DATABRICKS THEORETICAL Q&A

BY - SHUBHAM WADEKAR

1. What is Azure Databricks?

Azure Databricks is a cloud-based data platform that is built on Apache Spark and is fully integrated with Microsoft Azure. It is mainly used for big data analytics, data engineering, and machine learning.

From a candidate's point of view, I would say:
Azure Databricks provides a collaborative environment where I can work with notebooks to write code in languages like Python, SQL, Scala, or R. It allows me to process large amounts of data very quickly by distributing the work across multiple machines using Spark. The best part is that it automatically handles the setup, configuration, and scaling of the infrastructure, so I can focus more on writing my data pipeline or model code instead of managing servers.

It also works well with other Azure services, like Azure Data Lake, Azure Synapse, and Azure Machine Learning, which helps in building complete data solutions. For example, I can read raw data from Azure Data Lake, clean and transform it in Databricks, and write the output to a database or feed it into a machine learning model   all in one seamless workflow.

2. What are the advantages of Microsoft Azure Databricks?

Azure Databricks offers several advantages, especially when working on large-scale data and analytics projects. Some of the main benefits are:

- **Fast processing of large data**: Since it is built on Apache Spark, it can handle very large volumes of data and perform operations in parallel across many machines. This means faster execution of jobs.
- **Scalability**: It automatically scales up or down depending on the workload. This is helpful when I want to optimize cost and performance.
- **Easy collaboration**: It provides interactive notebooks where different team members like data engineers, data scientists, and analysts can work together in the same workspace.
- **Built-in security and integration**: It uses Azure's security model, including role-based access control (RBAC), network security, and managed identities. It also integrates well with Azure Data Lake, Azure Blob Storage, Azure Synapse, Power BI, and other Azure services.
- **Supports multiple languages**: I can write code in Python, SQL, Scala, or R depending on the requirement.
- **Job orchestration**: I can schedule jobs, set dependencies, and monitor workflows easily within the Databricks workspace.
- **Cost efficiency**: Since we pay only for what we use (based on DBUs), it allows better control over spending compared to traditional data processing setups.

3. Why is it necessary for us to use the DBU Framework?

DBU stands for Databricks Unit. It is a unit of processing capability per hour that is used to measure the usage of resources in Azure Databricks.

From my experience, using the DBU framework is necessary because it provides a clear and flexible way to track and control the cost of running Databricks workloads. Each type of workload (for example, standard, premium, or jobs) consumes DBUs at different rates depending on the cluster type and size. This helps me and my team to estimate how much a job will cost before running it, which is very useful for budgeting.

Also, since DBU usage is separated from the virtual machine (VM) cost, I can manage compute and Databricks processing costs more transparently. This separation allows better flexibility in choosing VM types while keeping track of how much I'm paying specifically for Databricks services.

In short, the DBU framework helps in measuring, optimizing, and managing the cost of data processing in Azure Databricks, and that's why it's an essential part of using the platform effectively.

4. When referring to Azure Databricks, what exactly does it mean to "auto-scale" a cluster of nodes?

Auto-scaling in Azure Databricks means that the platform can automatically increase or decrease the number of worker nodes in a cluster depending on the workload.

From a candidate's perspective, I would explain it like this:
When I run a job or a notebook that requires a lot of processing power, auto-scaling will add more worker nodes to handle the load. If the job is light or the load decreases, it will remove the extra nodes to save costs. This helps in optimizing both performance and cost.

For example, if I set a cluster to auto-scale between 2 and 8 workers, it might start with 2 nodes. When the job becomes more demanding, it may increase to 6 or 8 nodes. Once the demand drops, it might scale back down to 2 again.

This is very useful because I don't have to manually monitor the system or guess how many nodes I'll need. It also helps prevent overprovisioning or underprovisioning resources.

5. What actions should I take to resolve the issues I'm having with Azure Databricks?

When facing issues in Azure Databricks, here are the steps I usually follow to troubleshoot and resolve them:

1. **Check cluster logs**: I start by going to the cluster UI and checking the driver and worker logs. These logs often show errors or warnings that help identify the root cause.
2. **Review job run logs**: If I'm running a notebook or a scheduled job, I look at the job run output and logs to see exactly where it failed.
3. **Check cluster status and configuration**: Sometimes, issues happen because of misconfigured clusters (e.g., low memory, wrong instance type). I make sure the cluster is running and properly sized for the workload.
4. **Look at Spark UI**: The Spark UI gives detailed insights into stages and tasks. I use it to find long-running stages or memory issues.

5. **Check notebook code**: I check the code for any inefficient queries, incorrect syntax, or bad logic that could be causing performance or runtime issues.
6. **Azure resource limitations**: Sometimes, the issue is on the Azure side, like quota limits or region-specific problems. In that case, I contact Azure support or check the Azure portal for alerts.
7. **Use Databricks community or support**: If I can't resolve the issue myself, I reach out to Databricks support or search through their documentation and forums.

By following these steps, I'm usually able to narrow down and fix the problem effectively.

6. What is the function of the Databricks filesystem?

The Databricks File System (DBFS) is a distributed file system that is built on top of cloud storage. It acts as a layer between Databricks and the cloud storage, and it helps manage files that are used within Databricks notebooks and jobs.

From my point of view, DBFS is like a shared drive inside the Databricks workspace. It lets me upload, read, write, and manage data files easily. For example, I can upload a CSV file to DBFS and then read it using a simple file path like /dbfs/FileStore/mydata.csv.

The main functions of DBFS are:

- **Storage of input and output data**: I can store data files that are used in my Spark jobs or machine learning models.
- **Temporary workspace**: DBFS can be used for staging files or intermediate results.
- **Access to cloud storage**: Behind the scenes, DBFS stores data in Azure Blob Storage, so it's reliable and scalable.
- **Easy integration**: I can access DBFS from notebooks or through REST APIs, making it easy to automate tasks.

In summary, DBFS makes it simple to work with files in Azure Databricks, and it helps bridge the gap between the cloud storage and the code running in the notebooks.

7. What programming languages are available for use when interacting with Azure Databricks?

Azure Databricks supports multiple programming languages that are commonly used in data engineering, data science, and analytics. The main languages are:

- **Python**: This is one of the most popular languages used in Databricks. It's often used for data processing with PySpark, and for machine learning using libraries like scikit-learn, TensorFlow, or MLflow.
- **SQL**: This is widely used for querying structured data and is great for analysts or data engineers who prefer writing SQL queries over code.
- **Scala**: Since Databricks is built on Apache Spark, which is written in Scala, this language is good for high-performance Spark jobs.
- **R**: This is mainly used by data scientists for statistical computing and data visualization.

All of these languages can be used in the same workspace. In fact, within a single notebook, I can switch between these languages using magic commands like %python, %sql, %scala, and %r.

8. Is it possible to manage Databricks using PowerShell?

Yes, it is possible to manage Azure Databricks using PowerShell, especially when working with the Azure CLI and REST APIs.

From my experience, I can use PowerShell scripts to automate tasks such as:

- Creating and managing Databricks workspaces
- Assigning users and permissions
- Interacting with the Databricks REST API to create jobs, clusters, and notebooks
- Automating deployments and configurations as part of CI/CD pipelines

PowerShell doesn't directly interact with Databricks functions the way a notebook does, but it is very useful for infrastructure automation when setting up environments using ARM templates, Azure CLI, or Bicep scripts.

9. Which of these two, a Databricks instance or a cluster, is the superior option?

Actually, these two are not alternatives but are part of the same system and serve different purposes. Let me explain it clearly:

- A Databricks instance (also called a workspace) is the environment where I work. It includes notebooks, jobs, libraries, and access controls. It's like the main dashboard or UI where I manage all my work.
- A cluster is the actual compute environment where the code runs. It's made up of virtual machines and Spark components. I can have multiple clusters within one Databricks instance.

So, asking which one is superior is like comparing a car's dashboard to its engine. Both are needed, but they serve different roles.

In summary, a Databricks instance is the overall workspace, and a cluster is the engine that runs the workloads. I need both to do anything meaningful in Azure Databricks.

10. What is meant by the term "management plane" when referring to Azure Databricks?

The management plane in Azure Databricks refers to the part of the platform that is responsible for handling administrative and control tasks. This includes:

- Authentication and user management
- Workspace configuration
- Job scheduling and orchestration
- Cluster management and configuration
- Notebook and library management

In simple terms, the management plane is the control center that allows me to create and manage resources in Databricks. It does not run the actual code or jobs   that happens in the data plane, which is where the clusters live and execute the workloads.

The management plane is managed by Databricks and hosted in the Databricks control environment, while the data plane usually runs in the customer's Azure environment, ensuring data stays within the customer's boundary.

Understanding this separation is important for security, because it means sensitive data processed by Spark stays in the data plane, while control operations are handled by the management plane.

11. Where can I find more information about the control plane that is used by Azure Databricks?

To find more detailed and official information about the control plane in Azure Databricks, I usually refer to the Databricks documentation, specifically the security and architecture sections. The best place is the official Databricks documentation site:

- https://learn.microsoft.com/en-us/azure/databricks

There, I can find architectural diagrams and explanations about how the control plane and data plane are separated. Additionally, Databricks has a dedicated section for compliance, security architecture, and networking where the control plane is explained in depth.

