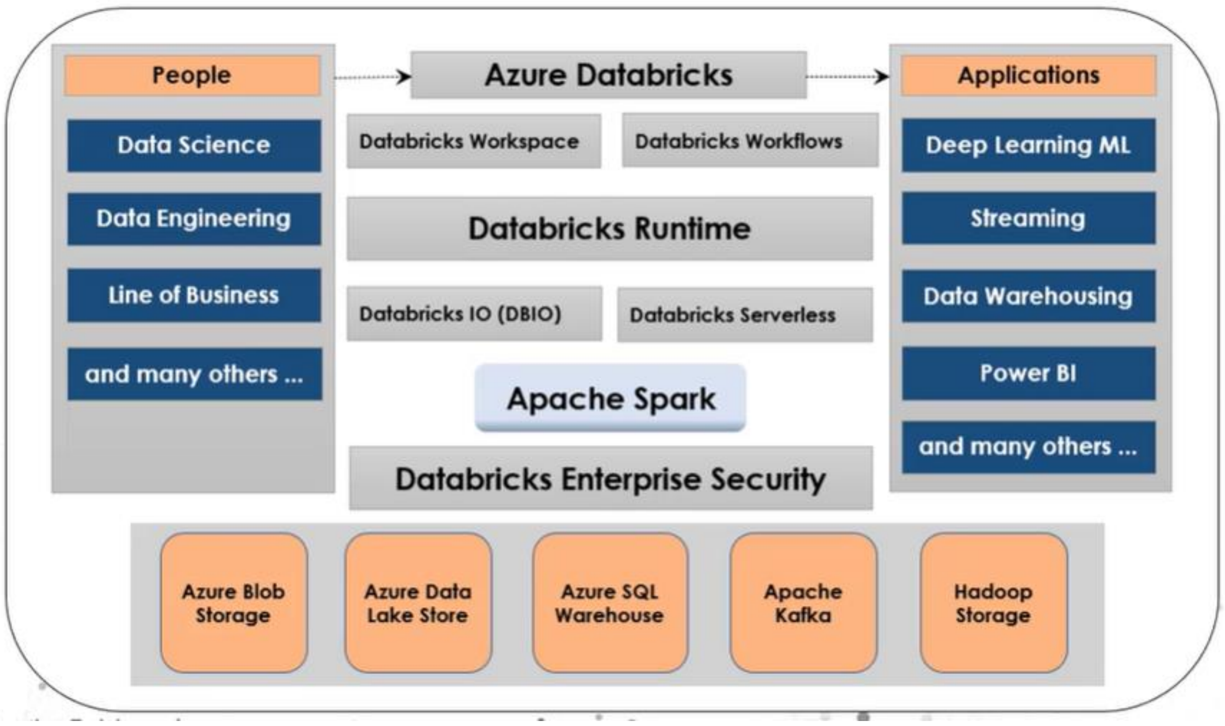
**Data Bricks:**

Data engineer will develop and test and maintains the infrastructure of databases and also in order to improve the quality and efficiency and reliability, we need to use variety of languages on tools like Python, Java, SQL then will provide the data for ready to use for data scientist are responsible for writing machine learning algorithms and statistical methods to develop the predictive models for data business analyst who are responsible for generating different types of reports in order to take a business decisions.

**Infrastructer of data bricks:**



**What is data bricks?**

it is an Apache spark-based analytics platform, which is optimised for cloud services. It helps to set up the spark environment quickly where the data scientists and analyst can work on. And data bricks enterprise security as well and supports azure blob storage , Azure Data lake Store, Azure Synapse analytics (Azure SQL warehouse), Apache Kafka, Hadoop storage

**Data Bricks Runtime?**

It is nothing but set of artifacts and used run on data bricks

**What is data bricks workspace?**

a workspace is the root folder for azure data bricks. Workspace is used to create clusters in a data bricks. workspace is environment to access and organize notebook, libraries and ML experiments.

**Databricks Workflows?**

It helps to build the complex workflows and  pipelines with dependants

**Data bricks IO (DBIO)?**

Data bricks IO is package to provide transactional rights to cloud storage for spark jobs

**Data bricks serverless?**

This provides the features to make use full power of Apache spark

**Cluster:**

Cluster provides a unified platform for running ETL pipelines, streaming analytics and machine learning experiment. There are two types of clusters in order to break namely **All-Purpose cluster and job cluster** and **pool**

**All Purpose cluster:** development puspose and it is interactive cluster is analysing data collaboratively with interactive notebooks,

**job cluster: runs** fast and robust automatically. And sheduleing jobs

Pool is when we want combine multiple clusters up and running.

## What are types of clusters in Databricks?

**Standard Clusters:** These are general-purpose clusters suitable for a wide range of workloads. Standard clusters can be configured with different instance types and sizes to meet specific performance and scalability requirements.

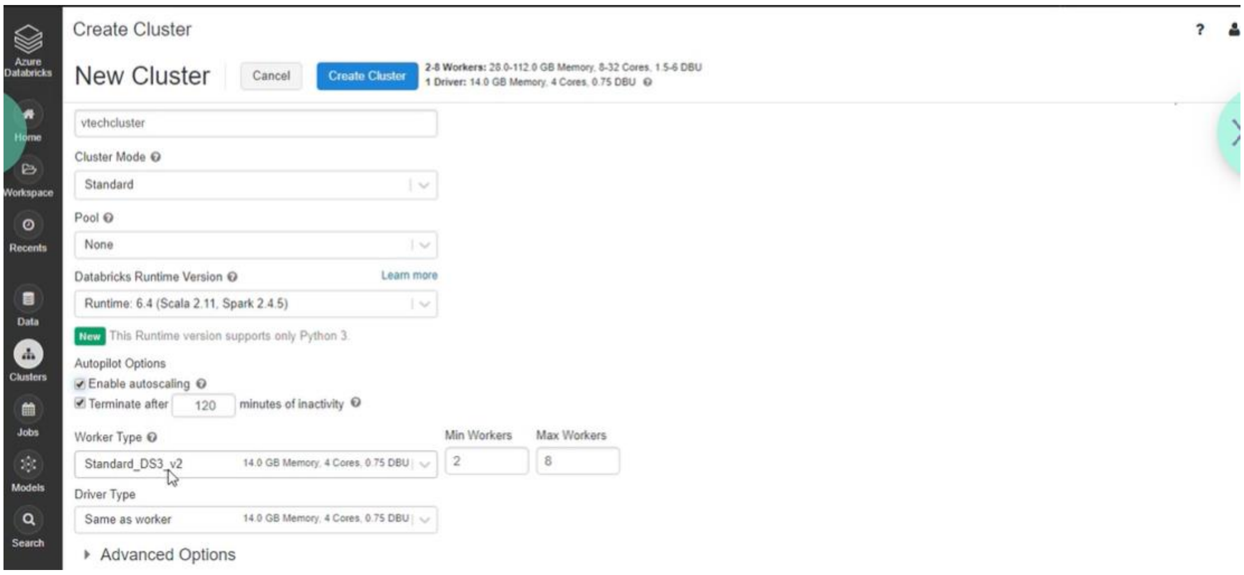
**High Concurrency Clusters:** These clusters are optimized for handling concurrent user queries and workloads in multi-user environments. They provide improved performance and resource isolation to ensure consistent performance even with high levels of concurrency.

