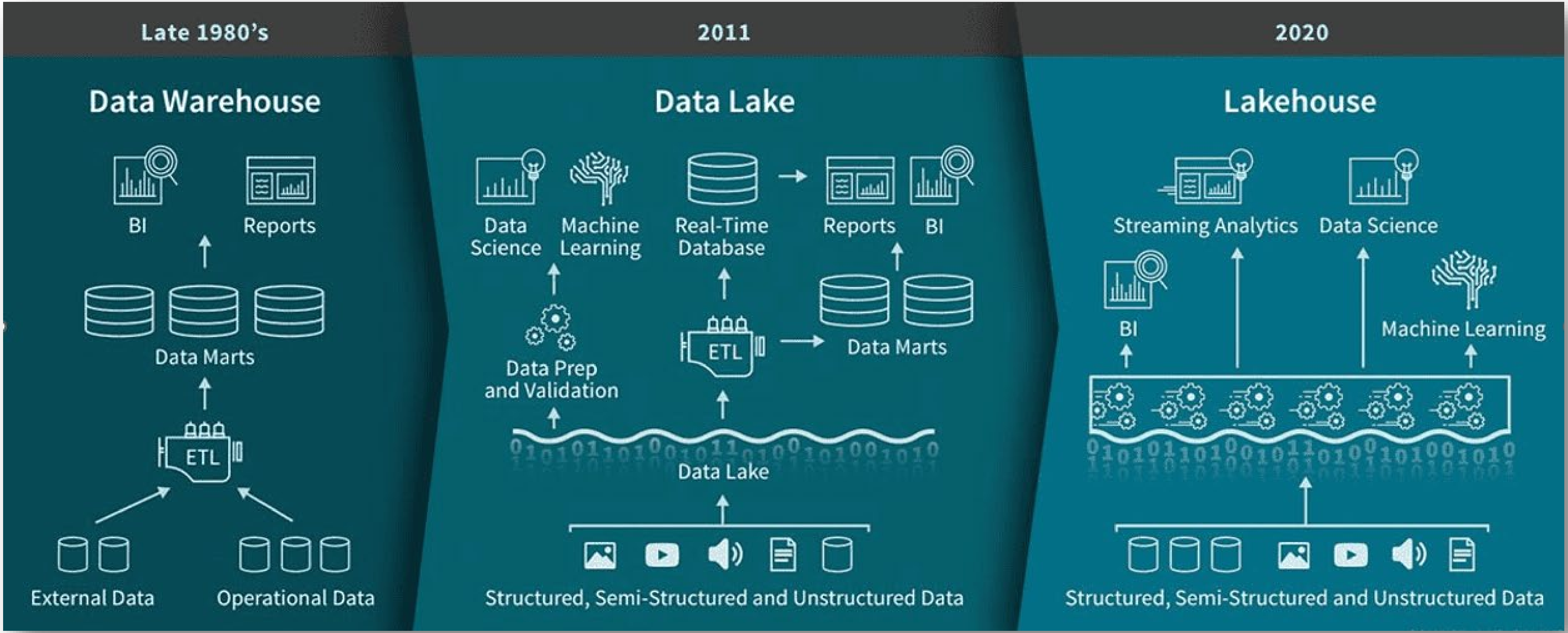
The first two — data warehouses and data lakes — have been leading the industry during different time periods. The lakehouse approach is a brand-new architecture for 2020 and comes with some obvious architectural differences.

Advantages of the Lakehouse compared to alternative technologies:

New systems are beginning to emerge in the industry that address the limitations with and complexity of the two different stacks for:

- Business intelligence (BI) (**Data Warehouses**)
- Machine learning (ML) (**Data Lakes**)

A **Lakehouse** is a new architecture that combines the best of both worlds, Data Warehouses and Data Lakes.



Late 1980's
2011
2020
Data Warehouse
BI
Reports
Data Marts
ETL
External Data
Operational Data
Data Lake
Data Science
Machine Learning
Real-Time Database
Reports
BI
Data Prep and Validation
ETL
Data Marts
Data Lake
Structured, Semi-Structured and Unstructured Data
Lakehouse
Streaming Analytics
Data Science
BI
Machine Learning
Structured, Semi-Structured and Unstructured Data

What is

DELTA LAKE

Delta Lake addresses the data reliability problems that have plagued data lakes, making them data swamps. The open-source storage layer that Delta Lake provides brings improved reliability to data lakes.

Figure 4-1 runs Delta Lake on top of your existing data lake and is fully compatible with Apache Spark APIs.

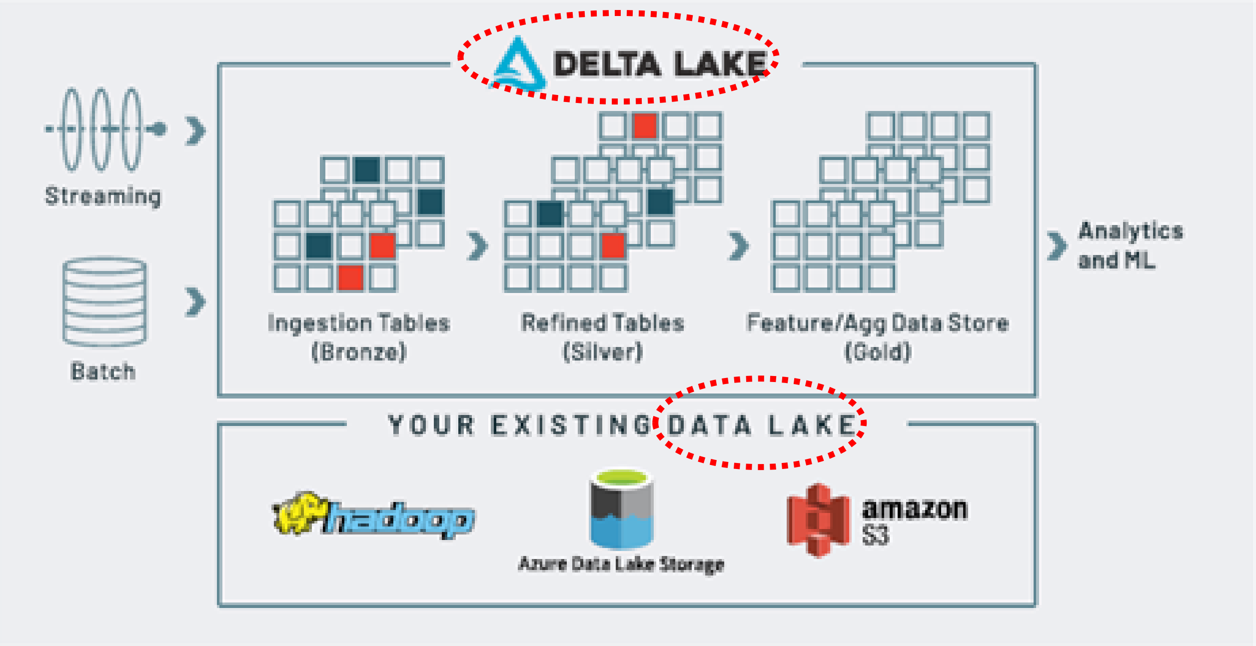


FIGURE 4-1: Delta Lake is an open-source storage layer that brings improved reliability to the lakehouse.

What is Lakehouse

The Lakehouse concept in Databricks is like having a centralised hub for all your data needs. Imagine your data as a big lake, with all sorts of information floating around. The Lakehouse brings together various data sources, like structured and unstructured data, into a single place. Here's the breakdown:

1. **Lake**: The “lake" part, is where all your data resides. It's like the central lake where all the information is stored, regardless of its type or format.

2. **House**: The "house" part, acts as the home or hub for your data lake. It provides tools and infrastructure for managing, processing, and analysing the data effectively.

Why Lakehouse?