If I'm working on a project that requires more detailed security planning, I also look into the "Secure cluster connectivity" and "Customer-managed VPC" features which explain how data flows through the control plane and how access is controlled.

12. What is meant by the term "data plane" when referring to Azure Databricks?

The data plane in Azure Databricks is the part of the platform where the actual data processing happens. It is where my Spark clusters run and where all compute operations are performed.

From a simple point of view, I think of it like this:
The data plane is where the jobs, notebooks, and commands are executed. It's where the raw data is read, transformed, and written. This data typically lives in cloud storage like Azure Data Lake or Blob Storage, and the compute (clusters) accesses that data within the same network boundary.

An important point is that, in Azure Databricks, the data plane runs inside my Azure subscription. This means the data never leaves my control. The control plane, on the other hand, is managed by Databricks and used to manage the platform itself.

This separation helps with data security and compliance, especially when handling sensitive or regulated data.

13. Is there a way to halt a Databricks process that is already in progress?

Yes, there are a few ways to stop or cancel a process in Azure Databricks if it's already running.

Here's what I usually do:

1. **Stop a notebook cell**: If a cell in a notebook is running and I want to stop it, I can click the "Cancel" button next to the running cell.
2. **Terminate a job**: If I launched a job (either manual or scheduled) from the Jobs UI, I can go to the Jobs tab, find the active job run, and click the "Cancel Run" button.
3. **Kill a Spark job**: If I want to stop a specific Spark stage or task, I can open the Spark UI, find the job/stage, and cancel it from there.
4. **Terminate the cluster**: If nothing else works or the job is unresponsive, I can shut down the cluster itself. This will kill all active processes running on that cluster.

These options are useful when a job is stuck in an infinite loop or running longer than expected.

14. What is a delta table in Databricks?

A delta table in Databricks is a type of table that is built on top of the Delta Lake storage format. It brings features like ACID transactions, version control, and schema enforcement to data lakes.

In simpler terms, it allows me to manage big data as if I'm working with a traditional database, but in a much faster and more flexible way.

Here's why I use delta tables:

- **ACID Transactions**: This means I can safely read and write data without worrying about data corruption, even when multiple users or processes are working at the same time.
- **Schema evolution**: I can add or change columns easily, and the table can automatically adapt to the new schema.
- **Time travel**: I can access older versions of the data using timestamps or version numbers. This is helpful for debugging or rollback.
- **Efficient updates and deletes**: Unlike traditional data lakes, I can perform UPDATE, DELETE, and MERGE operations on delta tables.
- **Faster performance**: Delta tables use indexing and caching to improve performance on large datasets.

Overall, delta tables allow me to treat my data lake like a transactional database, which is very useful for building reliable and maintainable data pipelines.

15. What is the name of the platform that enables the execution of Databricks applications?

The platform that enables the execution of Databricks applications is called **Apache Spark**.

Azure Databricks is built on top of Apache Spark, which is a powerful open-source distributed computing engine. Spark allows Databricks to process large volumes of data quickly by distributing the data across a cluster of machines and performing computations in parallel.

Databricks adds a lot of extra features on top of Spark, such as optimized connectors, interactive notebooks, collaboration tools, Delta Lake support, and easy integration with Azure services. But at its core, all the processing is done by the Apache Spark engine.

So whenever I run code in Databricks, whether it's Python, SQL, or Scala, that code is being executed on Spark behind the scenes.

16. What are workspaces in Azure Databricks?

A workspace in Azure Databricks is the main environment where users collaborate and manage all their Databricks assets. It's like the home base or project space where I can:

- Create and run notebooks
- Launch and manage clusters
- Schedule jobs
- Install libraries
- Manage users, permissions, and groups

Workspaces are helpful because they organize all the development work and resources in one place. Each Azure Databricks workspace is tied to an Azure resource and is deployed within an Azure subscription.

Inside the workspace, everything is arranged in folders, which makes it easy to manage notebooks and projects. I can also control access to these folders using role-based access control (RBAC), which is useful for team collaboration.

In short, the workspace is the central place where I build, test, and run my data and analytics projects in Azure Databricks.

17. In the context of Azure Databricks, what is a "dataframe"?

In Azure Databricks, a DataFrame is a distributed collection of data organized into named columns, just like a table in a database or a spreadsheet.

When I'm working with Spark using PySpark, Scala, or even SQL, the DataFrame is one of the main ways to work with structured data.

Here's what makes DataFrames useful in Databricks:

- They are lazy: Operations are not run until an action is triggered, which allows Spark to optimize the query plan.
- They support SQL-like operations: I can filter, group, join, and aggregate data using a familiar syntax.
- They are distributed: The data is split across multiple machines, so large datasets can be processed efficiently in parallel.
- They are versatile: I can create DataFrames from various sources like CSV files, JSON files, Delta tables, SQL queries, or even in-memory data.

For example, I might load a CSV file into a DataFrame using PySpark like this:

df = spark.read.csv("/mnt/data/myfile.csv", header=True, inferSchema=True)

Then I can run commands like df.select("name").filter("age > 25") to analyze the data.

18. Within the context of Azure Databricks, what role does Kafka play?

Apache Kafka is often used in Azure Databricks as a real-time data streaming source.

Kafka is a distributed messaging system that allows producers to send data into topics, and consumers (like Databricks) to read data from those topics.

In Databricks, Kafka plays the role of ingesting real-time data into the platform. For example, if I'm building a streaming pipeline, I can use Kafka to receive data from devices, logs, applications, or websites and then process it in near real-time using Spark Structured Streaming.

Here's how Kafka fits into Databricks workflows:

- Databricks connects to Kafka topics using connectors.
- Streaming data is read into a DataFrame using Spark Structured Streaming.
- I can perform operations like filtering, aggregating, and writing to other sinks like Delta Lake or Azure Synapse.

An example in PySpark would look like:

```
df = spark.readStream.format("kafka") \
  .option("kafka.bootstrap.servers", "broker:9092") \
  .option("subscribe", "my-topic") \
  .load()
```

In summary, Kafka is used in Databricks to support real-time data pipelines, which is important for scenarios like fraud detection, real-time analytics, or monitoring.

19. Is it only possible to access Databricks through the cloud, and there is no way to install it locally?

Yes, Databricks is a cloud-native platform, and currently, there is no supported way to install or run it entirely on a local machine.

Databricks is offered as a managed service on cloud platforms like Microsoft Azure, AWS, and Google Cloud. This means the infrastructure, updates, and management are all handled by Databricks, and I just focus on building data pipelines, analytics, or machine learning models.

While I can run Apache Spark locally for development and testing purposes, I won't get all the additional features that come with Databricks, such as collaborative notebooks, auto-scaling clusters, job scheduling, Delta Lake, and built-in integrations with cloud storage.

So, if I want to use the full functionality of Databricks, I need to access it through one of the supported cloud platforms   it's not meant to be run locally.

20. Is Databricks a Microsoft subsidiary or a subsidiary company?

Databricks is not a Microsoft subsidiary. It is an independent company that was founded by the creators of Apache Spark.

Microsoft and Databricks have a strong partnership, and that's why Azure Databricks exists   it's a joint offering between Databricks and Microsoft, integrated deeply into the Azure ecosystem. But Databricks itself remains a separate company with its own leadership, funding, and products.

In summary, Microsoft provides the Azure platform and integrates Databricks as a first-party service, but it does not own Databricks.

21. Could you please explain the many types of cloud services that Databricks offers?

Databricks provides several types of cloud services, mainly focused on big data, analytics, and machine learning. These include:

1. **Data Engineering**
   This service is used to build and manage data pipelines. It allows me to ingest, clean, and transform large amounts of data using Spark. It supports both batch and real-time data processing.

2. **Data Science and Machine Learning**
   Databricks offers a collaborative environment for building, training, and deploying machine learning models. It supports popular libraries like MLflow, scikit-learn, TensorFlow, and PyTorch. It also provides autoML tools.

3. **Data Analytics and BI**
   Databricks supports SQL analytics and dashboarding. I can query data using SQL in notebooks or use tools like Power BI for reporting. It also has built-in visualizations.

4. **Delta Lake**
   This is a storage layer that brings reliability to data lakes. It enables ACID transactions, versioning, schema enforcement, and time travel for data stored in cloud storage.

5. **Real-Time Streaming**
   Using Spark Structured Streaming, I can connect to streaming sources like Kafka or Event Hubs and process data in real-time for use cases like fraud detection or monitoring.

6. **Workflow Orchestration**
   Databricks Jobs and Task Orchestration features allow scheduling and managing pipelines and dependencies, similar to Airflow or Data Factory.

Each of these services can be used together to build complete end-to-end data and AI solutions in the cloud.

22. Which category of cloud service does Microsoft's Azure Databricks belong to: SaaS, PaaS, or IaaS?

Azure Databricks falls under the PaaS (Platform as a Service) category.

This is because it provides a fully managed environment where I can develop, run, and manage applications (like data pipelines or ML models) without having to deal with the underlying infrastructure such as servers, networking, or Spark installation.

In other words, with Azure Databricks:

- I don't manage the hardware or operating system (not IaaS).
- I'm not just consuming a finished app (so not SaaS either).
- I'm using a platform to build and run my own applications, which makes it PaaS.