**Serverless Pools**: Serverless pools provide on-demand Apache Spark clusters that automatically scale resources based on workload requirements. Users are billed based on the amount of data processed and the resources consumed, offering a flexible and cost-effective option for intermittent or unpredictable workloads.

**Notebook:**

Notebooks Contains renewable code. It is an interface for interacting with azure data bricks

**Creating Cluster**:



New Cluster

2-8 Workers: 28.0-112.0 GB Memory, 8-32 Cores, 1.5-6 DBU
1 Driver: 14.0 GB Memory, 4 Cores, 0.75 DBU

vtechcluster

Cluster Mode: Standard

Pool: None

Databricks Runtime Version: Runtime: 6.4 (Scala 2.11, Spark 2.4.5)

This Runtime version supports only Python 3.

Autopilot Options

- [✓] Enable autoscaling
- [✓] Terminate after 120 minutes of inactivity

Worker Type: Standard_DS3_v2 — 14.0 GB Memory, 4 Cores, 0.75 DBU

Min Workers: 2 Max Workers: 8

Driver Type: Same as worker — 14.0 GB Memory, 4 Cores, 0.75 DBU

**Worker**:

Workers run the spark executors and other services required for proper functioning of the clusters. When you distribute workload with spark, all the distributed processing happens on workers.

**Driver**: driver State maintains information of all notebooks attached to the cluster. It also runs the Apache Spark Master that coordinates with spark executors.

---

**Connection and Read data Blob storage into Databricks:**

```python
storage_account_name = "<storage_account_name>"
container_name = "<container_name>"
access_key = "<access_key>"
dbutils.fs.mount(source=f"wasbs://{container_name}@{storage_account_name}.blob.core.w
```

```
# Read data into a DataFrame
df = spark.read.csv(f"wasbs://{container_name}@{blob_storage_account_name}.blob.core.

# Show the DataFrame
display(df)
```

To create a temporary folder in Blob storage using Python code in Databricks, you can use the `dbutils.fs.mkdirs()` function. Here's how you can do it:

```python
# Define the path of the temporary folder in Blob storage
temp_folder_path = "/mnt/temp_folder"

# Create the temporary folder in Blob storage
dbutils.fs.mkdirs(temp_folder_path)
```

## Connection and read data from Azure SQL into Azure Data bricks:

**Connect to Azure SQL Database:**

In your notebook, use the `%sql` magic command to connect to Azure SQL Database using JDBC URL. Here's an example:

```python
%sql
CREATE TABLE IF NOT EXISTS my_table (
    column1 datatype,
    column2 datatype,
    ...
);
```

**Read Data from Azure SQL Database:**

Once connected, you can use the `%sql` magic command or Spark SQL to execute queries and read data from Azure SQL Database. Here's an example:

```python
%sql
SELECT * FROM my_table;
```

## Connection and read data from ADLS Gen2 into Azure Data Bricks:

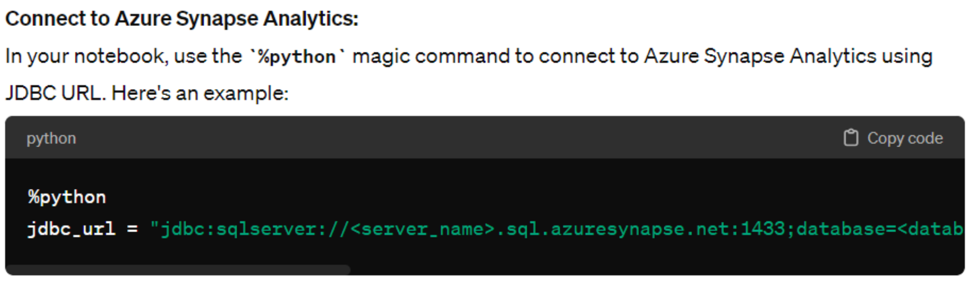
```python
# Define your ADLS Gen2 storage account name, file system name, and client ID
adls_storage_account_name = "<storage_account_name>"
file_system_name = "<file_system_name>"
client_id = "<client_id>"
tenant_id = "<tenant_id>"
client_secret = "<client_secret>"

# Define the path to your file in ADLS Gen2
file_path = f"abfss://{file_system_name}@{adls_storage_account_name}.dfs.core.windows

# Read data into a DataFrame
df = spark.read.csv(file_path, header=True, inferSchema=True)

# Show the DataFrame
df.show()
```

**Connect azure synapse analytics into azure data bricks**:

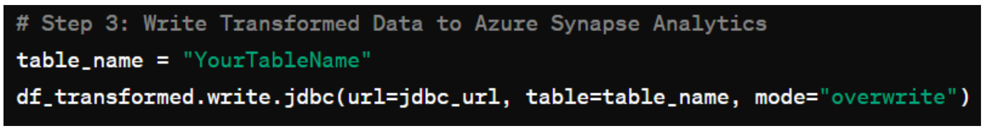


**Connect to Azure Synapse Analytics:**

In your notebook, use the `%python` magic command to connect to Azure Synapse Analytics using JDBC URL. Here's an example:

```python
%python
jdbc_url = "jdbc:sqlserver://<server_name>.sql.azuresynapse.net:1433;database=<datab
```