Solving Problems with a Lakehouse

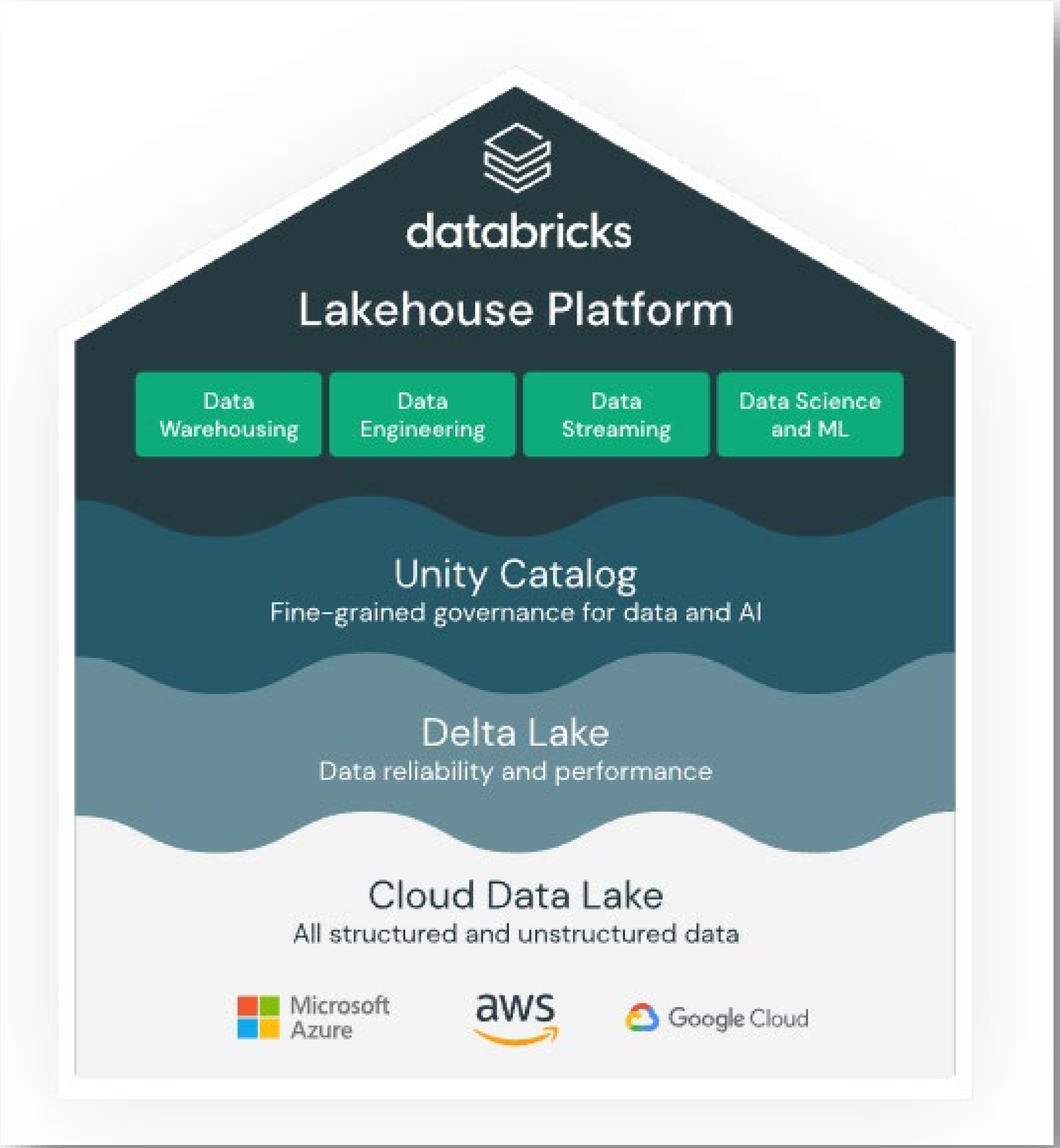
A lakehouse enables business analytics and ML at a massive scale. The challenges that can be overcome with a lakehouse approach are several:

- **Unifying data teams:** One of the biggest benefits of a lakehouse is that it unifies all your data teams — data engineers, data scientists, and analysts — on one architecture.
- **Breaking data silos:** A lakehouse approach facilitates breaking data silos by providing a complete and firm copy of all your data in a centralized location. This enables everyone in your organization to access and manage both structured and unstructured data.
- **Preventing data from becoming stale:** In a continuous manner, the lakehouse approach can process batch and streaming data, updating tables and dashboards in near real time so your data is always generating value, staying updated, and never becoming stale.
- **Reducing the risk of vendor lock-in:** The lakehouse approach uses open formats and open standards that allow your data to be stored independent of the tools you currently use to process it, making it easy at any time to move your data to a different vendor or technology.

How lakehouse has been built in Databricks?

Databricks introduced the term "**Lakehouse**" in 2020 for its **Delta Lake** software. Delta Lake is an open-source project aimed at bringing reliability to **data lakes**.

Lakehouse is a concept that Databricks company has introduced and implemented in the Databricks platform. Since it is a opensource concept others can make their Lakehouse if they want !!



So, What is a Lakehouse?

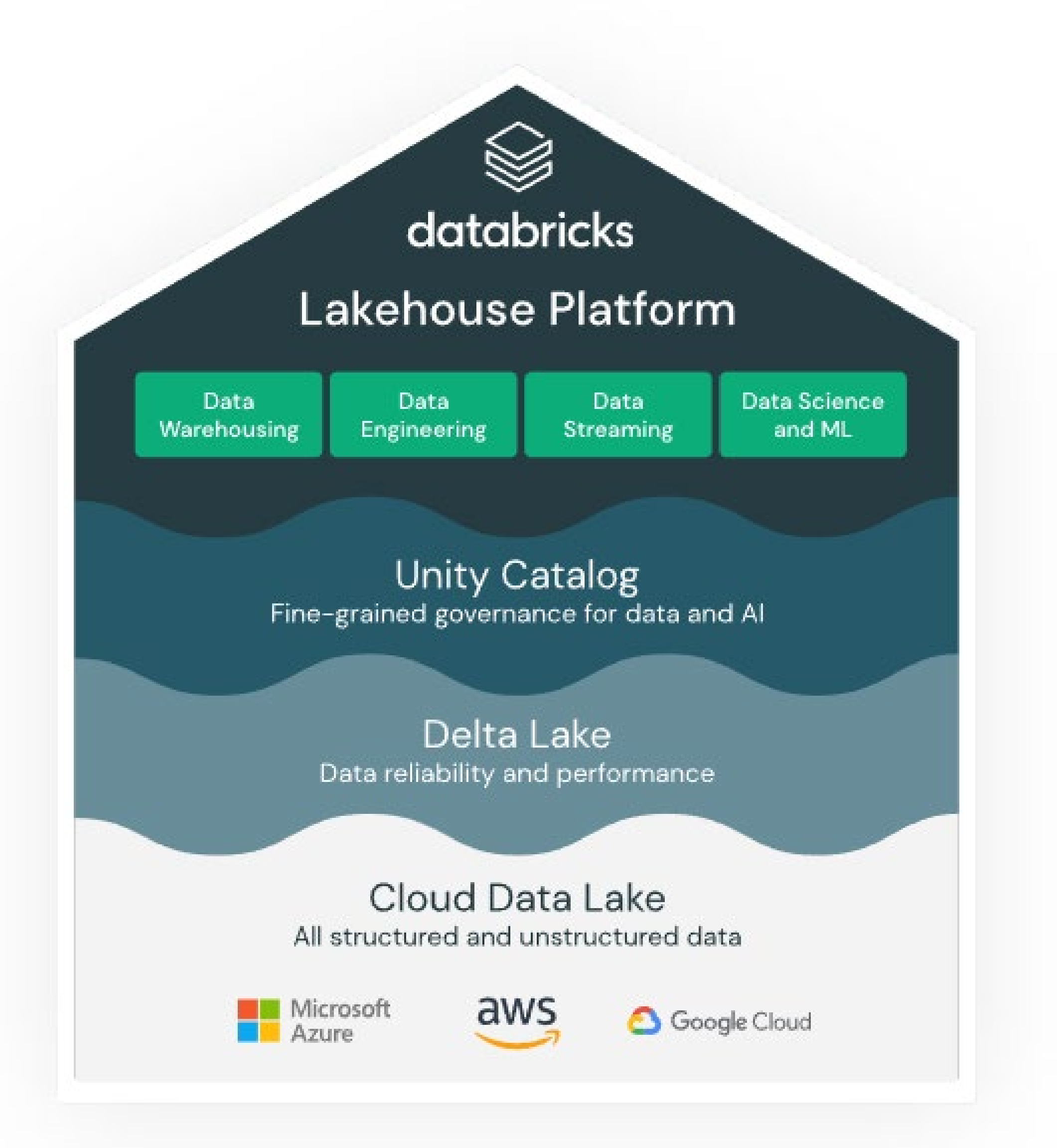
A Lakehouse is a full data management system that:

1. Uses a **Data Lake** for storage (cheap, scalable).

2. Uses **Delta Lake** for structure & reliability (ACID transactions, schema enforcement).

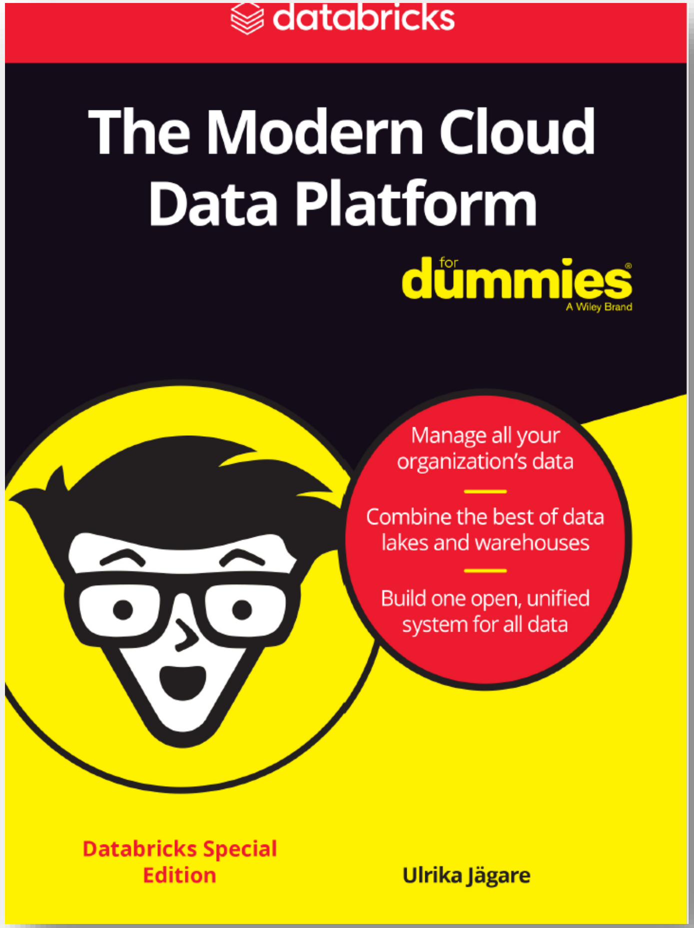
3. Provides **SQL, BI** support (like traditional warehouses).

4. Enables advanced **AI & ML** workloads (unlike traditional warehouses).



Read more about Databricks Lakehouse

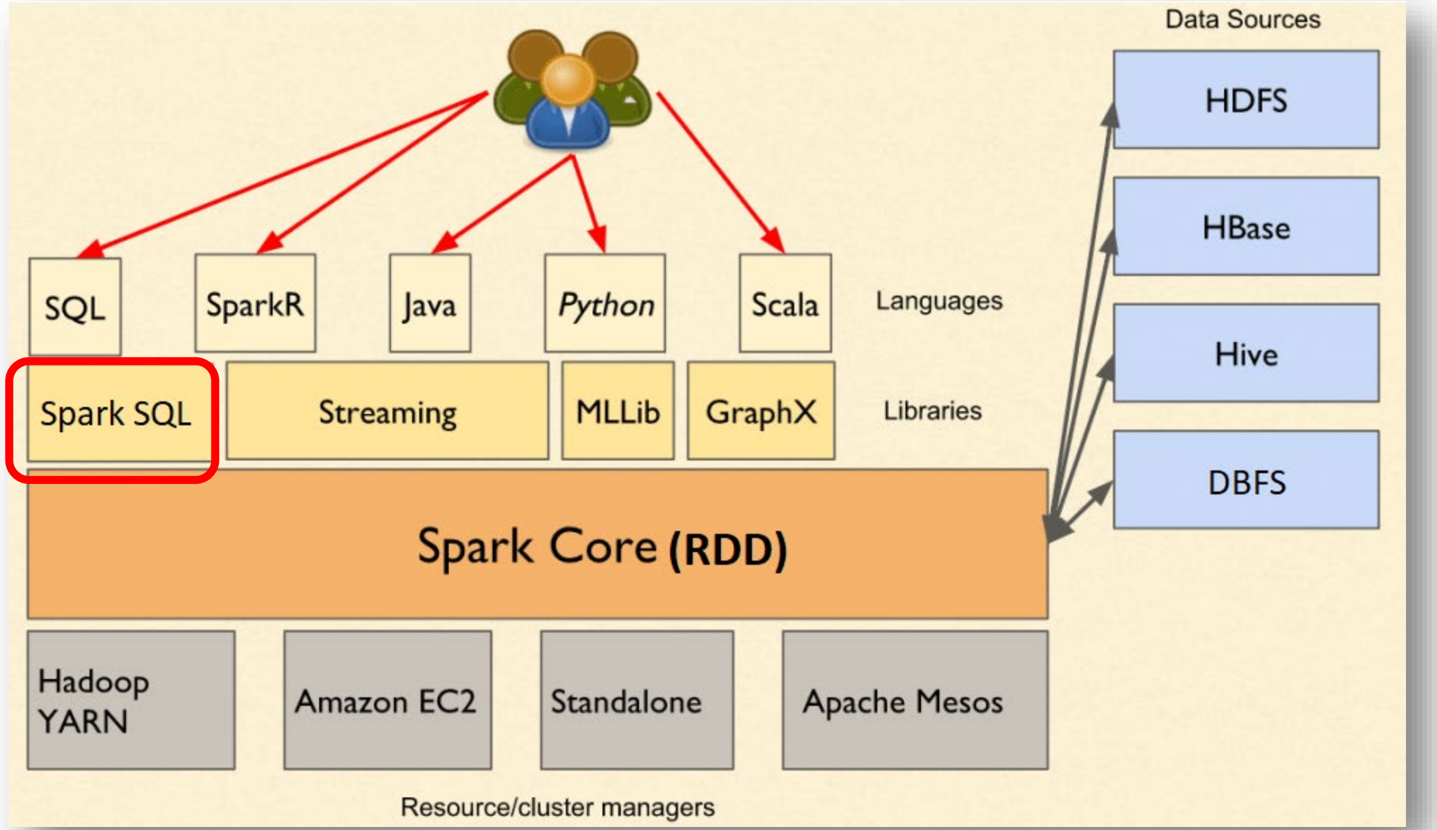
The book is in the Blackboard, Reading List and Resources



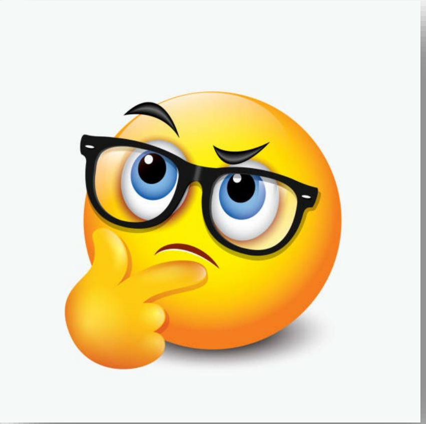
Section Two:
Spark SQL & Databricks

Apache Spark SQL

Today's lecture is mostly about this library



SQL has databases, tables and views but how accommodate them in this structure?



SQL Databases & Tables & Views

1. **SQL Databases**: Databases are like file cabinets in an office. This is like a container that holds all your data. It's where everything is stored.

2. **SQL Tables**: Tables are like the drawers in the file cabinet. They organize your data into neat, structured rows and columns. Each table typically represents a different type of information, like a table for employees, another for customers, etc.

3. **SQL Views**: Views are like temporary arrangement of copied papers on the desk for a particular task, without moving them from the file cabinet. Views are like customised perspectives on the data stored in the tables. Instead of physically rearranging the data, a view presents it in a particular way, like a virtual table. It's useful for simplifying complex queries or providing specific subsets of data without altering the original tables.

Data objects in the Databricks Lakehouse

The Databricks Lakehouse organises data stored with **Delta Lake** in cloud object storage with familiar relations like **database**, **tables**, and **views**.

In the workshop you will work with databases, tables, views

https://docs.databricks.com/lakehouse/

Different Data Types and Spark SQL

While Spark SQL excels at working with structured data in Databricks, it can also handle semi structured and unstructured data like images to a certain extent. Here's how it works:

Spark SQL and Unstructured Data (e.g. images):

- **Limited capabilities:** Spark SQL doesn't natively understand how to directly interpret or analyse images or other unstructured data types.
- **Accessing data:** You can use Spark SQL to access metadata associated with images like file names, timestamps, and tags stored in structured formats.
- **Basic transformations:** Spark SQL can perform basic transformations on image filenames (e.g., filtering by date) or tags (e.g., filtering by categories).

Different Data Types and Spark SQL

Spark SQL can act as a gateway to access and manage basic aspects of unstructured data like images, but for advanced processing and analysis, you need specialised libraries and tools within Databricks like TensorFlow, OpenCV, MLflow. Remember, combining different tools leverages the strengths of each for a comprehensive data analysis pipeline.