So, Azure Databricks is a platform that offers computing, collaboration, and storage features all abstracted from the user   which fits the PaaS definition perfectly.

23. Differences between Microsoft Azure Databricks and Amazon Web Services Databricks

Azure Databricks and AWS Databricks are both managed versions of the Databricks platform, but they differ in their integration with their respective cloud environments. Here's how I usually explain the key differences:

- **Cloud Platform Integration**
  Azure Databricks is deeply integrated with Azure services like Azure Data Lake Storage, Azure Synapse, Azure Key Vault, Azure Monitor, and Azure Active Directory.
  On the other hand, AWS Databricks integrates with AWS services such as S3 (for storage), Redshift, IAM (for access control), CloudWatch (for monitoring), and KMS (for encryption).
- **Workspace Deployment**
  In Azure, Databricks is offered as a first-party service, meaning I can create it directly from the Azure Portal, and billing is unified under Azure.
  In AWS, it is a third-party service, and I typically manage billing separately through Databricks.
- **Authentication and Security**
  Azure Databricks uses Azure Active Directory for authentication and access control, which helps in centralizing identity management.
  AWS Databricks uses AWS IAM roles and policies.
- **Networking and Setup**
  Azure has features like VNet injection and secure cluster connectivity to tightly integrate Databricks into the Azure virtual network.
  AWS uses similar options with VPCs and private link features.
- **UI and Experience**
  The core Databricks workspace experience is largely the same, but the way resources like clusters, jobs, and notebooks are managed can vary slightly based on the cloud platform's backend setup.

So in summary, both versions provide the same core Databricks features, but the integration points, management, and security options are specific to the cloud provider.

24. What does "reserved capacity" mean when referring to Azure?

In Azure, "reserved capacity" means committing to use a specific amount of computing resources for a longer period (like 1 or 3 years) in exchange for a lower cost.

From a user's perspective, it's like buying a subscription instead of paying per use. Instead of being charged hourly at the pay-as-you-go rate, I can reserve certain resources like virtual machines, SQL databases, or Synapse compute and get significant discounts   usually up to 60-70%.

For example, if I know that I'll be using a certain size of Azure SQL database or Databricks cluster regularly for the next year, I can reserve that capacity in advance. This helps with budget planning and cost optimization.

Reserved capacity is especially useful in production environments where I expect a consistent workload.

25. Does the deployment of Databricks necessitate the use of a public cloud service such as AWS or Microsoft Azure, or can it be done on an organization's own private cloud?

Yes, Databricks currently requires deployment on a public cloud service. It is designed as a cloud-native platform and is offered only on public cloud providers like Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform (GCP).

There is no official support for deploying Databricks in a private cloud or on-premises environment.

From a practical standpoint, Databricks takes advantage of the scalability, storage, and networking features of public clouds. It also relies on cloud services for things like distributed compute, identity management, and integration with other data services.

If an organization wants to work with Spark in a private cloud, they can install and manage Apache Spark directly on their own servers, but they would lose out on the features that Databricks offers like Delta Lake, collaborative notebooks, auto-scaling, job orchestration, and managed clusters.

So to use Databricks specifically, it must be deployed in a supported public cloud.

26. Is Apache Spark capable of distributing compressed data sources (.csv.gz) in a successful manner when utilizing it?

Yes, Apache Spark can successfully read and process compressed files like .csv.gz.

Spark automatically detects the compression type based on the file extension and decompresses it during reading. So if I have a .csv.gz file stored in a data lake or any file system accessible to Spark, I can load it just like a regular CSV file using Spark APIs.

Here’s an example in PySpark:

```
df = spark.read.option("header", True).csv("path/to/myfile.csv.gz")
```

This works without needing to manually unzip the file or write custom logic. Spark distributes the workload across its workers, so even large compressed files can be processed in parallel although it's important to know that .gz files are not splittable, which can affect parallelism.

For better performance with parallel reading, formats like Parquet or Avro (with snappy compression) are often preferred, since they support efficient columnar storage and can be split across tasks.

27. Is the implementation of PySpark DataFrames entirely unique when compared to that of other Python DataFrames, such as Pandas, or are there similarities?

PySpark DataFrames and Pandas DataFrames share many similarities, but they are not identical in how they work or are implemented.

Similarities:

- Both are structured data representations with columns and rows, similar to tables in a database.
- They support many common operations like select, filter, groupBy, agg, join, and more.
- Syntax is often similar, especially for basic transformations.

For example, in both I might write:

```
# Pandas

df[df["age"] > 30]


# PySpark

df.filter(df["age"] > 30)
```

Differences:

- **Execution model**: Pandas works in-memory on a single machine, while PySpark DataFrames run on top of Apache Spark and are distributed across multiple machines.
- **Performance**: PySpark can handle huge datasets that don't fit in memory, whereas Pandas is best for smaller datasets that fit in local RAM.
- **Laziness**: PySpark uses **lazy evaluation**, meaning transformations are not executed until an action (like show() or write()) is called. Pandas runs operations immediately.
- **APIs and behavior**: Some functions may exist in both but behave slightly differently due to how Spark handles data internally.

So while PySpark and Pandas DataFrames are similar in structure and purpose, their underlying architecture, scalability, and execution strategies are quite different. As a data engineer, I often start with Pandas for small-scale prototyping, and switch to PySpark when working with big data in distributed environments.

28. Explain the types of clusters that are accessible through Azure Databricks as well as the functions that they serve

Azure Databricks offers different types of clusters, each suited for specific use cases. These clusters are the compute environments where all the code is executed. The main types of clusters available in Azure Databricks are:

1. **Interactive Clusters (All-Purpose Clusters)**
   These clusters are used for interactive development, like running notebooks, exploring data, and doing ad hoc analysis.
   I use them mostly during the development phase, because I can attach multiple notebooks, run commands, and visualize data in real time. They are ideal when I need flexibility for experimentation and collaboration.

2. **Job Clusters**
   These are created specifically to run a job or task and are automatically terminated once the job is done.
   I use these for production workloads or scheduled jobs to save cost and ensure that resources are only used when needed. Job clusters are isolated, so they help in reducing risks like conflicts between different workloads.

3. **High Concurrency Clusters**
   These are optimized for serving multiple users at the same time, mainly for BI tools or dashboards.
   They provide strong resource isolation and are usually used when integrating with tools like Power BI or Tableau where many queries are expected from different users.

Each of these cluster types can also be configured for auto-scaling, custom libraries, and different runtime versions depending on the workload.

29. What are the Benefits of Using Kafka with Azure Databricks?

Using Kafka with Azure Databricks is very beneficial when working with real-time or streaming data. Here are some of the key benefits:

- **Real-Time Data Processing**
  Kafka lets me stream data in real time from various sources like IoT devices, applications, or logs. Databricks can consume that data and process it instantly using Spark Structured Streaming.
- **Scalability**
  Both Kafka and Databricks are highly scalable. Kafka can handle a huge number of events per second, and Databricks can process those events in parallel across multiple nodes.
- **Reliability and Fault Tolerance**
  Kafka stores data in a distributed and durable way. If something fails in the pipeline, I can re-read messages from Kafka to recover.
- **Seamless Integration**
  Databricks provides built-in connectors to consume data from Kafka (including Azure Event Hubs, which is Kafka-compatible). This makes it easy to integrate without writing complex code.
- **End-to-End Streaming Pipelines**
  I can use Kafka as the source, perform data cleaning or enrichment in Databricks, and then write the processed data to Delta Lake, databases, or push to downstream applications.

Overall, Kafka with Databricks is ideal for building streaming data pipelines such as real-time analytics, monitoring systems, fraud detection, and alerting applications.

30. Do I have the freedom to use various languages in a single notebook, or are there significant limitations? Would it be available for usage in further phases if I constructed a DataFrame in my python notebook using a %Scala magic?

Yes, in Azure Databricks, I do have the flexibility to use multiple languages within a single notebook. Databricks notebooks support language mixing using magic commands, such as:

- %python
- %sql
- %scala
- %r

This is very useful because I can write most of my logic in one language, like Python, and then switch to SQL for querying or use Scala for performance-critical tasks.

However, there are some limitations to be aware of when sharing data across languages in the same notebook:

- Data sharing is not automatic between languages. If I create a DataFrame in Python, I cannot directly access it in Scala or vice versa just by switching the cell language.
- To pass data between languages, I need to use intermediate storage or tables. For example:

    Write the DataFrame to a Delta table in Python and read it back in Scala using %scala.

    Register the DataFrame as a temporary view using .createOrReplaceTempView() and query it from %sql.

For example:

```
# Python cell

df = spark.read.csv("/mnt/data/sample.csv", header=True)

df.createOrReplaceTempView("my_temp_table")


// Scala cell

val dfScala = spark.sql("SELECT * FROM my_temp_table")
```

So, while multiple languages are supported in the same notebook, I must use shared storage or temporary views if I want to access data across those languages.

31. Is it possible to write code with VS Code and take advantage of all of its features, such as good syntax highlighting and IntelliSense?

Yes, it is definitely possible to write code for Databricks using Visual Studio Code (VS Code), and I can take full advantage of features like syntax highlighting, IntelliSense, linting, and Git integration.