**load transformed data in azure synapse analytics with azure data bricks by using python:**

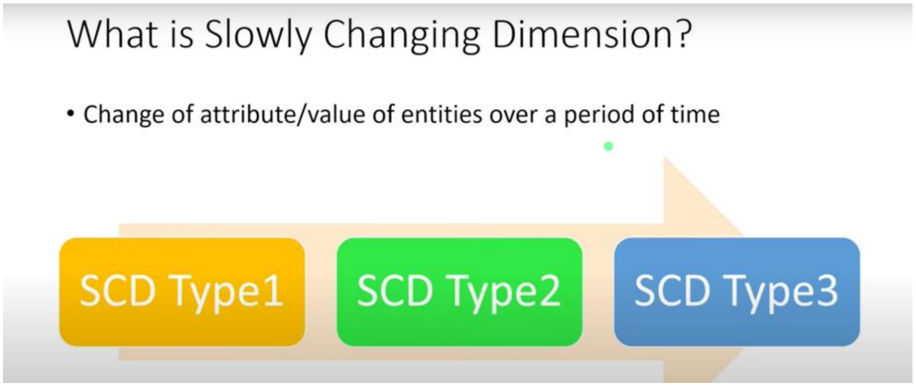


```python
# Step 3: Write Transformed Data to Azure Synapse Analytics
table_name = "YourTableName"
df_transformed.write.jdbc(url=jdbc_url, table=table_name, mode="overwrite")
```

---

**SCD Type : Slowly Changing Dimension**

It is used to change the values / records over period of time



What is Slowly Changing Dimension?

- Change of attribute/value of entities over a period of time

SCD Type1
SCD Type2
SCD Type3

**Scenario:** Customers has address, now address got changed

**SCD Type1:** Here we will update with new address means that the previous records lost

Hence that we will lose the history of records.

**SCD Type2**: We have start date, end date, Flog Key for tracking status of record.

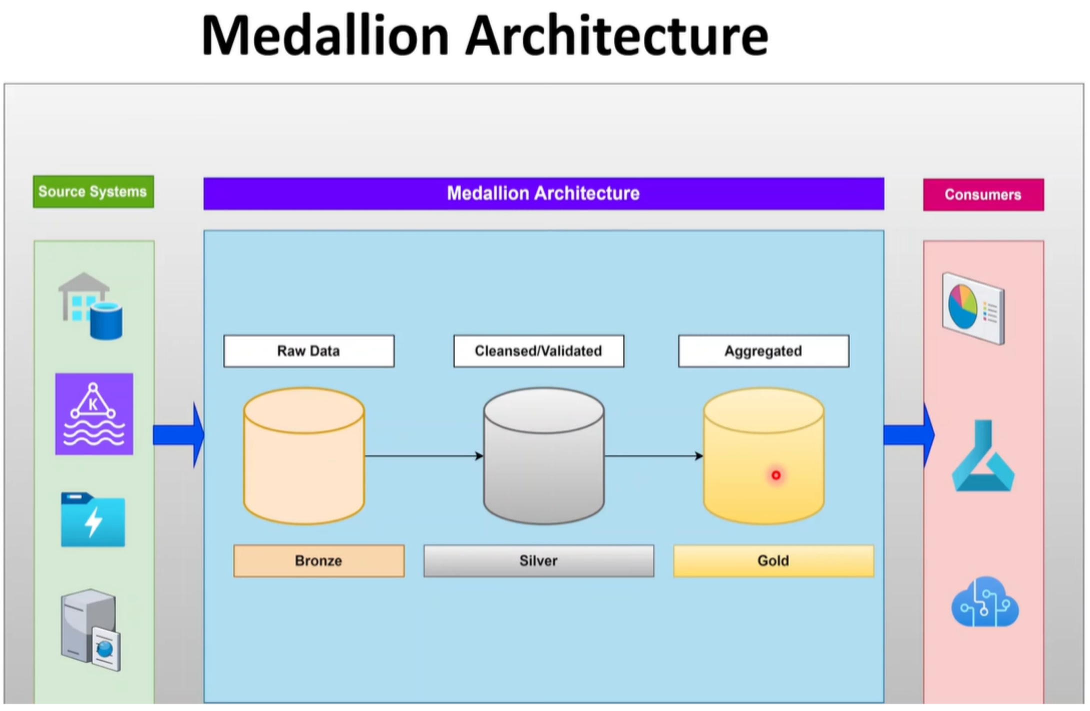
Hence that we maintain both historical and current records at a time.

1. Compare the new data with existing records if matches found, then make it existing records as inactive and insert new records
2. If matches not found then ignore the existing records and insert the new records
3. we can make previous records as inactive and the new records will be inserted and created duplicate records of primary keys.

**SCD Type3:** here we will do instead of use SCDtype2, we make column as current records for new entries and for older records as previous records.

---

Lake house architecture: for example, blob storage we have ….



we follow three-layer architecture

In the context of Databricks and data processing pipelines, the terms "bronze," "silver," and "gold" are often used to represent different stages or layers of data refinement and processing. These layers are part of an architecture pattern called the Delta Lake architecture. Here's how they typically break down:

1**. Bronze Layer:**

- Raw Data Ingestion: The bronze layer represents the raw, ingested data in its original form as it's brought into the data lake. This data is often unprocessed and may contain duplicates, errors, or inconsistencies.

- Schema Enforcement: While the data in the bronze layer is largely raw, schema enforcement is applied to ensure that it adheres to a predefined schema. This helps maintain consistency and prepares the data for further processing.

**2. Silver Layer:**

- Cleaned and Structured Data: In the silver layer, the raw data from the bronze layer undergoes cleaning, validation, and structuring processes. This involves removing duplicates, handling missing values, and transforming the data into a more structured format.

- Quality Checks: Data quality checks are performed in the silver layer to ensure that the data meets certain quality standards. This may involve verifying data completeness, accuracy, and consistency.