SCHOOL OF SCIENCE, ENGINEERING & ENVIRONMENT

Different Data Types and Spark SQL

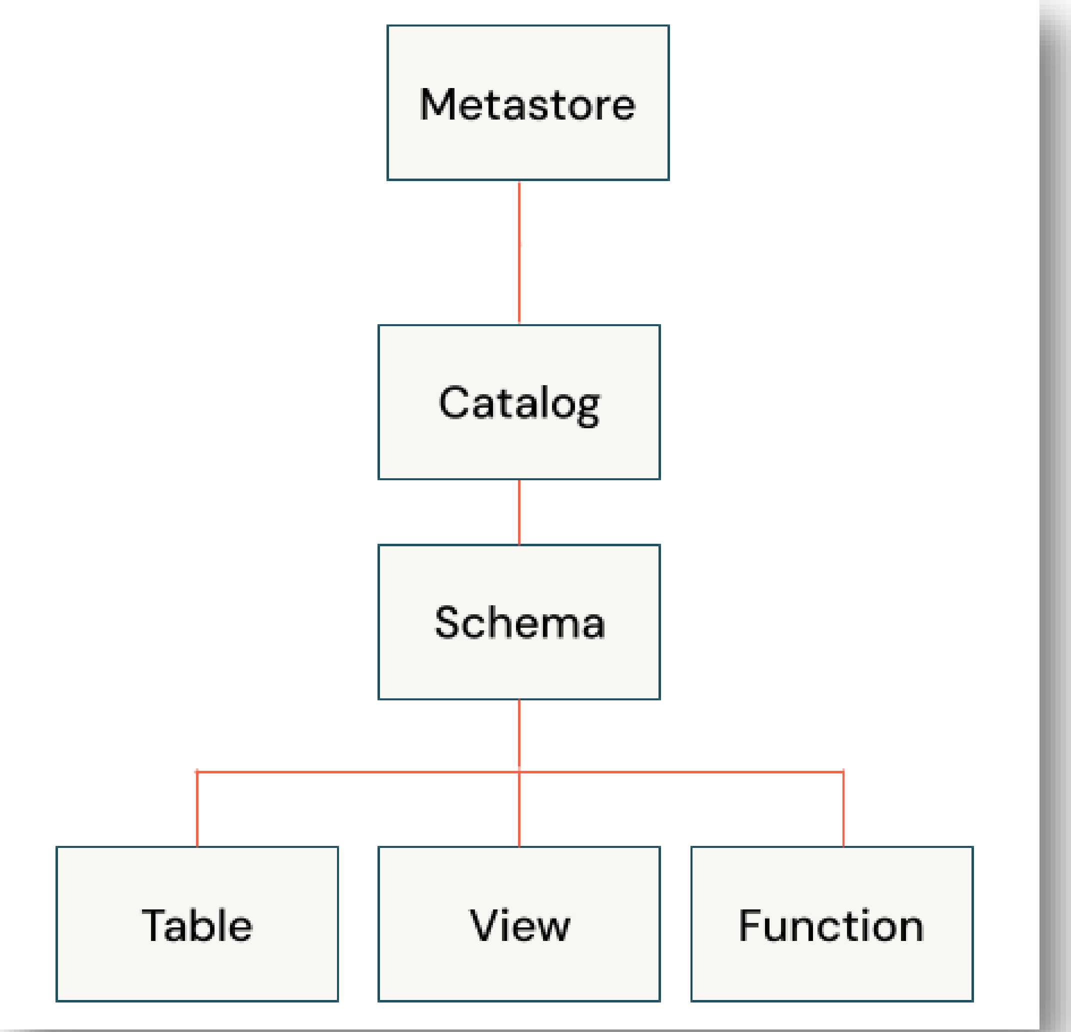
Spark SQL can act as a gateway to access and manage basic aspects of unstructured data like images, but for advanced processing and analysis, you need specialised libraries and tools within Databricks like TensorFlow, OpenCV, MLflow. Remember, combining different tools leverages the strengths of each for a comprehensive data analysis pipeline.

SCHOOL OF SCIENCE, ENGINEERING & ENVIRONMENT

Data objects in the Databricks Lakehouse

The Databricks Lakehouse architecture combines data stored with the Delta Lake protocol in cloud object storage with metadata registered to a metastore. There are five primary objects in the Databricks Lakehouse:

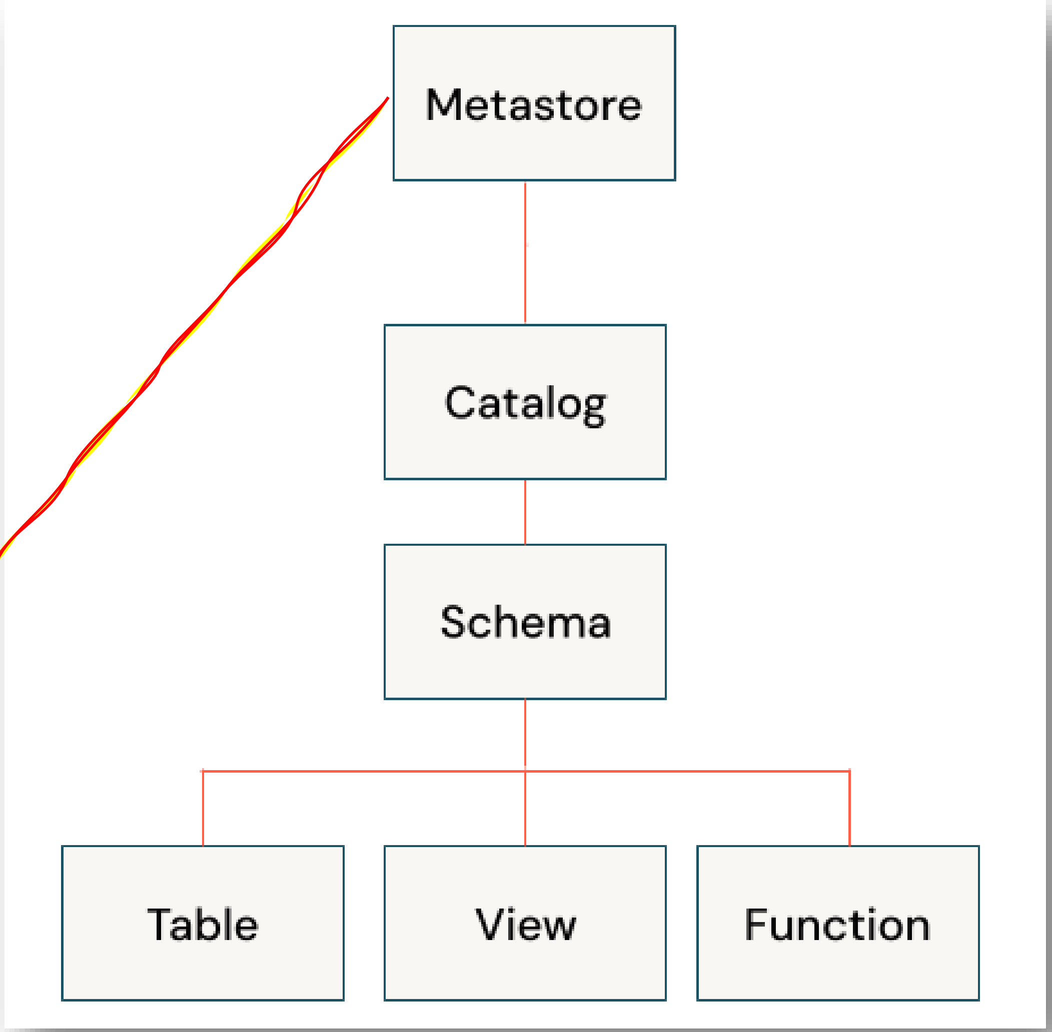
- **Catalog**: a grouping of databases.
- **Schema or Database:** a grouping of objects in a catalog. Databases contain tables, views, and functions.
- **Table**: a collection of rows and columns stored as data files in schema/database.
- **View**: a saved query typically against one or more tables.
- **Function**: saved logic that returns a scalar value or set of rows.



https://docs.databricks.com/lakehouse/

Data objects in the Databricks Lakehouse

But, what is a Metastore?



https://docs.databricks.com/lakehouse/

What is a metastore?

Imagine you have a huge library, but instead of books, it's filled with different kinds of data files. The **metastore** is like the **library's catalog system**. It doesn't hold the actual books (or data) but keeps a detailed record of where every book is located, what it's about, and how it's organized.

The **metastore** contains all the **metadata** that defines data objects in the lakehouse. Databricks provides the following metastore options:

- **Hive metastore**: Databricks stores all the metadata for the **built-in Hive metastore** as a managed service. An instance of the metastore deploys to each cluster and securely accesses metadata from a central repository for each customer workspace.
- **External metastore**: you can also bring your own metastore to Databricks.
- **Unity Catalog**: you can create a metastore to store and share metadata across multiple Databricks workspaces. Unity Catalog is managed at the account level.

Metastore contains metadata not data!!

https://docs.databricks.com/lakehouse/

Databases and tables on Databricks

- Two types of tables: global and local
  - Global: available across all clusters, registered to Hive metastore
  - Local: not accessible from other clusters, not registered in Hive metastore. Also known as temporary view.