To work effectively with Databricks from VS Code, I typically use:

1. **Databricks Connect**
   This allows me to run Spark code locally in VS Code, while the computations actually happen on a remote Databricks cluster. This way, I get local development experience with full IntelliSense and autocomplete while leveraging Databricks compute.
2. **Databricks VS Code Extension**
   Microsoft and Databricks provide a Databricks extension in the Visual Studio Code marketplace. With this, I can:
    1. Browse and edit Databricks notebooks
    2. Run cells directly from VS Code
    3. Upload or sync code between VS Code and the Databricks workspace
3. **Python and Jupyter extensions**
   These provide language-specific support, formatting, and Jupyter-style notebook development if I'm using .ipynb or .py scripts.

So yes, by connecting VS Code with Databricks, I can enjoy a full-featured development environment and maintain better version control and productivity.

32. Is there any way we can stop Databricks from establishing a connection to the internet?

Yes, we can limit or completely block outbound internet access from Databricks clusters using secure networking configurations, but it depends on the cloud setup and use case.

In Azure Databricks, this is done using:

1. No Public IP (NPIP) Clusters
   By configuring clusters without public IPs, all traffic goes through the organization's virtual network (VNet). This prevents the cluster nodes from accessing the public internet directly.
2. VNet Injection and Private Link
   I can deploy Azure Databricks into a VNet using VNet injection. Combined with Azure Private Link, this allows communication between services entirely within the Azure backbone network   no public internet exposure.
3. Network Security Groups (NSGs) and firewall rules
   These can be used to block or restrict outbound traffic from the data plane. For example, I can prevent traffic to non-approved destinations.
4. Custom DNS and Proxy Configuration
   Organizations can configure DNS and proxies to ensure that internet access is monitored or restricted through internal rules.

So yes, with the right architecture (especially using the Enhanced Security option in Azure Databricks), I can stop or tightly control internet access to meet compliance or security requirements.

33. What is Delta Lake and how does it enhance Azure Databricks?

Delta Lake is an open-source storage layer that brings ACID transactions, versioning, and data reliability to data lakes. It is one of the key technologies that makes Azure Databricks powerful for building modern data pipelines.

Here’s how Delta Lake enhances Azure Databricks:

1. **ACID Transactions**
   Delta Lake supports atomic, consistent, isolated, and durable operations. This means I can safely perform concurrent reads and writes without data corruption   even when multiple users or jobs access the same data.
2. **Schema Enforcement and Evolution**
   Delta automatically checks data against the defined schema before writing. I can also enable schema evolution to update the table schema as data changes.
3. **Time Travel**
   Delta stores a version history of data. I can query a previous version of a table by using a timestamp or version number. This helps in debugging, auditing, or rollback scenarios.

4. **Efficient Updates and Deletes**
   Unlike traditional data lakes, Delta supports operations like UPDATE, DELETE, and MERGE directly on large datasets, which is very helpful in ETL and data warehousing tasks.

5. **Faster Performance**
   Delta tables store data in Parquet format and use transaction logs (_delta_log) for faster reads and writes. It supports caching and indexing, improving performance significantly on large datasets.

6. **Streaming and Batch in One**
   With Delta Lake, I can use the same table for both batch and streaming jobs. This simplifies pipeline design and reduces complexity.

In short, Delta Lake turns a traditional data lake into a reliable, high-performance, transactional data store, which makes building and maintaining data pipelines in Databricks much easier and safer.

34. What is Delta Live Tables (DLT) and how is it used in Databricks?

Delta Live Tables (DLT) is a framework in Databricks for building reliable, automated, and maintainable data pipelines using simple declarative code. It is designed to simplify ETL (Extract, Transform, Load) processes by managing the complexity of data engineering tasks such as dependency resolution, data quality checks, and job orchestration.

Here's how I typically use Delta Live Tables in Databricks:

- I define data transformation logic in SQL or Python using decorators like @dlt.table.
- DLT automatically understands the dependencies between tables and orchestrates them in the correct order.
- It handles incremental processing, retries, and logging for me.
- I can define expectations (data quality rules) to validate data as it flows through each stage.
- The pipeline automatically creates managed Delta tables, tracks metadata, and logs lineage, which helps in monitoring and debugging.

For example, in Python, I can write:

```
@dlt.table

def clean_data():

  return spark.read.table("raw_data").filter("status = 'valid'")
```

DLT is used to build production-grade pipelines that are easier to manage and monitor than traditional notebooks or job scripts.

35. What are the different ways to load data in Databricks (e.g., COPY INTO, Autoloader)?

Databricks provides several methods to **ingest and load data**, depending on the use case, data format, and data volume. The main options include:

1. **COPY INTO**
   This SQL command is used to load data into Delta tables from files stored in cloud storage (like Azure Data Lake). It's great for batch loading from external sources and supports schema inference, file skipping, and idempotent behavior.

Example:

COPY INTO my_table

FROM '/mnt/data/raw/'

FILEFORMAT = PARQUET

2. **Auto Loader**
   Auto Loader is a feature for **incrementally and automatically** loading data as new files arrive in a cloud directory. It's designed for high-volume, real-time ingestion and supports both batch and streaming modes.
3. **Spark APIs (read/write)**
   I can use standard PySpark or Scala to load data using .read() methods from various sources like CSV, Parquet, JSON, JDBC, or Delta.
4. **Databricks REST APIs or Databricks CLI**
   These are useful for automating file uploads or triggering jobs that load data into tables.
5. **Data Ingestion Pipelines using Delta Live Tables (DLT)**
   With DLT, I define declarative pipelines that continuously ingest and transform data.
6. **Azure Data Factory or Synapse Pipelines**
   These external tools can be used to move data into Databricks using notebooks or REST API calls as activities in a pipeline.

Each method serves different needs for example, COPY INTO is good for manual or scheduled batch loads, while Auto Loader and DLT are great for continuous, large-scale ingestion.

36. What is Autoloader in Databricks and how does it support batch and streaming data?

Auto Loader is a Databricks feature for incremental, scalable, and schema-aware ingestion of new data files from cloud storage (like ADLS or S3).

Here's how it works and supports both batch and streaming:

- **File detection**: Auto Loader automatically monitors a cloud directory for new files.
- **Schema inference and evolution**: It can automatically detect the structure of incoming data and adapt to schema changes.
- **Streaming support**: It works with Spark Structured Streaming, so I can process files in real-time as they arrive.
- **Batch-like behavior**: Even though Auto Loader uses streaming under the hood, I can trigger it to run in **trigger once** mode, which behaves like batch   processing all available data and then stopping.
- **Checkpointing**: Auto Loader keeps track of what files have already been processed, so it doesn't reprocess the same data.

Example usage in PySpark:

```
df = (spark.readStream
    .format("cloudFiles")
    .option("cloudFiles.format", "json")
    .option("cloudFiles.schemaLocation", "/mnt/schema/")
    .load("/mnt/data/"))
```

In summary, Auto Loader makes data ingestion scalable, automated, and reliable, and it's perfect for use cases where new data is constantly being added to a storage location   like log files, sensor data, or external feeds.

37. What is the purpose of the coalesce() function, and how does it differ from repartition()?

The coalesce() function in Spark is used to reduce the number of partitions in a DataFrame. It is mostly used when we want to optimize performance before writing data to disk, especially when writing to formats like Parquet or Delta.

The main difference between coalesce() and repartition() is how they move data across partitions:

- coalesce(n) reduces the number of partitions without shuffling data across the network. It just merges adjacent partitions together. This makes it much faster and more efficient than repartition() if I'm only reducing partitions.
- repartition(n) can both increase or decrease the number of partitions. It always triggers a full shuffle of the data, meaning all the data is redistributed evenly across the new partitions. This is more expensive but ensures a more balanced distribution of data.

So I use coalesce() when I want to shrink partitions before writing output, and I use repartition() when I need to evenly distribute data (for example, before a join or groupBy).

38. What is the repartition() function, and how does it work in Spark?

The repartition() function in Spark is used to change the number of partitions in a DataFrame or RDD. It is helpful when I need to improve parallelism or rebalance data across the cluster.

Here’s how it works:

- When I call df.repartition(n), Spark performs a full shuffle of the data. That means it redistributes the rows across n partitions, trying to make them as even as possible.
- I can also use repartition(n, column) to partition the data based on a specific column. This is useful when doing operations like joins, where I want the same keys to be in the same partition.

For example:

df = df.repartition(10) # evenly spread across 10 partitions

df = df.repartition(5, "region") # repartition by the 'region' column

The shuffle makes repartition() more computationally expensive, but sometimes it's necessary to get better performance for downstream operations like joins or aggregations.

39. What are the drawbacks of using coalesce() in Spark?

While coalesce() is efficient for reducing partitions, it does have some limitations and drawbacks:

1. **Uneven Partition Sizes**
   Since coalesce() avoids shuffling, it may create unevenly sized partitions. This can cause performance problems because some tasks may process much more data than others, leading to skew and longer job runtimes.
2. **Only Reduces Partitions**
   coalesce() can only be used to reduce the number of partitions. If I try to increase the number, it won’t work properly or will result in unexpected behavior.
3. **Risk of Small File Problem**
   If I don’t coalesce enough before writing to storage, I might end up with too many small files. But if I coalesce too much, I might underutilize the cluster resources, especially when writing large datasets.
4. **Potential Bottleneck**
   If I reduce to a very small number of partitions (like 1), I may create a bottleneck, where a single partition is processing too much data. This hurts parallelism and job performance.