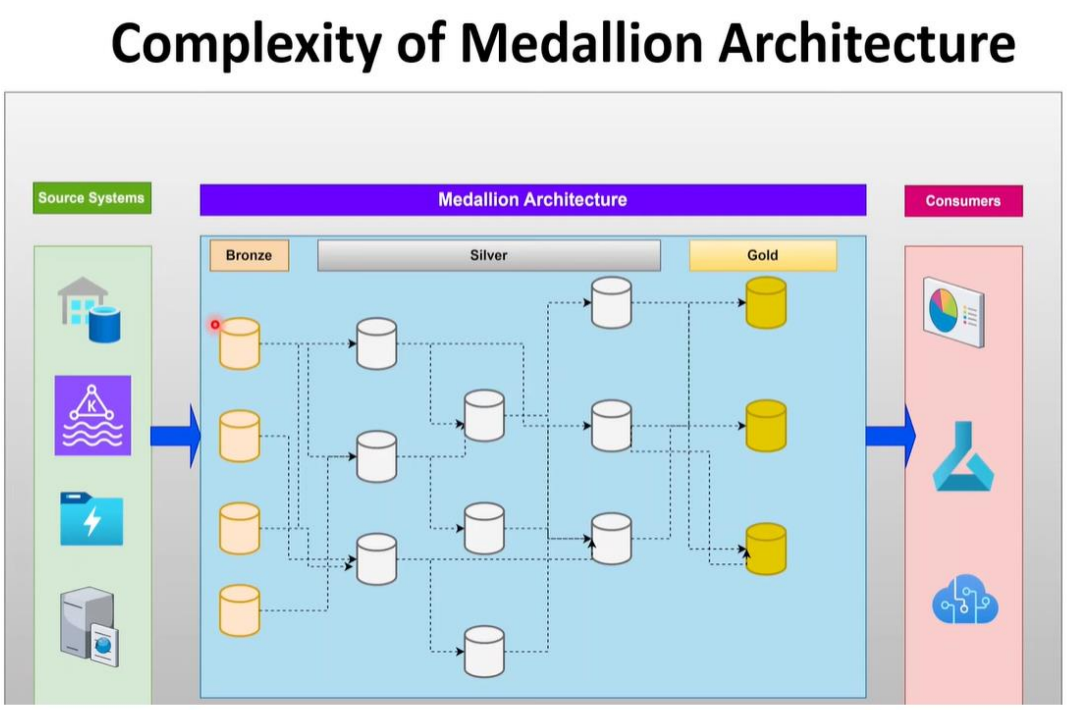
**3. Gold Layer:**

- Curated and Optimized Data: The gold layer represents the refined and optimized data that is ready for consumption by end-users or downstream applications. This data is typically well-structured, validated, and optimized for performance.

- Business Logic and Aggregations: Business logic, aggregations, and calculations are applied to the data in the gold layer to derive insights and support analytical queries.

- Data Governance and Security: Data governance policies, access controls, and security measures are enforced in the gold layer to ensure data integrity, confidentiality, and compliance.

The bronze, silver, and gold layers help organizations manage the end-to-end data lifecycle, from raw ingestion to curated insights. They provide a structured approach to data processing and refinement, enabling efficient data management, analysis, and decision-making. Databricks, with its support for Delta Lake and Apache Spark, provides a powerful platform for implementing these data processing pipelines.



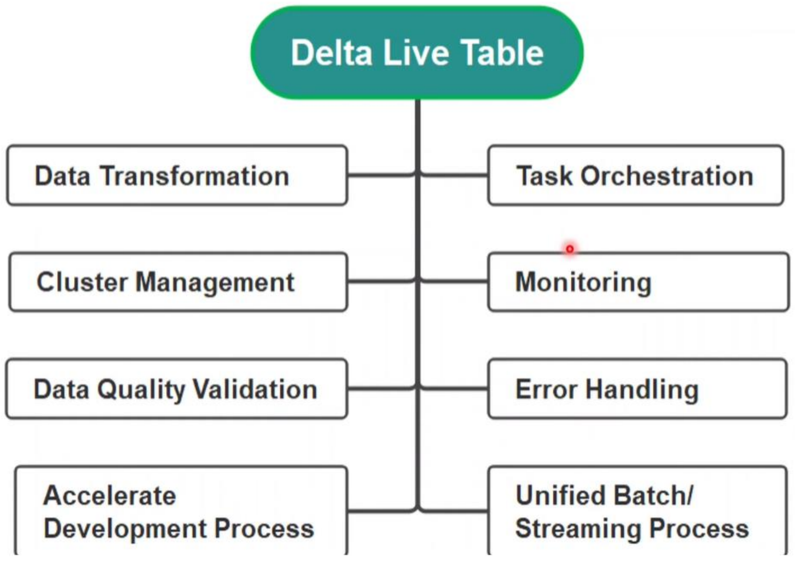
---

**Delta Live Tables:** it is development framework provided by data bricks which simplifies and accelerate the ETL pipelines.

DLT follows declarative approach to build pipelines instead of procedural approach.

DLT framework automatically manage the infrastructure at scale

With help of DLT, engineers can be focus more developing solution rather than infrastructure tools.



**How to Create Delta Live Table?**

Create live tables using Notebooks.

Create DLT pipeline using workflow

Run/Schedule pipeline

Monitor metrics in UI

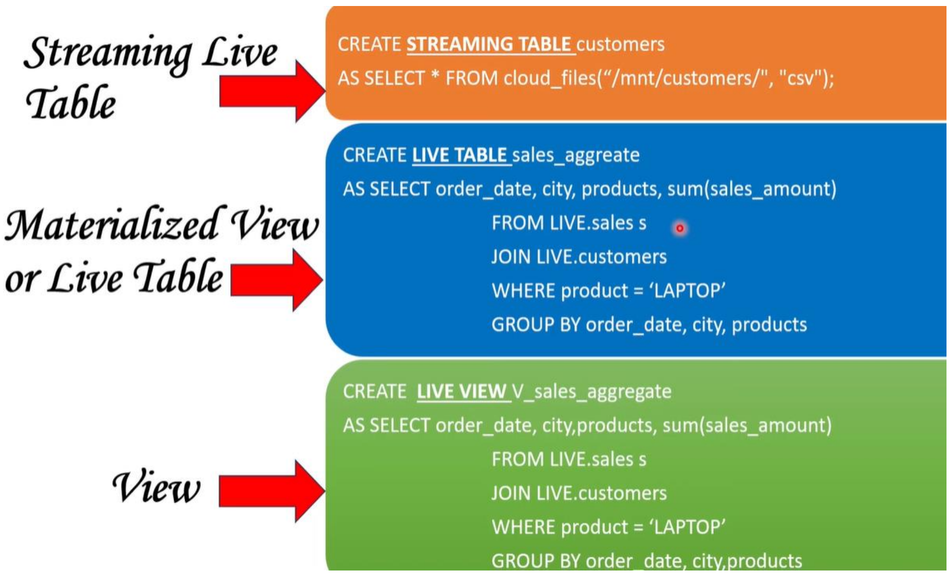
**Different types of Delta Live Tables:**