Location of the metastore:

```
spark.conf.get("spark.sql.warehouse.dir")
```

Cmd 1

```
spark.conf.get("spark.sql.warehouse.dir")
```

```
Out[1]: 'dbfs:/user/hive/warehouse'
```

Command took 0.14 seconds

Cmd 2

```
dbutils.fs.ls("/user/hive/warehouse")
```

```
Out[2]: [FileInfo(path='dbfs:/user/hive/warehouse/bdtt_db.db/', name='bdtt_db.db/', size=0, modificationTime=0),
 FileInfo(path='dbfs:/user/hive/warehouse/webtable/', name='webtable/', size=0, modificationTime=0)]
```

Command took 0.29 seconds

You will check the metasore in the workshop

Section Three: SQL Queries in Spark

There are two ways to implement SQL queries in Spark

spark.sql API: This approach explicitly uses the spark.sql API for query execution. It offers greater flexibility and control, allowing data processing and integration with other Spark operations like RDDs and Dataframes before querying.

SQL Notebooks: These operate in an environment resembling traditional SQL tools. They offer a simplified interface for writing and running direct SQL queries, primarily focused on querying existing tables and views.

spark.sql API:

Cmd 1

Python

```
myDF = spark.sql("SELECT * FROM people WHERE pcode = 94020")
```

SQL Notebooks:

Cmd 2

SQL

```
%sql
SELECT * FROM people WHERE pcode = 94020
```

spark.sql Queries

Spark SQL Queries

- You can query data in Spark SQL using SQL commands
  - Similar to queries in a relational database
  - Spark SQL includes a native SQL parser
- Spark SQL in Databricks is compatible with Apache Hive
- You can query Hive tables or DataFrame/Dataset views
- Spark SQL queries are particularly useful for
  - Developers or analysts who are comfortable with SQL
  - Doing ad hoc analysis
- Use the `spark.sql` function to execute a SQL query on a table
  - Returns a DataFrame

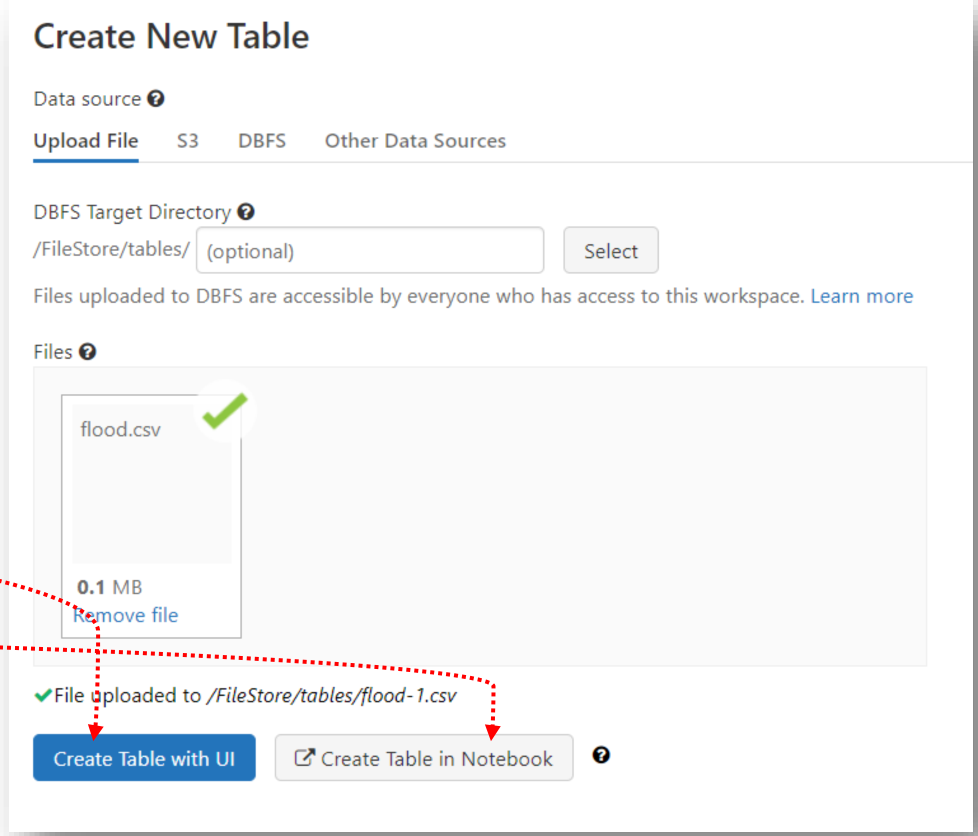
Creating a table on Databricks

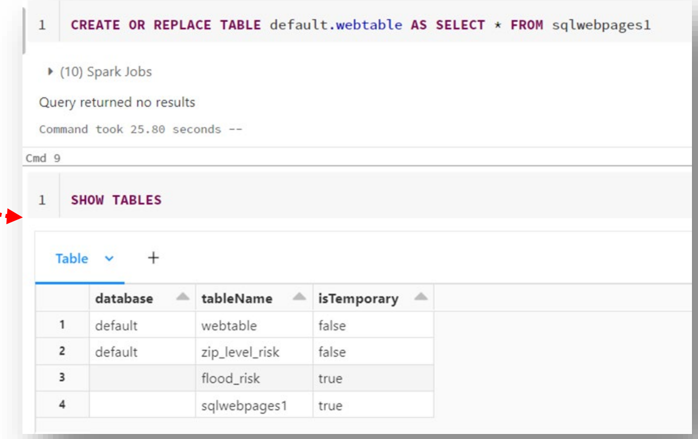
3 ways to create a table

- Using the UI (User Interface)
- In a notebook
- Programmatically

After uploading data in DBFS

- Using the UI (User Interface)
- In a notebook
- Programmatically

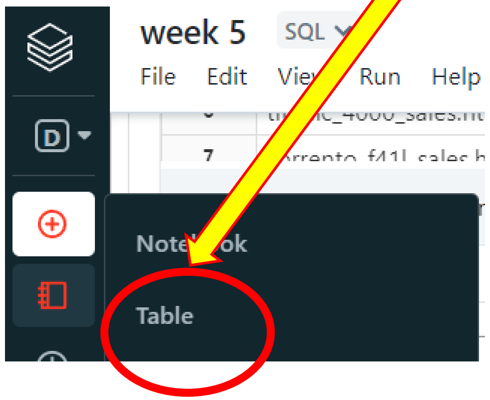




Creating a table on Databricks: using the UI

- Click **Import & Transform Data** from the **Get Started** page.
- **Select** existing **DBFS** or **S3** file(s) or upload a new file.
- **Click Create Table with UI**.
- **Click Preview Table** to view the table.
- In the **Table Name** field, *optionally* override the default table name.
- In the **Create in Database** field, *optionally* override the selected default database.
- In the **File Type** field, *optionally* override the inferred file type.
- For **CSV files**: Select delimiter, select presence of header, decide whether to infer a schema.
- For **JSON files**: Indicate whether the file is multi-line.
- **Click Create Table**.

If dataset already is in DBFS, instead use create table



Creating a table on Databricks: in a notebook

Databricks provides you a quickstart notebook – for S3 or DBFS, there's a **Create Table in Notebook** option you can click.

- You fill in the location of the table and values of the options and a dataframe will be created for you from the data.
- Default: creates a temporary view, so only available in that notebook (and will need to be re-run if cluster is restarted)
- Can make table permanent using `saveAsTable` (in last cell).

Creating a table on Databricks: in a notebook

Create New Table

Data source

Upload File S3 DBFS Other Data Sources

DBFS Target Directory

/FileStore/tables/ (optional) Select

Files uploaded to DBFS are accessible by everyone who has access to th

Files

flood.csv

0.1 MB

Remove file

File uploaded to */FileStore/tables/flood-1.csv*

Create Table with UI | Create Table in Notebook

Cmd 1

Markdown

Overview

This notebook will show you how to create and query a table or DataFrame that you uploaded to DBFS. DBFS is a Databricks File System that allows you to store data for querying inside of Databricks. This notebook assumes that you have a file already inside of DBFS that you would like to read from.

This notebook is written in **Python** so the default cell type is Python. However, you can use different languages by using the `%LANGUAGE` syntax. Python, Scala, SQL, and R are all supported.