So, while coalesce() is a great tool for reducing partitions quickly and efficiently, I need to use it carefully to avoid creating performance imbalances or bottlenecks.

40. What causes out-of-memory issues in Spark, and how can they be resolved?

Out-of-memory (OOM) issues in Spark happen when an executor or driver runs out of available memory during processing. This can cause a task to fail, or even crash the whole application. Common causes and resolutions include:

Causes:

1. **Too Much Data in One Partition**
   If one partition holds a large portion of the data, that executor may run out of memory trying to process it.
2. **Improper Caching**
   If I cache or persist large datasets unnecessarily or forget to unpersist them, memory can get filled up quickly.
3. **Large Shuffles**
   When I perform operations like joins or groupBy on large datasets, Spark may need to shuffle a lot of data, which can use up memory.
4. **Inadequate Memory Allocation**
   If the executor or driver memory settings are too low for the job's workload, it may fail.
5. **Inefficient Code**
   Using collect(), broadcasting large datasets, or operations that require sorting in memory can also cause OOM.

Solutions:

- Repartition or coalesce data to better balance the workload.
- Increase executor memory in the Spark configuration (e.g., spark.executor.memory).
- Use efficient data formats like Parquet or Delta and apply column pruning and filtering early.
- Avoid using .collect() unless necessary; use .show() or .limit() for inspecting data.
- Use broadcast joins only when the broadcasted dataset is small.
- Use spillable transformations and avoid expensive aggregations without considering memory impact.
- Cache only what's needed and unpersist after use.

So, tuning partitioning, memory settings, and code logic are key to solving out-of-memory issues in Spark.

41. What is data skewness and how does it impact Spark performance?

Data skewness happens when data is unevenly distributed across partitions, especially in join or groupBy operations. Some partitions might have a lot of data, while others have very little. This uneven distribution causes performance problems.

Impact on Spark Performance:

- Some tasks finish quickly while others (with large partitions) take much longer, causing job delays.
- The cluster's parallelism is reduced, because many executors are sitting idle while one is overloaded.
- It can also cause out-of-memory errors if a single task gets too much data to process.

So, skewness can lead to inefficient resource usage and longer processing times.

42. What causes data skewness and what are the outcomes and solutions?

Causes of Data Skewness:

1. **High frequency of certain keys**
   For example, in a join or groupBy on a column where one value appears much more than others.
2. **Unbalanced input data**
   One file or partition may be much larger than others.
3. **Improper partitioning strategy**
   Using a poor hash function or default number of partitions can lead to imbalance.

Outcomes:

- Slower job execution.
- Task failures or timeouts.
- Executors running out of memory.
- Resource underutilization in the cluster.

Solutions:

1. **Salting**
   Add a random number to skewed keys before joins or aggregations to spread the data across more partitions.
2. **Skew Join Optimization** (Databricks-specific)
   Databricks provides automatic skew join handling using the spark.sql.adaptive.skewJoin.enabled setting.
3. **Repartitioning**
   Use repartition() on the skewed column to better distribute the data.
4. **Broadcast Join**
   If one side of the join is small, broadcast it to avoid shuffling the large dataset.
5. **Filter skewed keys separately**
   Process skewed keys in a different job or logic path to reduce their impact on the rest of the data.

43. How do you mount Azure Data Lake Storage to Databricks?

To access Azure Data Lake Storage (ADLS) directly from Databricks, I can mount the storage account to the Databricks File System (DBFS). Mounting makes the data available like a local directory, which simplifies reading and writing files.

Here's how I usually do it:

1. **Get Storage Credentials**
   I need:
   - The storage account name
   - The container name
   - Either a storage account key, service principal credentials, or a managed identity (for secure access)
2. **Use dbutils.fs.mount()** to mount the storage
   Here's an example using a service principal:

```
configs = {
  "fs.azure.account.auth.type": "OAuth",
  "fs.azure.account.oauth.provider.type":
"org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",
  "fs.azure.account.oauth2.client.id": "<application-id>",
  "fs.azure.account.oauth2.client.secret": "<application-secret>",
  "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/<tenant-
id>/oauth2/token"
}


dbutils.fs.mount(
  source = "abfss://<container-name>@<storage-account-name>.dfs.core.windows.net/",
  mount_point = "/mnt/<mount-name>",
  extra_configs = configs)
```

3. **Access Files**
   Once mounted, I can access files using:

```
spark.read.csv("/mnt/<mount-name>/data.csv")
```

Mounting is helpful when I need consistent, directory-style access across notebooks. But for secure, one-time access or production environments, using access credentials directly in read/write operations is often preferred.

44. What are the various optimization techniques in Spark?

There are many ways I optimize Spark jobs to improve performance, reduce memory usage, and make processing faster. Here are some of the key techniques I use:

1. **Partitioning Optimization**

    Use repartition() or coalesce() to optimize the number of partitions.

    Partition data by key columns to reduce shuffle during joins or aggregations.

2. **Caching and Persistence**

    Cache intermediate DataFrames when reused in multiple operations.

    Use unpersist() when the cache is no longer needed.

3. **Predicate Pushdown**

    Read only the data I need by applying filters early. Supported file formats like Parquet and Delta push filters to the source.

4. **Column Pruning**

    Select only the necessary columns instead of using select("*").

5. **Broadcast Joins**

    Broadcast small DataFrames to avoid large shuffles in joins.

6. **Use Delta Lake**

    Delta format provides faster reads and writes, and supports time travel and indexing.

7. **Avoid Wide Transformations**

    Use narrow transformations like map and filter instead of wide ones like groupByKey.

8. **Enable Adaptive Query Execution (AQE)**

    This allows Spark to adjust plans at runtime based on actual data.

    I set: spark.sql.adaptive.enabled = true

9. **Skew Join Optimization**

    Handle skewed keys using salting or enable skew join optimization in Databricks.

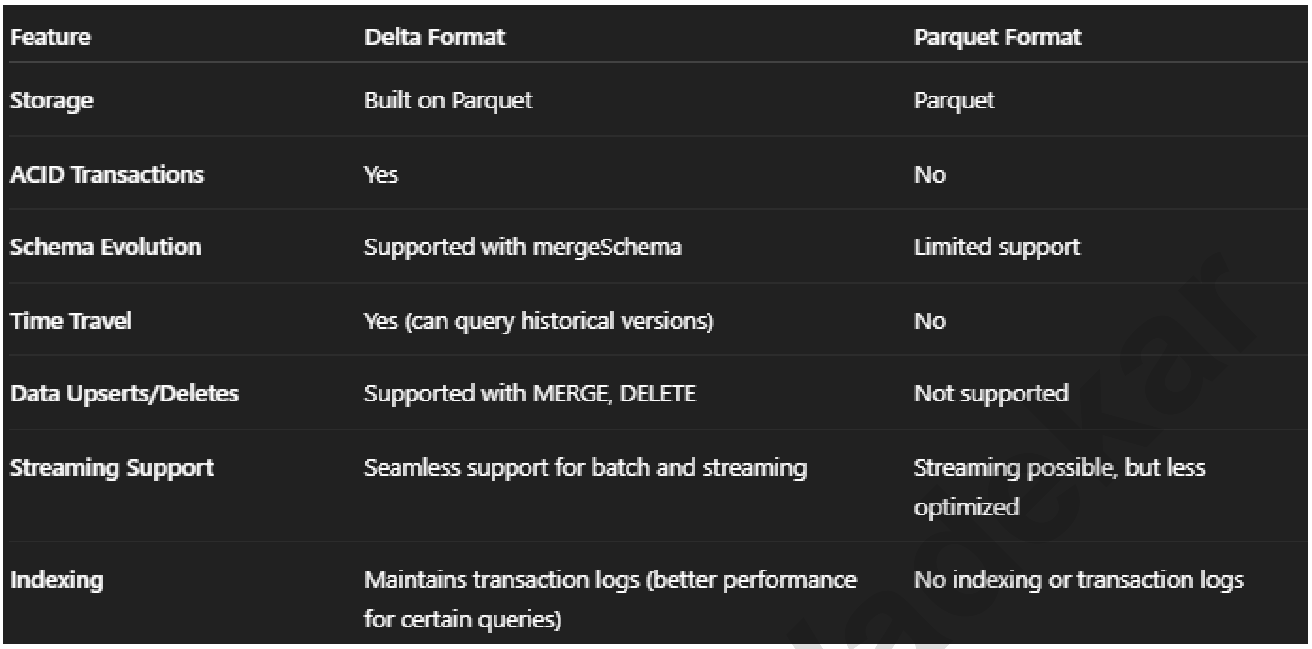
10. **Use Efficient File Formats**

    Parquet and Delta are columnar and compressed, making them faster than CSV or JSON.

By combining these techniques, I can make Spark jobs much faster and more efficient.