Streaming Table – bronze layer incrementally data / raw data

Materialized view  or live table – silver and gold layer its store as live table

View Table – Quality checks

**Create table using Spark SQL**

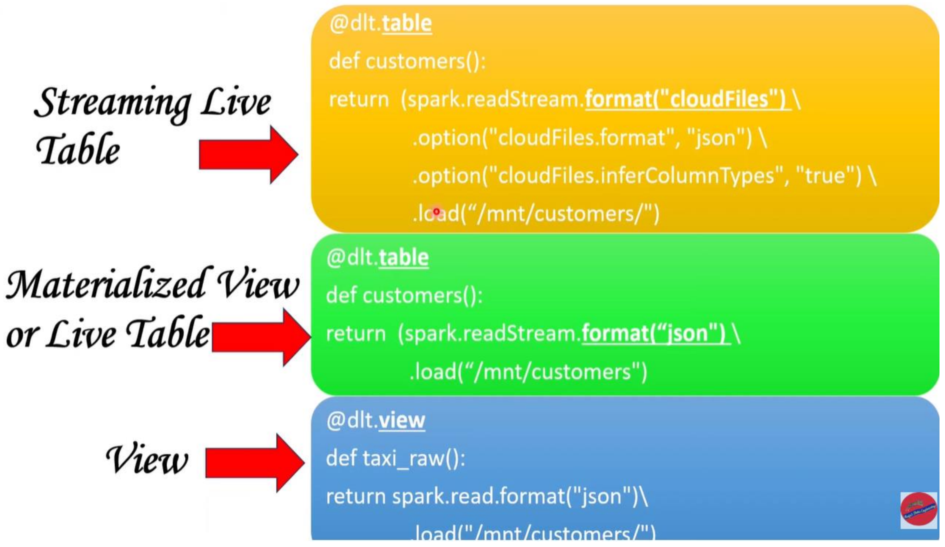


```
CREATE STREAMING TABLE customers
AS SELECT * FROM cloud_files("/mnt/customers/", "csv");

CREATE LIVE TABLE sales_aggreate
AS SELECT order_date, city, products, sum(sales_amount)
          FROM LIVE.sales s
          JOIN LIVE.customers
          WHERE product = 'LAPTOP'
          GROUP BY order_date, city, products

CREATE  LIVE VIEW V_sales_aggregate
AS SELECT order_date, city,products, sum(sales_amount)
          FROM LIVE.sales s
          JOIN LIVE.customers
          WHERE product = 'LAPTOP'
          GROUP BY order_date, city,products
```

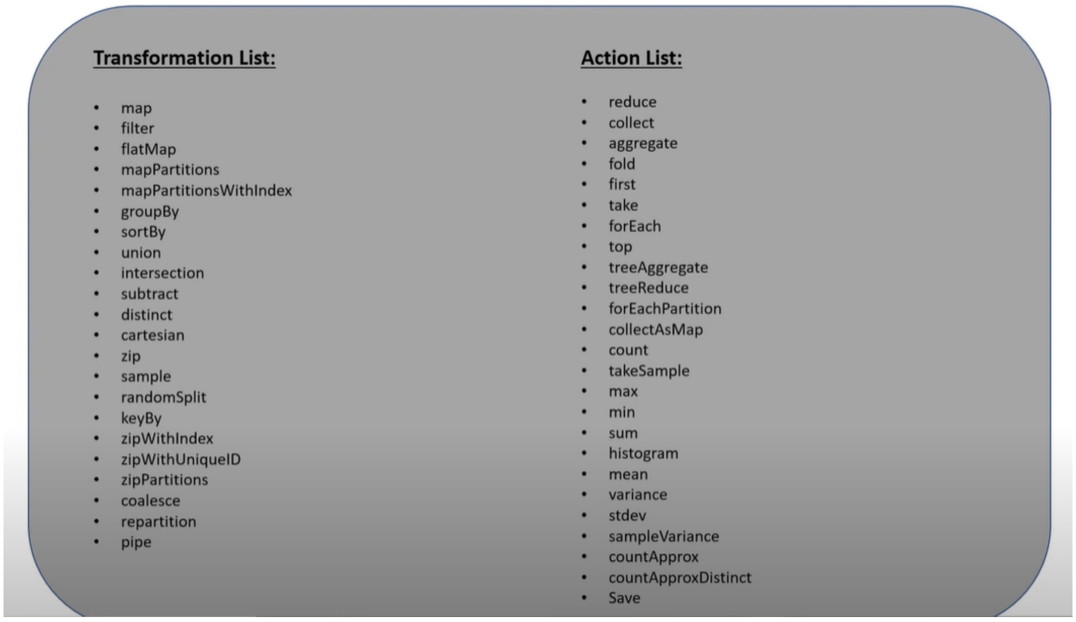
**Create tables using Spark code**



```
@dlt.table
def customers():
return  (spark.readStream.format("cloudFiles") \
          .option("cloudFiles.format", "json") \
          .option("cloudFiles.inferColumnTypes", "true") \
          .load("/mnt/customers/")

@dlt.table
def customers():
return  (spark.readStream.format("json") \
          .load("/mnt/customers")

@dlt.view
def taxi_raw():
return spark.read.format("json")\
          .load("/mnt/customers/")
```



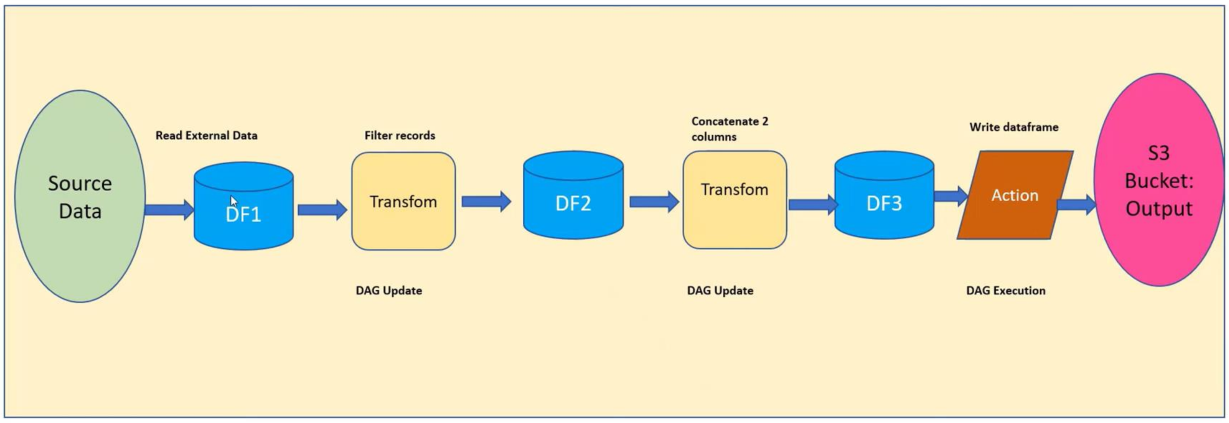
**Transformation List:**

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy
- union
- intersection
- subtract
- distinct
- cartesian
- zip
- sample
- randomSplit
- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- pipe