Cmd 2

```
# File location and type
file_location = "/FileStore/tables/flood.csv"
file_type = "csv"

# CSV options
infer_schema = "false"
first_row_is_header = "false"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be ignored.
df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)

display(df)
```

Cmd 3

```
# Create a view or table

temp_table_name = "flood_csv"
```

Creating a table on Databricks: programmatically

A permanent (global) table is created using:

```
dataFrame.write.saveAsTable("<table-name>")
```

A temporary view:

```
dataFrame.createOrReplaceTempView("<table-name>")
```

Also can use SQL to create a table which will be listed in Hive metastore:

```
CREATE TABLE <table-name> ...
```

Example: Spark SQL query

First: create an RDD and then a dataframe from people.txt dataset

Cmd 1

```
peopleRDD = sc.textFile("/FileStore/tables/people.txt")
peopleRDD = peopleRDD.map(lambda line: line.split(","))

peopleDF = peopleRDD.toDF(["pcode","first name","last name","age"])

#peopleRDD.take(5)
peopleDF.show()
```

(5) Spark Jobs

peopleDF: pyspark.sql.dataframe.DataFrame = [pcode: string, first name: string ... 2 more fields]

```
+------+----------+---------+---+
| pcode|first name|last name|age|
+------+----------+---------+---+
|02134 |    Hopper |    Grace | 52|
|94020 |    Turing |     Alan | 32|
|94020 |  Lovelace |      Ada | 28|
|87501 |   Babbage |  Charles | 49|
|02134 |     Wirth |  Niklaus | 48|
+------+----------+---------+---+
```

Example: spark.sql query

Second: create a temporary view in SQL and execute a query on the view and store the result table on a dataframe (mytable)

Cmd 2

```
peopleDF.createOrReplaceTempView("people")
```

Command took 0.25 seconds

Cmd 3

```
mytable = spark.sql("SELECT * FROM people WHERE pcode = 94020")

mytable.show()
```

(2) Spark Jobs

mytable: pyspark.sql.dataframe.DataFrame = [pcode: string, first name: string ... 2 more fields]

```
+-----+----------+---------+---+
| pcode|first name|last name|age|
+-----+----------+---------+---+
|94020 |   Turing |     Alan | 32|
|94020 | Lovelace |      Ada | 28|
+-----+----------+---------+---+
```

Example: a more complex query with SQL Notebook

Cmd 6

SQL

```
%sql

SELECT MEAN(age) AS mean_age , STDDEV(age)  AS sdev_age FROM people WHERE pcode LIKE '94%'
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [mean_age: double, sdev_age: double]

Table +

	mean_age	sdev_age
1	30	2.8284271247461903

Take average and standard deviation of 2 columns for a subset of postcodes

Pay attention to the cell type, **SQL**. Choose SQL and you will see the **%sql** on top of the cell

SQL queries and DataFrame queries

- SQL queries and DataFrame transformations provide equivalent functionality
- Both are executed as series of transformations
  - Optimized by the Catalyst optimizer
- The following Python examples are equivalent

```
myDF = spark.sql("SELECT * FROM people WHERE pcode = 94020")
```

```
myDF = spark.read.table("people").where("pcode=94020")
```

SQL queries on files

You can query directly from Parquet or JSON files that are not Hive tables

```
spark. \
  sql("SELECT * FROM parquet.`/FileStore/people.parquet`
       WHERE firstName LIKE 'A%' "). \
  show()

+-----+--------+---------+---+
|pcode|lastName|firstName|age|
+-----+--------+---------+---+
|94020|  Turing|     Alan| 32|
|94020|Lovelace|      Ada| 28|
```

SQL queries on views

Creating a view

- DataFrame.createTempView(view-name)
- DataFrame.createOrReplaceTempView(view-name)
- DataFrame.createGlobalTempView(view-name)

SQL queries on views

CreateTempView:

- Creates a new temporary view if the specified name doesn't already exist.
- Throws an error if a temporary view with the same name already exists.

CreateOrReplaceTempView:

- Creates a new temporary view if the specified name doesn't already exist.
- **Replaces** an existing temporary view with the same name with the new definition.
- This means any subsequent queries referencing the replaced view will use the new definition.

In summary:

- Use **CreateTempView** when you want to create a new view and are sure it doesn't exist already.
- Use **CreateOrReplaceTempView** when you want to either create a new view or replace an existing one with the same name.

SQL queries on a view

After defining a DataFrame view, you can query with SQL just as with a table

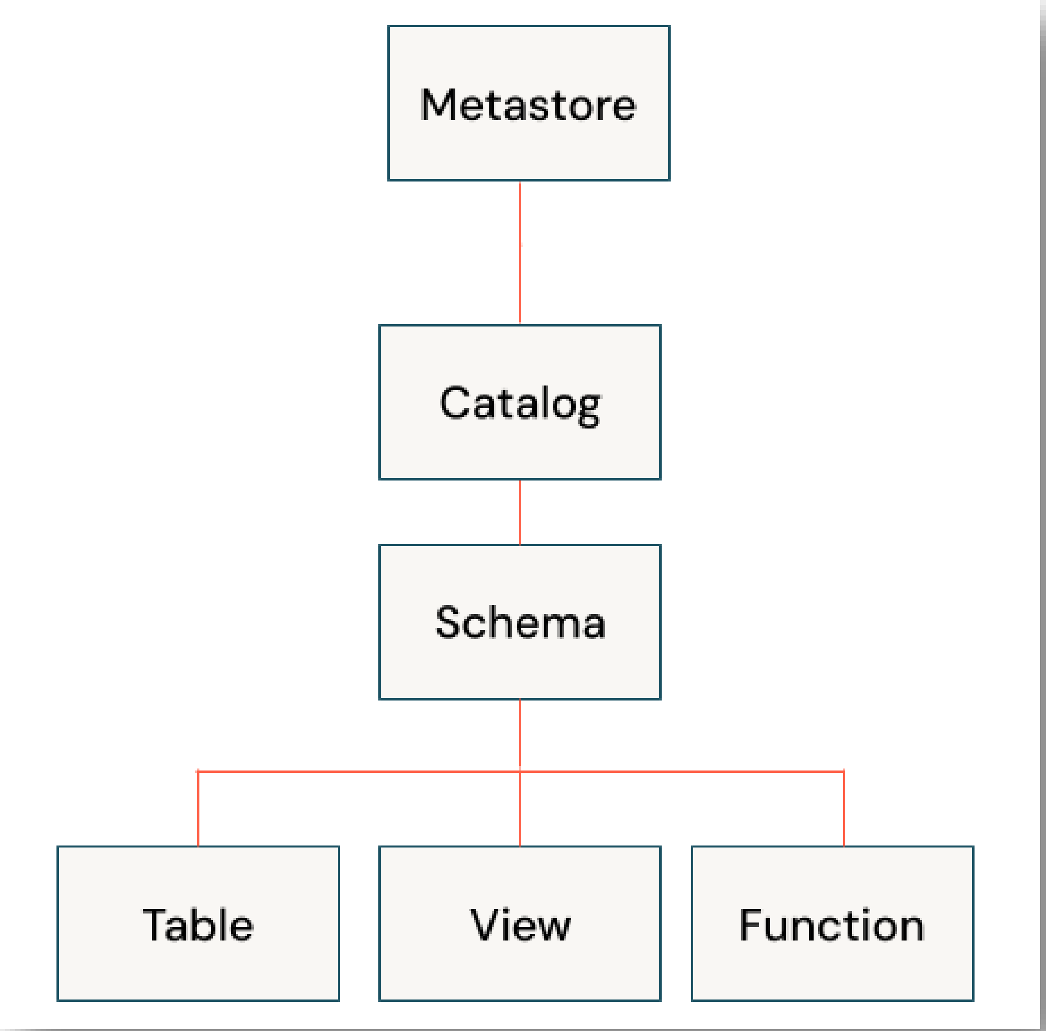
```
spark.read.load("/FileStore/people.parquet"). \
  select("firstName", "lastName"). \
  createTempView("user_names")

spark.sql( \
    "SELECT * FROM user_names WHERE firstName LIKE 'A%'" ). \
  show()
+---------+--------+
|firstName|lastName|
+---------+--------+
|     Alan|  Turing|
|      Ada|Lovelace|
+---------+--------+
```

The Catalog API

A catalog is the highest abstraction (or most granulated) in the Databricks Lakehouse relational model.

Every database will be associated with a catalog.

Catalogs exist as objects within a metastore.



https://docs.databricks.com/lakehouse/data-objects.html#metastore

The Catalog API

- Use the Catalog API to explore tables and manage views
- The entry point for the Catalog API is `spark.catalog`
- Functions include
  - `listDatabases` returns a list of existing databases
  - `setCurrentDatabase(dbname)` sets the current default database for the session
    - Equivalent to the USE statement in SQL
  - `listTables` returns a list of tables and views in the current database
  - `listColumns(tablename)` returns a list of the columns in the specified table or view
  - `dropTempView(viewname)` removes a temporary view

Spark.catalog.listDatabases() ...

The Catalog API

Cmd 15