45. What is the difference between Delta format and Parquet format?

Delta and Parquet are both columnar storage formats, but Delta provides many advanced features built on top of Parquet. Here's how they differ:



Feature	Delta Format	Parquet Format
Storage	Built on Parquet	Parquet
ACID Transactions	Yes	No
Schema Evolution	Supported with mergeSchema	Limited support
Time Travel	Yes (can query historical versions)	No
Data Upserts/Deletes	Supported with MERGE, DELETE	Not supported
Streaming Support	Seamless support for batch and streaming	Streaming possible, but less optimized
Indexing	Maintains transaction logs (better performance for certain queries)	No indexing or transaction logs

46. What is the Databricks Unity Catalog and what are its capabilities?

The Databricks Unity Catalog is a unified governance solution for all data and AI assets within the Databricks Lakehouse Platform. It provides fine-grained access control, centralized metadata management, and auditability across workspaces, making it easier to manage data securely and at scale.

Key capabilities include:

- **Centralized Metadata Management**: It manages metadata (like table schemas, views, volumes) across all Databricks workspaces from a central catalog.
- **Fine-Grained Access Control**: Uses ANSI SQL-standard GRANT statements to define access at the table, row, column, and view levels. It integrates with Identity Providers like Azure AD for RBAC.
- **Data Lineage**: Automatically tracks lineage for tables, views, and notebooks, enabling data teams to understand data flow and dependencies.
- **Support for Multiple Data Assets**: Unity Catalog supports Delta Tables, ML models, notebooks, volumes, and files in object storage.
- **Audit and Compliance**: Logs all access to data for auditing purposes, helping organizations meet compliance requirements.
- **Multi-Tenant Architecture**: Enables secure sharing of data across different business units and teams within or across organizations.

47. What are persistence levels in Spark and how do you choose between them?

In Apache Spark, persistence levels determine how RDDs or DataFrames are stored in memory or disk across transformations. The most common persistence levels are defined in the StorageLevel class.

Some key persistence levels include:

- MEMORY_ONLY: Stores RDD as deserialized Java objects in memory. If not enough memory, recomputes on access.
- MEMORY_AND_DISK: Tries to store in memory, spills to disk if not enough memory.
- DISK_ONLY: Stores data only on disk.
- MEMORY_ONLY_SER / MEMORY_AND_DISK_SER: Stores data as serialized objects to save space.
- OFF_HEAP: Uses off-heap memory for storage (requires Tungsten and off-heap memory configuration).

How to choose:

- If data fits comfortably in memory and you prioritize speed → MEMORY_ONLY.
- If data doesn't fit in memory but you want fault-tolerance → MEMORY_AND_DISK.
- For large datasets that don't need fast access → DISK_ONLY.
- If memory is limited and GC overhead is high → Use serialized levels like MEMORY_ONLY_SER.
- For highly optimized use cases in large clusters → OFF_HEAP (requires tuning and setup).

In practice, I use MEMORY_AND_DISK by default during iterative computations like machine learning, and switch to other levels based on performance profiling.

48. What is shuffle in Spark and what causes it?

Shuffle in Spark refers to the re-distribution of data across partitions, often across nodes, that occurs when operations require data to be moved, such as aggregations or joins.

Shuffle is caused by wide transformations, which are transformations where output depends on data from multiple partitions. Some examples include:

- groupByKey, reduceByKey, join, distinct, repartition, and sortBy.

What causes shuffle:

- Joins between large datasets.
- GroupBy / Aggregations that need to bring all values of a key to a single executor.
- Repartitioning to increase or decrease the number of partitions.
- Sorting operations that require global ordering.

Why shuffle is important:

- Shuffle is expensive it involves disk I/O, data serialization, and network transfer.
- Excessive shuffle can lead to performance bottlenecks, skew, and even out-of-memory errors.

Optimization tips:

- Use reduceByKey instead of groupByKey to reduce shuffle size.
- Use broadcast joins when one table is small.
- Avoid unnecessary repartitioning.
- Monitor shuffle size and stage duration in the Spark UI.

49. What are wide transformations in Spark? Provide examples.

Wide transformations are those operations in Spark where data needs to be moved between different partitions, and possibly across machines. This movement of data is known as a shuffle. These transformations usually take more time and resources because of the data transfer involved.

In simple terms, if the operation needs data from multiple partitions to complete, it's a wide transformation.

Some common examples of wide transformations are:

- groupByKey(): Groups data by key and brings values with the same key together from different partitions.
- reduceByKey(): Reduces values by key and also involves shuffling of data.
- join(): Combines rows from two datasets based on a common key. Spark needs to move data so matching keys are on the same partition.
- distinct(): Removes duplicate rows, which requires comparing all rows across partitions.
- repartition(): Redistributes the data into a new number of partitions, which causes a full shuffle.

These transformations are more expensive, so I try to minimize their use or optimize them using techniques like broadcast joins or using reduceByKey instead of groupByKey when possible.

50. What are narrow transformations in Spark? Provide examples.

Narrow transformations are the operations where each partition of the input data is used by only one partition of the output. In other words, there's no shuffling or data movement between nodes. These operations are generally faster and more efficient.

Examples of narrow transformations include:

- map(): Applies a function to each row without changing the partitioning.
- filter(): Keeps only the rows that meet a condition, within the same partition.
- flatMap(): Similar to map, but returns zero or more output elements for each input row.
- union(): Combines two datasets without shuffling.
- sample(): Randomly samples data but doesn't cause a shuffle.

Since narrow transformations are more performance-friendly, they are preferred when I want to process data quickly without involving any expensive shuffling.

51. What is the difference between sort() and orderBy() in Spark?

Both sort() and orderBy() are used to sort data in Spark, but there are a few differences:

- sort() is mostly used when working with DataFrames or Datasets and is more optimized. If we don't provide any column, it uses natural ordering. It can be used in both global and per-partition sorting depending on the context.
- orderBy() always performs a full shuffle to guarantee total ordering across all partitions. It is more strict and guarantees the final output is sorted globally.

The main difference is performance:

- sort() can sometimes avoid a full shuffle if used smartly, like after repartitioning.
- orderBy() always performs a full shuffle and is generally more expensive.

So, if I need global ordering and final output to be perfectly sorted, I use orderBy(). But if I just want local sorting or want to control performance, I prefer sort().

52. What is the difference between distinct() and dropDuplicates() in Spark?

Both distinct() and dropDuplicates() are used to remove duplicates from a DataFrame, but there is a key difference:

- distinct() removes duplicate rows by considering all columns. It checks for full row duplication.
- dropDuplicates() gives us more flexibility. We can pass one or more specific columns to consider for checking duplicates. If we don't pass any column names, it behaves exactly like distinct().

For example:

```
df.dropDuplicates(["name", "city"])
```

This will remove duplicates based only on name and city columns.

In most of my projects, I use dropDuplicates() when I want to remove duplicates based on some business keys or important columns, and I use distinct() when I want to remove complete row-level duplicates.

53. What is whole-stage code generation in Spark?

Whole-stage code generation is a performance optimization technique used by Spark to speed up the execution of queries. It works by converting a sequence of Spark transformations into a single block of Java code at runtime.

Instead of running each transformation like filter, project, or join separately, Spark compiles them into one piece of optimized code that processes multiple rows at once, which reduces the overhead of function calls and improves CPU usage.

This technique is especially helpful for operations on DataFrames and Datasets, because Spark can understand the structure of the data and plan the execution more efficiently.

From my experience, whole-stage code generation leads to better performance, especially for queries with multiple operations in a row, like filtering, aggregating, and selecting.

54. What is lazy evaluation in Spark and how does it benefit performance?

Lazy evaluation means that Spark doesn't execute a transformation as soon as it's defined. Instead, it builds up a logical plan of all transformations and only runs them when an action is triggered.

For example, if I write code with multiple filter() and select() statements, Spark will just remember those steps but won't process any data until I call an action like show(), count(), or collect().

The benefit of lazy evaluation is that Spark can optimize the execution plan before running it. It can rearrange operations, combine steps, or skip unnecessary work, which improves overall performance.

This concept is very powerful because it helps avoid repeated computations and ensures Spark only reads and processes the data when needed.

55. What are SQL Endpoints in Databricks?

SQL Endpoints in Databricks are compute resources used specifically for running SQL queries. They are part of Databricks SQL, which is the interface designed for analysts and BI tools to query data in the Lakehouse using SQL.

An SQL Endpoint is like a cluster, but it is optimized for running SQL workloads. It supports features like:

- Auto-scaling of clusters
- Integration with BI tools like Power BI and Tableau
- Query caching for faster performance
- Secure access and governance using Unity Catalog

In simple terms, when I want to run pure SQL queries on Databricks, especially from a dashboard or BI tool, I use a SQL Endpoint rather than a notebook or job cluster.

56. What is the difference between a Databricks job cluster and an interactive cluster?

The main difference lies in how and when the clusters are used:

- **Job Cluster**: This is a temporary cluster that gets created when a job runs and is terminated when the job finishes. It is used for scheduled or automated tasks like ETL pipelines or production jobs. It ensures fresh resources for every run and is cost-effective because it only runs during job execution.
- **Interactive Cluster**: This is a manually created cluster that stays alive and is mainly used for development, exploration, and testing. It allows users to run notebooks, write code interactively, and view results immediately.