**Action List:**

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap
- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct
- Save

## Lazy Evaluation & DAG



**Narrow Transformations:** it’s simple and in expensive, not shuffle data

**Wide Transformations:** it’s very expensive and more complexity, shuffle the data across the data on nodes

**DB utils**:

dbutils.hep()

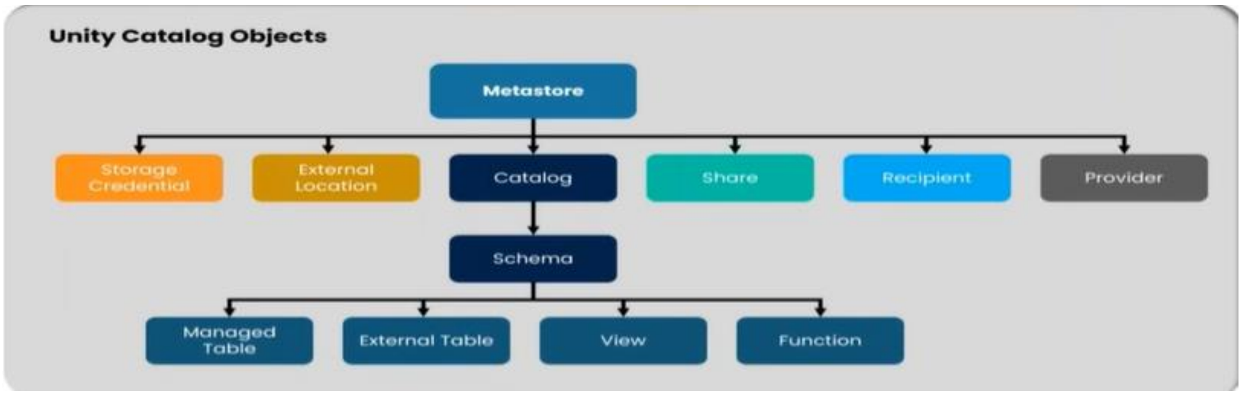
Data utils: it used to find the mean and median values, mostly used by data scientist

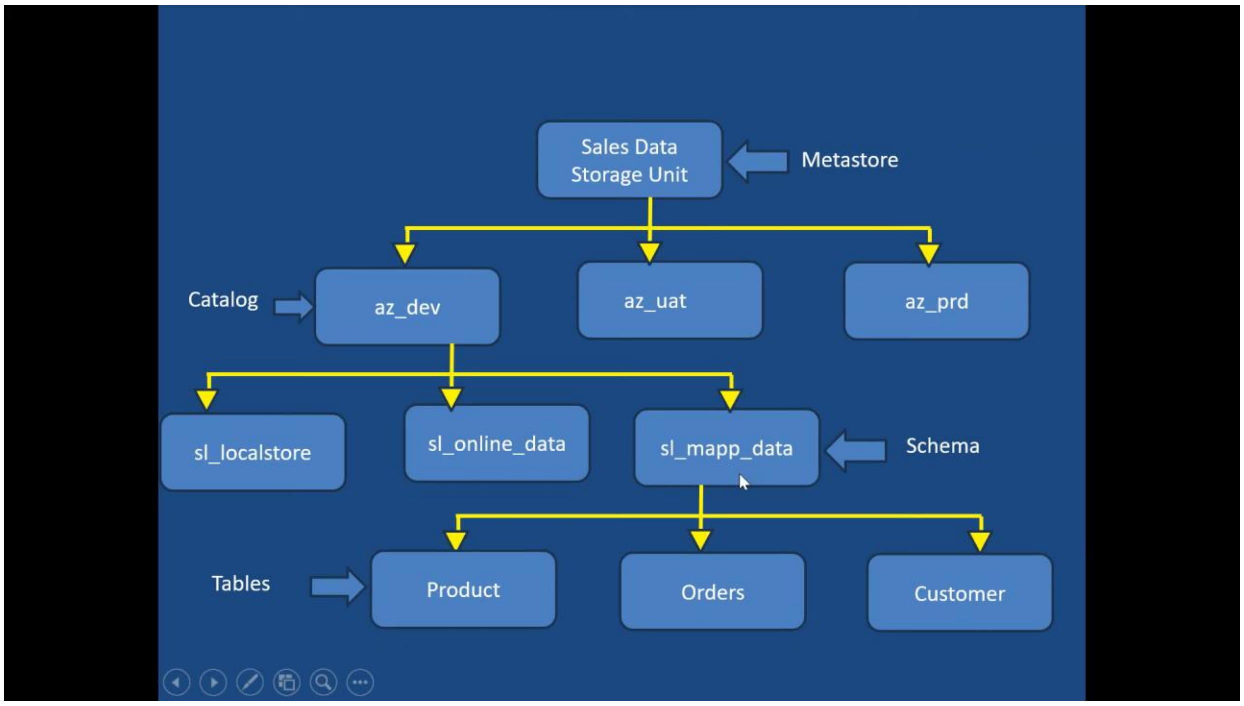
fs.utils: ls,mv,cp,

exit.utils : its exit with values,

## UNITY CATALOG :

- Centralized Metastore Repository for databricks workspace that provides data governance, lineage and it is used manage data across multiple data bricks workspace, users and applications
- for catalog migration, admin team can manage this migration part along dev team
- we need global admin access





---

**Managed Table**: basically, if you do not specify any location path, then it will create a managed table and store the metadata and data on its default path of databricks

If you drop managed table , it will remove metadata from spark_catalog and data on its own path

Default path: %fs ls 'dbfs:/users/hive/warehouse/customer_dim'

create table customer_dim (id int, name string, loc string)

**External Table**: if you specify the location then it will create external table and store the metadata and data into specified path

If you drop external table, it will remove only metadata from spark_catalog but data will be available in external path

Create table customer location '/users/hive/warehouse/customer'

create table customer (id int, name string, loc string)

we can see which table by using

1.show create table customer

2. %sql

describe table extended table name

* If you do not using format type by default it will take delta table  (default databricks format type)

Create table customer (id int, name string, loc string)

*you can use format type like below

Create table customer (id int, name string, loc string) using csv

---

CI / CD ?