```
spark.catalog.listTables()
```

(2) Spark Jobs

```
Out[25]: [Table(name='people', catalog=None, namespace=[], description=None, tableType='TEMPORARY', isTemporary=True)]
```

Cmd 17

```
spark.catalog.listDatabases()
```

(2) Spark Jobs

```
Out[26]: [Database(name='default', catalog='spark_catalog', description='Default Hive database', locationUri='dbfs:/user/hive/warehouse')]
```

Name and location of tables and databases

Persistence for DataFrames

Persistence for DataFrames

Persistence in Spark SQL refers to the practice of storing DataFrames in memory or on disk, making them readily accessible for future operations. This can significantly improve performance by avoiding recomputing the same data repeatedly. Here's a breakdown of the key points:

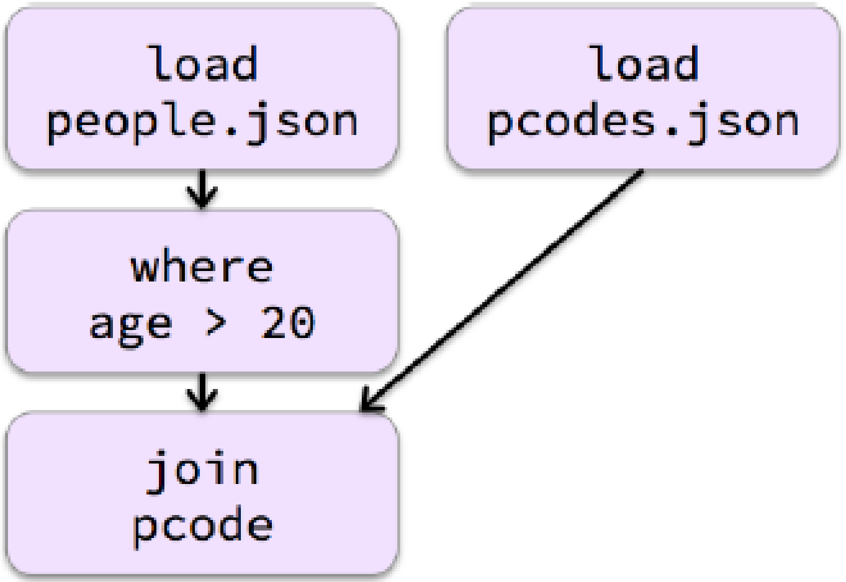
- **Caches:** DataFrames are stored in memory (or disk) for quicker access. Subsequent operations on the same DataFrame don't need to recompute the data from scratch.

- **Improves Performance:** Reusing cached data saves time and resources compared to re-reading or re-computing it.

- **Fault tolerance:** Persisted data can be rebuilt if a node fails, ensuring data integrity.

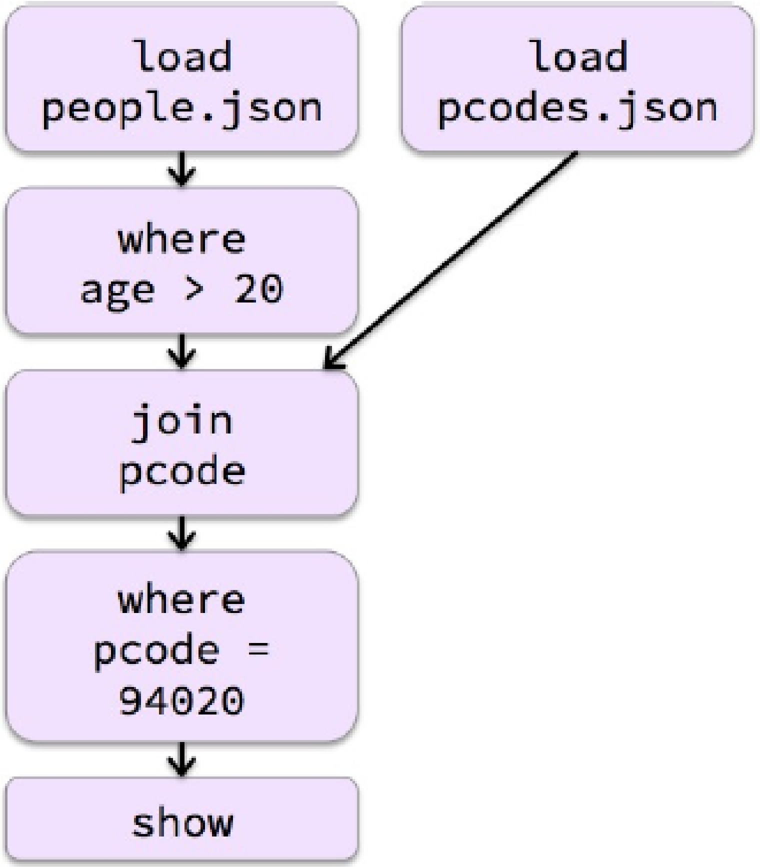
dataframe persistence (part 1)

```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")

pcodesDF = spark.read. \
  json("pcodes.json")

joinedDF = over20DF. \
  join(pcodesDF, "pcode")
```



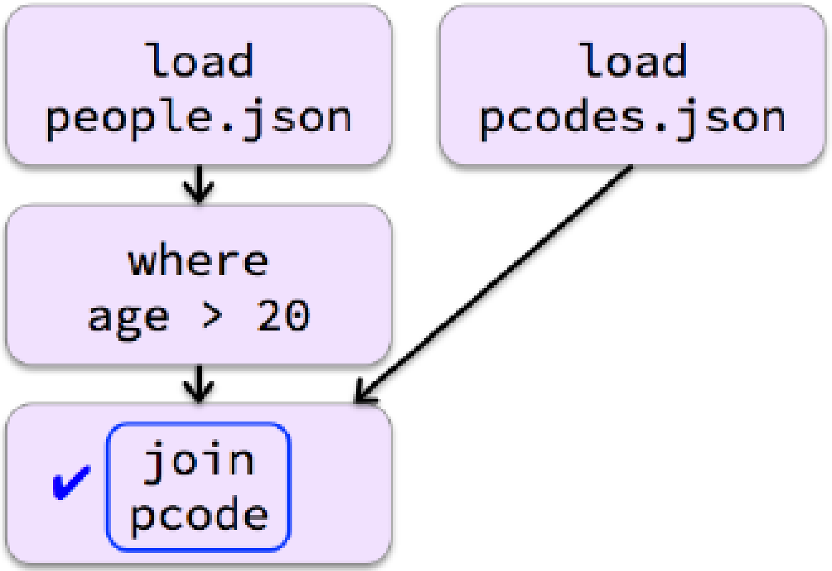
dataframe persistence (part 2)

```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")

pcodesDF = spark.read. \
  json("pcodes.json")

joinedDF = over20DF. \
  join(pcodesDF, "pcode")

joinedDF. \
  where("pcode = 94020"). \
  show()
```



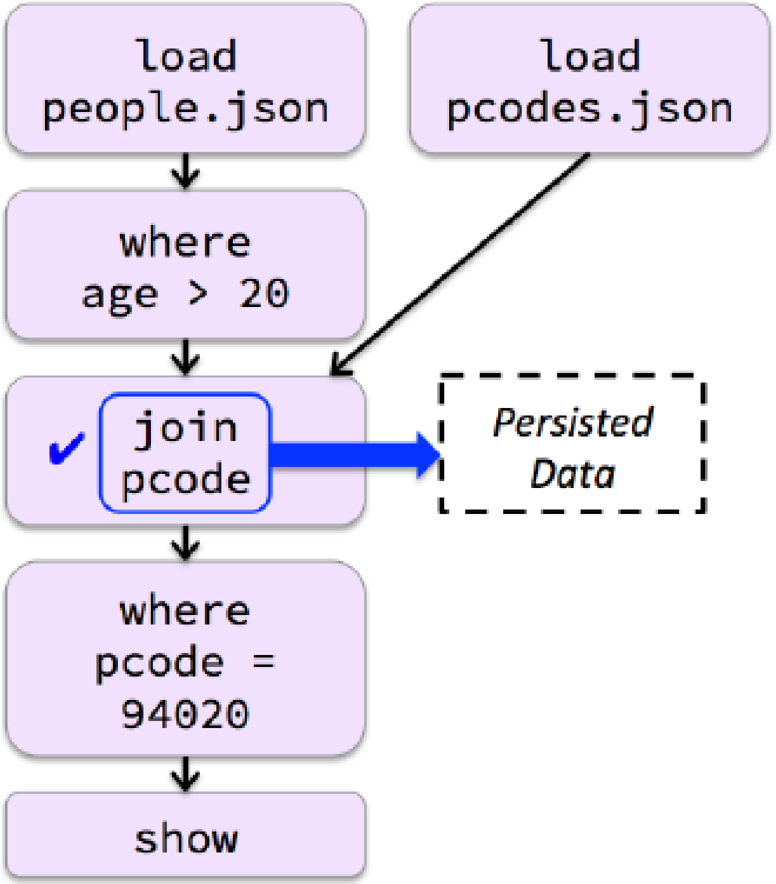
dataframe persistence (part 3)