In my day-to-day work:

- I use interactive clusters while developing and testing code in notebooks.
- Once the code is ready for production, I schedule it using Databricks Jobs, which automatically run on job clusters.

This separation helps balance performance, cost, and collaboration.

57. What is a managed table in Databricks?

A managed table in Databricks is a table where both the metadata and the actual data are managed by Databricks itself. When I create a managed table using a CREATE TABLE or saveAsTable() command without specifying a location, Spark automatically stores the data in the default location inside the Databricks-managed storage.

The main point is that when I drop a managed table, both the table metadata and the underlying data files are deleted permanently.

This is useful when I want Databricks to fully manage the lifecycle of the data and I don't need the data to be shared outside of the Lakehouse.

58. What is an unmanaged table in Databricks?

An unmanaged table (also called an external table) is a table where the data is stored outside the default Databricks storage location, and Databricks only manages the metadata.

When I create an unmanaged table, I specify the path where the data already exists, like in a mounted ADLS folder or external location. So even if I drop the table, the actual data files remain safe and only the metadata is removed from the metastore.

I usually use unmanaged tables when:

- The data is shared across multiple platforms or systems.
- I don't want to risk losing the data by dropping the table.
- I want full control over where the data is stored.

59. How do you configure the number of cores in a worker in Databricks?

In Databricks, I don't directly set the number of cores per worker. Instead, I select the instance type when configuring a cluster. Each instance type (like Standard_D4s_v3 or i3.xlarge) comes with a fixed number of vCPUs (cores) and memory.

So to configure the number of cores in a worker node, I:

1. Go to the cluster configuration page.
2. Choose the worker node type under the cluster settings.
3. Pick an instance that has the desired number of cores.

If I need more parallelism or processing power, I either:

- Increase the number of worker nodes, or
- Choose a larger instance with more cores per node.

Also, I can use spark.default.parallelism or spark.sql.shuffle.partitions to fine-tune how Spark uses those cores during processing.

60. How do you handle bad records in Databricks during ingestion?

When I ingest data, especially from raw sources like CSV or JSON, bad records can occur due to corrupted rows, data type mismatches, missing values, or unexpected formats.

In Databricks, I handle bad records in the following ways:

1. **Using the badRecordsPath option**:
   While reading data using spark.read, I can set badRecordsPath to save all problematic rows to a separate location for later inspection.
   Example:

```
spark.read.option("badRecordsPath", "/mnt/errorRecords").csv("/mnt/input")
```

2. **Using mode option**:

   PERMISSIVE (default): Keeps good records and puts nulls for corrupt ones.

   DROPMALFORMED: Drops the bad records completely.

   FAILFAST: Fails immediately if a bad record is found.
   Example:

```
spark.read.option("mode", "DROPMALFORMED").csv("/mnt/input")
```

3. **Schema validation and casting**:
   I define the expected schema during read to avoid incorrect automatic inference. That way, I can catch mismatches early.

4. **Logging and alerting**:
   I usually log the count of bad records and notify the team if the number exceeds a threshold, especially in production pipelines.

This approach helps me ensure data quality without stopping the pipeline, and allows me to fix issues with bad data separately.

61. What does the map() transformation do in RDDs?

The map() transformation in RDDs is used to apply a function to each element of the RDD and return a new RDD with the transformed values.

It processes data element by element. For example, if I have an RDD of numbers and I use map() to multiply each number by 2, it will go through each element and apply that logic.

Example:

```
rdd = sc.parallelize([1, 2, 3])

rdd2 = rdd.map(lambda x: x * 2)

# rdd2 will have [2, 4, 6]
```

It is useful when I want to perform row-level transformations like converting strings to uppercase, applying calculations, or extracting values from a tuple.

62. What does the mapPartitions() transformation do in RDDs, and when is it useful?

The mapPartitions() transformation is similar to map(), but instead of processing one element at a time, it processes an entire partition at once. It gives me more control and can be more efficient in some situations.

In mapPartitions(), I write a function that operates on an iterator over the partition's data, which means I can open a database connection or use external resources just once per partition instead of per element.

Example:

```
def process_partition(iterator):

  return (x * 2 for x in iterator)

rdd2 = rdd.mapPartitions(process_partition)
```

It's useful when:

- I want to reduce the overhead of resource creation per element.
- I need to perform operations that benefit from batch processing.
- I'm doing things like writing to a database or API where connection reuse is better.

I use it carefully because if something goes wrong in one partition, all elements in that partition could be lost.

63. What is an accumulator in Spark and how is it used?

An accumulator in Spark is a variable used for summing up values or collecting metrics across all tasks in a distributed way, especially during transformations.

It is mainly used for side-effect operations like counting the number of errors, tracking how many records meet a condition, or logging metrics. The value of the accumulator can be updated in workers, but it can only be read on the driver.

Example:

```
acc = sc.accumulator(0)


def count_even(x):
  if x % 2 == 0:
    acc.add(1)
  return x


rdd.map(count_even).collect()
print("Even numbers:", acc.value)
```

One important point is that accumulators are **not reliable for program logic**, because Spark might run the same task more than once (due to retries), which can cause over-counting.

64. What is a broadcast variable and why is it useful in Spark?

A broadcast variable in Spark is a read-only variable that is cached on each executor so that it doesn't need to be sent again with every task.

Normally, when I use a large lookup table or a small reference dataset in my transformation logic, Spark sends a copy of it to each task, which is inefficient. A broadcast variable avoids that by distributing the data only once to each node.

Example:

```
broadcastVar = sc.broadcast({"CA": "California", "NY": "New York"})


def map_state(abbr):
  return broadcastVar.value.get(abbr, "Unknown")


rdd.map(map_state)
```

Broadcast variables are very useful in joins where one dataset is small. Instead of doing a regular join that causes shuffle, I broadcast the small dataset and do a map-side join, which is much faster and avoids unnecessary data movement.

65. What is the default number of shuffle partitions in Spark?

In Spark, the default number of shuffle partitions is 200.

This means whenever a wide transformation happens like groupBy, join, or reduceByKey Spark will divide the shuffled data into 200 partitions by default.

This default works well for small to medium datasets, but for larger datasets, I usually increase this number to ensure better parallelism. For smaller datasets, I sometimes reduce it to avoid creating too many small tasks that slow down processing.

I can change it using the configuration:

```
spark.conf.set("spark.sql.shuffle.partitions", 100)
```

Tuning this value is one of the common steps in performance optimization for Spark jobs.

66. What is the default partition size in Spark, and how is it calculated?

Spark doesn't have a hard-coded default "partition size" in terms of file size like 128MB, but there are a few common practices and internal heuristics.

When reading files (especially from Parquet or CSV), Spark tries to create partitions that are around 128MB by default for each split, based on the file size. This value comes from Hadoop configuration, and Spark inherits it unless overridden.

So, if I have a 1GB file, Spark will try to create roughly 8 partitions (1GB / 128MB). This is done to balance the load across the cluster.

However, the number of partitions also depends on:

- Number of cores
- Input file format and block size
- Manual settings using .repartition() or .coalesce()

I can override the default block size by setting spark.sql.files.maxPartitionBytes, for example:

```
spark.conf.set("spark.sql.files.maxPartitionBytes", 134217728)  # 128MB
```

This is useful when I want to control how data is split and distributed for better parallelism or memory usage.

67. How are zip files distributed across nodes in Spark?

Spark doesn't natively split compressed files like ZIP or GZIP across multiple nodes for parallel processing, because most compression formats are not splittable. This means a single ZIP file will be processed by a single executor, no matter how big it is.

In practice, this limits performance, especially with large zip files, because Spark can't divide the contents across workers.

To work around this:

- I extract zip files beforehand in storage (like ADLS or S3) before loading into Spark.
- If I must read compressed files in Spark, I prefer splittable formats like bzip2 or use file formats like Parquet, which are optimized for parallel reads.

So in summary, Spark will send the whole zip file to one node, which may cause memory and speed issues if the file is large.

68. What is the difference between from pyspark.sql.types import * and from pyspark.sql.types import XYZ?

Both are import styles, but the main difference is how much and what you're importing from the pyspark.sql.types module.

- from pyspark.sql.types import *
  This imports everything from pyspark.sql.types, including types like StringType, IntegerType, StructType, etc.
  It's convenient, but not recommended for production code because it can make the code less readable and might cause name conflicts.
- from pyspark.sql.types import StringType, IntegerType
  This is more explicit and cleaner. It makes it clear which types I'm using in the code and avoids confusion. It's considered best practice in team projects or production pipelines.

For example:

# Not recommended

from pyspark.sql.types import *

# Recommended

from pyspark.sql.types import StructType, StructField, StringType, IntegerType

I usually prefer the second option, especially when working in collaborative environments or writing reusable code.

69. How can you combine data into a single output file in Databricks?

In Databricks, when we write a DataFrame to a file system like ADLS or DBFS, Spark by default creates multiple output files one per partition. To combine everything into a single output file, I use the coalesce(1) transformation before writing the data.

Example:

df.coalesce(1).write.mode("overwrite").csv("/mnt/output")

This combines all the partitions into a single one and writes just one file. However, it's important to note that this is suitable only for small to medium datasets, because combining everything into one partition can cause performance issues for large data.