Continuous integration and continuous delivery which helps to automate the code deployment

CI : once we commit, it will develop the code and it will do testing thoroughly, and analyse quality check in the code then it will ready to use for CD.

CD : once testing done, its successfully integrated then it will deploy the code across the dev environment and prod environment accordingly.

Procedure:  we have to setup repository like git hub, azure devops

Then we have integrated with data bricks. In the repository we have to create main branch which copy of dev resource group. Also have to create future branch here we have to deploy the new codes changes, once all done, we have raise pull request to merge the code into main branch.

Main branch will trigger the CI/CD pipeline and it will deploy the changes across different environments

Creating a CI/CD pipeline for Azure Databricks using GitHub and Azure DevOps involves several steps. Here's a high-level overview of the process:

1.Setting up Azure DevOps:

Create a new project in Azure DevOps.

Navigate to Pipelines and create a new pipeline.

Connect your GitHub repository to Azure DevOps.

2.Configuring Environments:

In Azure DevOps, navigate to Environments and create environments for Dev, QA1, QA, UAT, and Production.

3.Defining Build Pipeline:

Create a YAML or classic build pipeline in Azure DevOps to build your Databricks code.

Define necessary build steps such as pulling the code from GitHub, installing dependencies, compiling code, etc.

Ensure the build artifacts are published.

4.Setting up Release Pipeline:

Create a release pipeline in Azure DevOps.

Define stages corresponding to each environment (Dev, QA1, QA, UAT, Production).

Add pre-deployment conditions and approvals as needed.

5.Deploying to Environments:

Configure deployment tasks within each stage of the release pipeline to deploy your Databricks code to the corresponding environment.

Use Azure Databricks REST API or Azure CLI tasks in Azure DevOps to interact with Azure Databricks.

Configure necessary parameters for each environment (e.g., cluster configurations, Databricks workspace details, etc.).

6.Managing Configurations:

Utilize Azure Key Vault or Azure DevOps variable groups to manage sensitive configurations and secrets.

Ensure that environment-specific configurations are properly managed and passed during deployments.

7.Testing:

Implement automated testing within your CI/CD pipeline to validate changes before promoting them to the next environment. Define appropriate test cases for each environment and automate their execution.

8.Monitoring and Logging:

Implement monitoring and logging solutions to track the performance and behaviour of your Databricks workloads.

Integrate with Azure Monitor or other monitoring tools to monitor resource usage, job execution, and any errors.

9.Continuous Improvement:

Regularly review and update your CI/CD pipeline to incorporate new features, optimize performance, and address any issues.

Collect feedback from stakeholders and incorporate improvements based on their input.

By following these steps, you can establish a robust CI/CD pipeline for deploying Azure Databricks solutions using GitHub and Azure DevOps across multiple environments. Adjustments may be needed based on the specific requirements and nuances of your project.

---

## Assume we have setup all the things as mentioned above. Now, I have made some changes in the PySpark code existing in the Databricks. Now, how can I deploy those changes?

## Assume my senior has to review my code before merging the Changes in main branch and these changes will move from one environment to other environment by manual triggering

If you have set up your CI/CD pipeline as described and you've made changes to the PySpark code in Databricks that need to be deployed, you can follow these steps:

Commit Changes to GitHub:

After making changes to the PySpark code in Databricks, commit your changes to the appropriate branch in your GitHub repository.

Pull Request (PR) and Code Review:

Create a pull request from your branch to the main branch in GitHub.

Assign your senior for code review.

Your senior will review the changes and provide feedback/comments.

Make any necessary changes based on the feedback.

Merge Changes to Main Branch:

Once the code review is complete and any necessary changes have been made, merge the changes into the main branch in GitHub.

Trigger Deployment:

After merging the changes into the main branch, manually trigger the deployment in Azure DevOps.

Navigate to the Azure DevOps pipeline and select the appropriate release pipeline.

Click on the "Create Release" button or trigger the release manually.

Select the version of the code (e.g., from the main branch) to deploy.

Choose the appropriate environment to deploy to (e.g., Dev, QA1, QA, UAT, Production).

Confirm and start the deployment.

Deployment to Environments:

The release pipeline will deploy the changes to the selected environment.

Follow the deployment progress in Azure DevOps and monitor for any errors or issues.

Once the deployment is complete, verify that the changes are successfully deployed and functioning as expected in the environment.

Repeat for Other Environments:

If the deployment is successful in the first environment, repeat the process for deploying to other environments (e.g., QA1, QA, UAT, Production) by manually triggering the release in Azure DevOps for each environment.

By following these steps, you can deploy changes made to the PySpark code in Databricks through a controlled process that includes code review, merging changes, and manual triggering of deployments to different environments via Azure DevOps.

## What is fact table?

A fact table is a central table in a star schema or snowflake schema of a data warehouse. It contains quantitative data (facts) about a business process or activity. Fact tables primarily consist of numeric data representing measurements or metrics. Fact tables contain foreign keys that link to dimension tables. These keys are used to establish relationships between the facts and the various dimensions associated with those facts.

## What is dimensional table?

A dimensional table, also known as a dimension table, is a type of table in a star schema or snowflake schema of a data warehouse

**Descriptive Attributes:** Dimension tables contain descriptive attributes that provide context or additional information about the data in the fact table. These attributes could include things like product names, customer demographics, geographic locations, time periods, etc.

**Primary Key**: Dimension tables typically have a primary key that uniquely identifies each record in the table. This primary key is often used as a foreign key in the fact table to establish relationships between the fact table and the dimension table.

**Example:**

In a retail business scenario, dimensional tables might include:

Product dimension table: containing attributes such as product ID, product name, category, brand, etc.

Customer dimension table: containing attributes such as customer ID, name, age, gender, location, etc.

These dimension tables provide additional context and descriptive information about the sales data stored in the fact table, allowing for more meaningful analysis and reporting.

## What is star schema and snowflake schema? what are the difference between star schema and snowflake schema?

Star schema and snowflake schema are two common approaches used for designing data warehouse schemas. Both schemas are used to organize data in a dimensional model, which is optimized for querying and analysing large volumes of data in a data warehouse environment.