```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")

pcodesDF = spark.read. \
  json("pcodes.json")

joinedDF = over20DF. \
  join(pcodesDF, "pcode"). \
  persist()
```



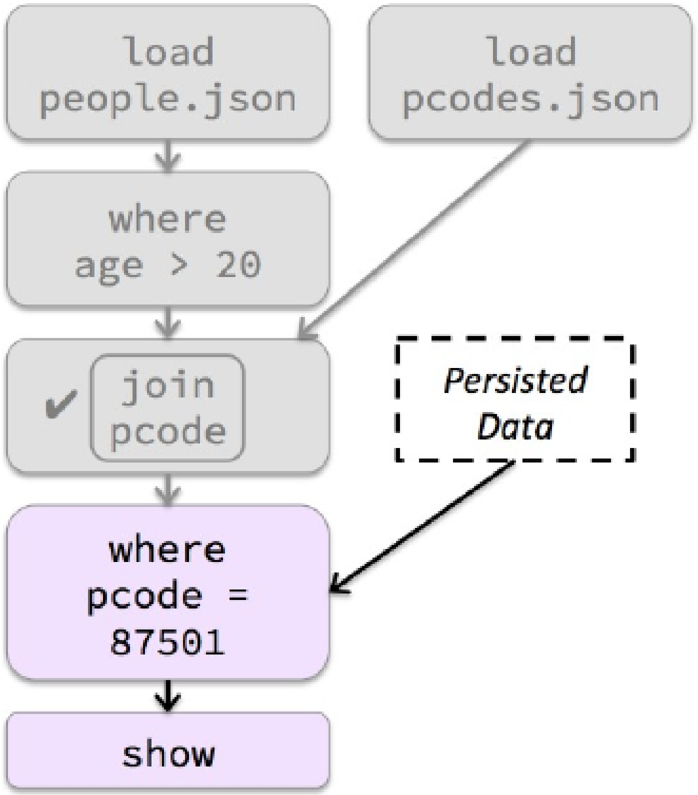
dataframe persistence (part 4)

```
over20DF = spark.read. \
 json("people.json"). \
 where("age > 20")

pcodesDF = spark.read. \
 json("pcodes.json")

joinedDF = over20DF. \
 join(pcodesDF, "pcode"). \
 persist()

joinedDF. \
 where("pcode = 94020"). \
 show()
```



dataframe persistence (part 5)

```
over20DF = spark.read. \
 json("people.json"). \
 where("age > 20")

pcodesDF = spark.read. \
 json("pcodes.json")

joinedDF = over20DF. \
 join(pcodesDF, "pcode"). \
 persist()

joinedDF. \
 where("pcode = 94020"). \
 show()

joinedDF. \
  where("pcode = 87501"). \
  show()
```



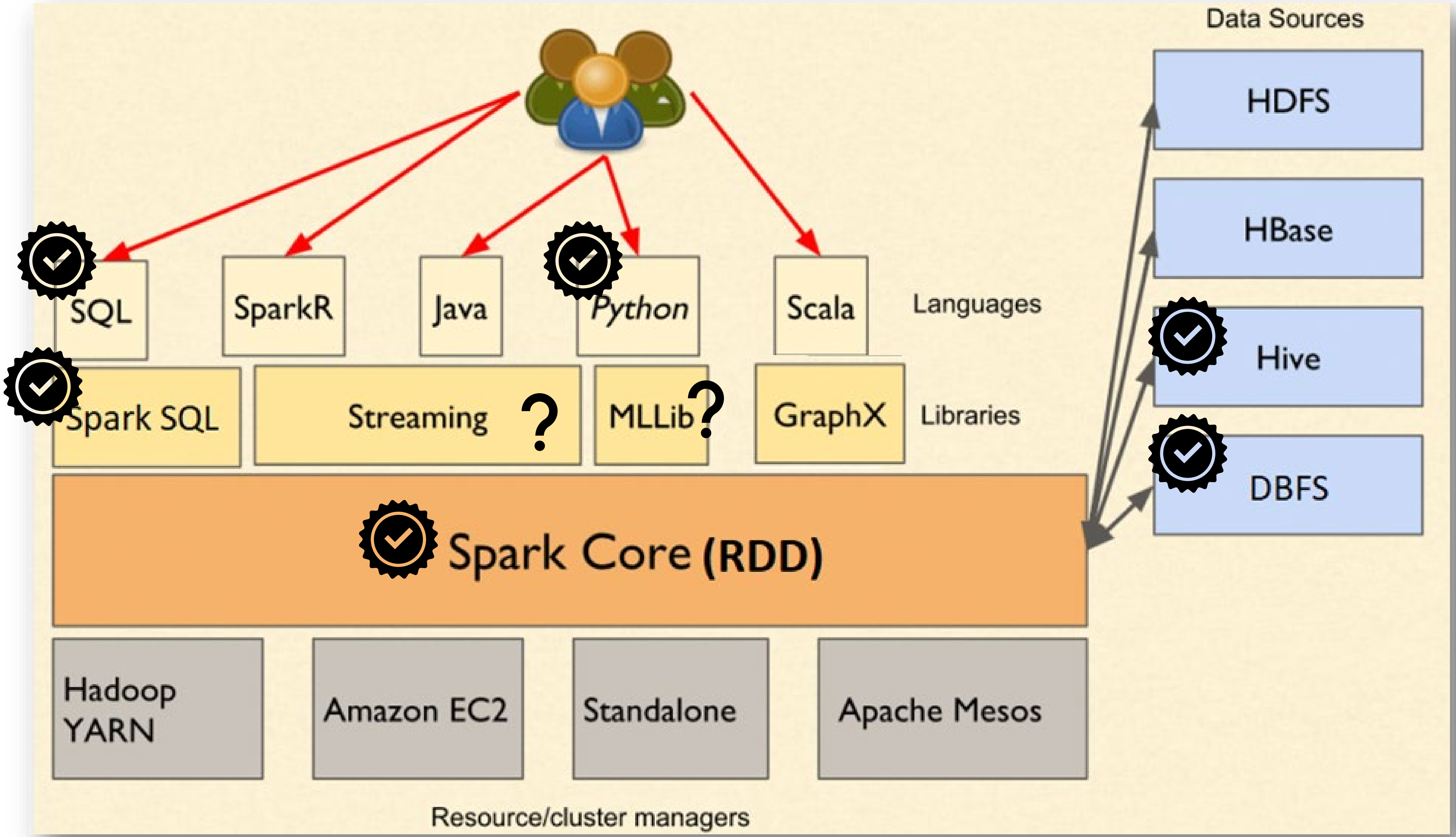
Saving DataFrames

Data in DataFrames can be saved to a data source

- `insertInto` – save to an existing table in a database
- `saveAsParquetFile` – save as a Parquet file (including schema)
- `saveAsTable` – save as a Hive table
- `save` – generic base function

Apache Spark:

what you have learnt
and
What you will learn!


Data Sources
HDFS
HBase
Hive
DBFS
SQL
SparkR
Java
Python
Scala
Languages
Spark SQL
Streaming ?
MLLib?
GraphX
Libraries
Spark Core (RDD)
Hadoop YARN
Amazon EC2
Standalone
Apache Mesos
Resource/cluster managers

A grasp of HDFS and HBase

HDFS (Hadoop Distributed File System):

- **What is it:** A distributed file system designed for storing and managing large datasets across clusters of commodity hardware.
- **In Databricks:** Databricks directly integrates with HDFS, allowing you to:
  - Read and write data from HDFS using Spark DataFrames.
  - Explore and manage HDFS files through the Databricks workspace UI.
  - Configure HDFS access through Databricks clusters and notebooks.

HBase:

- **What is it:** A NoSQL database built on top of HDFS, designed for storing and retrieving large datasets with schema flexibility.
- **In Databricks:** Databricks supports reading and writing data from HBase tables using Spark DataFrames and SQL.

A grasp of GraphX Libray

GraphX is a powerful **graph processing library** built on top of Apache Spark.

It allows you to efficiently handle and analyse data represented as graphs, which are structures consisting of nodes (vertices) and edges (connections) between them.

Applications:

- **Social network analysis:** Understand user connections, identify influencers, and track information flow.
- **Recommendation systems:** Suggest products or services based on user interactions and preferences.
- **Fraud detection:** Analyse financial transactions to identify suspicious patterns.
- **Logistics and routing optimisation:** Find efficient routes for vehicles or goods delivery.
- **Biological network analysis:** Explore relationships between genes, proteins, and other biomolecules.