### Star Schema:

1.In a star schema, the central fact table is surrounded by multiple dimension tables.

2.The fact table sits at the center of the schema and contains quantitative data (facts) related to a specific business process or activity.

3.Dimension tables are connected to the fact table through foreign key relationships.

4.Dimension tables contain descriptive attributes that provide context to the data in the fact table.

The structure of a star schema resembles a star, with the fact table at the center and dimension tables radiating out from it like the arms of a star.

5.Star schemas are denormalized, meaning that dimension tables are typically denormalized to eliminate redundancy and optimize query performance.

6.Star schemas are simple to understand and query, making them suitable for most analytical and reporting needs.

### Snowflake Schema:

1.A snowflake schema is an extension of the star schema where dimension tables are normalized into multiple related tables.

Unlike the star schema, where dimension tables are denormalized, in a snowflake schema, dimension tables are normalized, resulting in a more complex structure.

2.Dimension tables in a snowflake schema may be broken down into multiple related tables, each containing a subset of attributes.

Normalization reduces redundancy and improves data integrity but can complicate queries and impact query performance, especially for complex analytical queries.

3.The structure of a snowflake schema resembles a snowflake, with the fact table at the center and dimension tables branching out into multiple related tables like the arms of a snowflake.

4.Snowflake schemas are more complex to understand and query compared to star schemas, but they offer improved data integrity and potentially better storage efficiency.

## Differences between Star Schema and Snowflake Schema:

**Structure**: Star schema has a simpler structure with denormalized dimension tables, while snowflake schema has a more complex structure with normalized dimension tables.

**Normalization:** Star schema is denormalized, while snowflake schema is normalized.

**Query Performance:** Star schema typically offers better query performance compared to snowflake schema, especially for simpler queries. Snowflake schema may suffer from performance issues due to the normalized structure and additional joins required.

**Complexity:** Snowflake schema is more complex to understand and manage compared to star schema due to the normalization of dimension tables.

**Data Integrity**: Snowflake schema may offer better data integrity due to normalization, while star schema may have some redundancy but is simpler to maintain.

---

## **What is Data Modelling**?

Data modelling is the process of designing the structure and organization of data in a database or data warehouse system. It involves defining the relationships between different data elements, specifying the rules for how data should be stored and accessed, and creating a blueprint for how data will be represented and managed within the system.

**Key aspects of data modelling include:**

**Entity-Relationship Modelling:** This involves identifying the entities (objects or concepts) in the domain of interest and defining the relationships between them. Entities represent real-world objects such as customers, products, orders, etc., while relationships describe how entities are related to each other.

**Schema Design:** Data modelling includes designing the schema, which defines the structure of the database or data warehouse. This involves defining tables, columns, keys, constraints, and other structural elements necessary to represent the data.

**Data Integrity:** Data modelling includes defining constraints and rules to enforce data integrity, such as primary key constraints, foreign key constraints, unique constraints, and check constraints. These constraints ensure that data remains consistent and accurate throughout the database.

**Data Types and Attributes:** Data modelling involves specifying the data types and attributes for each column in the database tables. This includes defining the size, format, and constraints for each data element.

## How to integrate azure data bricks notes into azure data factory pipeline ?

Yes, we have to launch databricks workspace and create notebook and make sure attach cluster

Now launch the Azure data factory then open one existing pipeline or new pipeline.

In the activities, choose the databricks and while providing connection choose cluster name and notebook name. make a connection

## How to change code in existing pipeline in in databricks ?

Yes, we can modifications for that we have open the existing pipeline in the workflows then open pipeline settings and choose UI / JSON , I will go with JSON because we can change code dynamically

## Azure Data Factory:

Azure Data Factory is a cloud-based data integration service provided by Microsoft Azure. It allows you to create, schedule, and orchestrate data pipelines to move and transform data from various sources to various destinations.

**Linked services:**

It is used to establish the connection between the source storage to sink storages.

**Dataset:**

It used to read and write the data from source and destination storages.

## Activities:

**Copy Activity:**

This activity is used for moving data between different data storages.

**Get Metadata Activity:**

This activity retrieves metadata information about a dataset or a folder such as file properties, schema information, it is often used for dynamic dataset resolution and validation.

**Lookup Activity:**

The lookup activity is used to retrieve data from a dataset, configuration, files and and It is commonly used for parameterizing pipelines and passing dynamic values to downstream activities.

**For Each Activity:**

This activity is used to iterate over a collection and executes specified activities in a loop.

**Wait Activity:**

It is used to delay or pause within a pipeline's execution. This activity is particularly useful for scenarios where you need to wait for a certain period before executing subsequent activities in the pipeline.

**Until Activity:**

It executes a set of activities in a loop until the condition associated with the activity evaluates to true. If an inner activity fails, the Until activity does not stop. You can specify a timeout value for the until activity.

**If Condition Activity:**

It executes a set of activities when the condition is true and another set of activities when the condition checks to false

## Triggers:

**Schedule trigger:**

Schedule trigger in Azure Data Factory (ADF) enables the automatic execution of pipelines on a predefined schedule, it provides flexibility in scheduling pipelines based on time intervals or specific dates and times.

**Tumbling trigger:**

This trigger is used to define time-based windows, such as hourly, daily, or weekly intervals, during which pipeline executions occur. It's particularly useful for data processing tasks aligned with specific time windows.

**Event Based trigger:**

An event-based trigger initiates pipeline executions based on events occurring in external systems or services. It can be triggered by events such as file arrival, HTTP request, or Azure Blob Storage events.

## Integration Runtimes in ADF

**Azure Integration Runtime:**

The Azure Integration Runtime is used for data movement between cloud data stores within Azure. It is fully managed by Azure Data Factory and requires no additional setup. This runtime is used for activities such as copying data between Azure Blob Storage, Azure SQL Database, Azure Data Lake Storage, etc.

**Self-Hosted Integration Runtime:**

The Self-hosted Integration Runtime allows Azure Data Factory to connect to on-premises data sources. It is installed on an on-premises machine or a virtual machine (VM) within your network and acts as a gateway between your on-premises data sources and Azure. This runtime is used for activities such as reading from or writing to on-premises databases, file systems, or APIs.

## Azure Key Vault

Azure Key Vault in Azure Data Factory (ADF) securely stores and manages sensitive information such as credentials, secret keys and certificates used by ADF pipelines. It enables centralized management and access control for storing and retrieving sensitive data, enhancing security and compliance in data integration workflows.