For large datasets, it’s better to write multiple files and then combine them using tools like Azure Data Factory or a post-processing script.

70. What are Databricks widgets and how are they used in notebooks?

Databricks widgets are input fields like text boxes, dropdowns, or multiselects that I can create inside a notebook to make it interactive. They allow users to provide input values without editing the code directly.

I use widgets to parameterize notebooks for example, to pass dates, file paths, or environment names dynamically.

Example:

dbutils.widgets.text("input_date", "2024-01-01", "Date")

date_value = dbutils.widgets.get("input_date")

Widgets are especially useful when:

- Running the same notebook with different inputs
- Scheduling notebooks using Databricks Jobs
- Reusing notebooks across different environments (like dev and prod)

They help make notebooks cleaner and easier to use for non-technical users too.

71. How can you optimize the performance of Spark jobs in Databricks?

There are several ways I optimize Spark job performance in Databricks, depending on the type of workload. Some of the most effective strategies include:

1. **Partition tuning**:

    Set the right number of partitions using repartition() or coalesce().

    Adjust spark.sql.shuffle.partitions to match the size of the data and number of cores.

2. **Broadcast joins**:

   Use broadcast joins when one of the tables is small to avoid shuffle.

```
from pyspark.sql.functions import broadcast
df.join(broadcast(small_df), "id")
```

3. **Caching and persistence**:

   Cache intermediate data if reused multiple times.

```
df.cache()
```

4. **Column pruning and predicate pushdown**:

   Read only the columns and rows needed by applying filters early.

   Supported well with formats like Parquet.

5. **Choosing the right file format**:

   Use Parquet or Delta instead of CSV or JSON for better compression and performance.

6. **Delta Lake optimizations**:

   Use OPTIMIZE, ZORDER, and VACUUM for managing Delta tables efficiently.

7. **Cluster sizing**:

   Use auto-scaling and right-sized instances for the workload.

   Avoid over-provisioning, especially for job clusters.

By combining these techniques, I've seen significant improvements in performance and cost-efficiency in my Spark pipelines.

72. How do you manage and version control notebooks in Azure Databricks?

In Azure Databricks, there are a few ways I manage and version control notebooks:

1. **Using Git integration**:
   Databricks supports direct integration with Git providers like GitHub and Azure DevOps. I link my notebook to a Git repo and manage version control using standard Git operations like commit, push, pull, and branches.

Steps:

   In the notebook, click on "Revision History" → "Link to Git".

   Choose the provider and enter the repo details.

   After linking, I can commit and push changes directly from the UI.

2. **Using notebook revision history**:
   Databricks automatically keeps a revision history for each notebook. I can view, compare, and restore previous versions. This is helpful for tracking changes over time.

3. **Exporting notebooks**:
   I can export notebooks as .dbc, .ipynb, or .py files and store them in source control manually if Git integration is not enabled.

4. **Using Repos**:
   Databricks Repos is a feature that allows me to clone entire Git repositories directly into the workspace. This keeps notebooks in sync with the Git repo and supports collaborative development.

Version control is very important for collaboration, rollback, and change tracking, especially when working in a team or deploying production code.

73. What are the different types of clusters in Azure Databricks?

In Azure Databricks, there are mainly two types of clusters:

1. **Interactive Cluster (All-purpose cluster)**
   This cluster is used for developing, running, and testing code in notebooks. It stays active until manually terminated, which makes it suitable for interactive data exploration, visualization, or collaboration.

I use this type when:

- I'm developing a new pipeline.
- I need to test different transformations.
- I'm working with teammates on a shared notebook.

2. **Job Cluster**
   This is a temporary cluster created automatically when a Databricks job is run. It starts with the job and shuts down after the job finishes. It's ideal for scheduled, production workloads.

I prefer job clusters for:

- ETL pipelines
- Scheduled batch jobs
- Production code that doesn't need manual intervention

Using job clusters is more cost-effective and scalable for automated workloads, while interactive clusters are better for exploratory work.

74. How do Spark tasks and stages break down when two datasets with different partitions are joined?

When two datasets with different numbers of partitions are joined, Spark first checks the partitioning schemes of both datasets. If they don't align, Spark performs a shuffle operation to redistribute the data so that rows with matching keys end up on the same partition.

Here's how Spark breaks it down:

1. Stages:
   The job is broken into stages separated by shuffle boundaries. Each stage is a group of tasks that can run in parallel without shuffling.
2. Tasks:
   Each stage is further divided into tasks, and each task operates on one partition.

If dataset A has 200 partitions and dataset B has 50, Spark needs to shuffle one or both datasets to a common number of partitions (usually 200) so that each task in the join stage works on matching partitioned data.

This repartitioning and shuffling step increases the job complexity and time, so I try to:

- Repartition both datasets to the same number of partitions before the join.
- Use broadcast joins if one dataset is small.

75. What are the various join strategies available in Spark and when should each be used?

Spark supports several join strategies, and the best one depends on the size of the datasets and the cluster configuration. The main join strategies are:

1. **Shuffle Hash Join**
   This is the default join when both datasets are large. Spark performs a full shuffle of both datasets and builds a hash table on one side.

Use this when:

Both datasets are large.

No broadcast is possible.

2. **Broadcast Hash Join**
   This is the most efficient join when one dataset is small enough to fit in memory. Spark sends the smaller dataset to all executors.

Use this when:

One dataset is small (usually < 10MB to 100MB).

You want to avoid shuffling large data.

Example

df_large.join(broadcast(df_small), "id")

3. **Sort Merge Join**
   Used when both datasets are large and sorted on the join keys. Spark will sort both datasets and then merge them.

Use this when:

Join keys are already sorted.

Datasets are too large for broadcasting.

4. **Cartesian Join**
   Joins every row from one dataset with every row in another. This is very expensive and should be avoided unless explicitly needed.

Use this only:

When there is no join condition.

For special use cases like cross product scenarios.

In my projects, I mostly use broadcast joins for lookup tables and sort-merge joins for joining large fact and dimension tables in star schemas.

76. What are Databricks notebooks and how are they used?

Databricks notebooks are interactive web-based environments where I can write and run code in multiple languages like Python, SQL, Scala, or R, all in the same notebook. They are similar to Jupyter notebooks but are optimized for use with the Databricks platform.

They are used for:

- Writing and testing code interactively
- Building and running data pipelines
- Creating visualizations from queries
- Collaborating with other team members
- Parameterizing workflows using widgets
- Scheduling jobs or developing prototypes

Features I find very useful:

- **Cell-based execution**: I can run one block of code at a time.
- **Built-in visualizations**: Like bar charts or line graphs with one click.
- **Collaboration tools**: Like comments and revision history.
- **Git integration**: To track code changes and manage versions.

Overall, notebooks make it easy for me to develop, document, and share code, especially when working with cross-functional teams like data analysts, engineers, and scientists.

77. What is data caching in Spark and when should you use CACHE TABLE vs persist()?

Data caching in Spark is a technique used to store intermediate results in memory (or memory + disk) so that they can be reused across multiple actions without being recomputed. This is very helpful when the same dataset is used multiple times in a job, like in iterative algorithms or reporting tasks.

There are two common ways to cache data:

1. **CACHE TABLE**
   This is used when I want to cache a table in SQL. It stores the table in memory for fast access.

CACHE TABLE my_table;

I use this when working with SQL queries or when I want to cache a table registered in the metastore. It is simple and useful when running multiple SQL queries on the same table.

2. **persist()**
   This is more flexible and is used with DataFrames or RDDs in code. It allows me to choose the storage level, like MEMORY_ONLY, MEMORY_AND_DISK, etc.

df.persist(StorageLevel.MEMORY_AND_DISK)

I use persist() when:

I want to control how and where data is stored.

I'm working with large datasets that may not fit entirely in memory.

In general, I use CACHE TABLE for SQL workloads and persist() when I need more control or I'm working in Python/Scala code with DataFrames.

78. What steps can be taken to avoid data loss in Databricks?

To avoid data loss in Databricks, I follow these best practices:

1. **Use Delta Lake**
   Delta Lake provides ACID transactions, data versioning, and rollback capabilities. This prevents accidental data overwrites or corruption.
2. **Write to external storage**
   I always store data in durable external storage like ADLS Gen2, S3, or mounted containers, not in temporary DBFS paths.
3. **Enable Auto Loader checkpoints**
   When using Auto Loader for streaming, I configure checkpoint locations to keep track of processed files and prevent duplicates or data loss.
4. **Use overwrite mode carefully**
   I avoid overwriting tables or files unless absolutely necessary. When I do, I use overwriteSchema and partition handling options cautiously.
5. **Enable versioning and backups**
   If using Delta Lake, I can access previous versions of data using time travel. I also automate backups of critical tables by copying data or exporting snapshots.
6. **Cluster auto-termination**
   I set up auto-termination to avoid accidental costs, but I make sure to save important results or checkpoints before the cluster shuts down.
7. **Use notebooks in Repos**
   I version-control notebooks using Git integration so that my code and logic are not lost due to accidental changes or deletions.

79. What is the difference between Synapse and Databricks in terms of Spark usage?

Both Synapse and Databricks support Spark, but they differ in how they use and manage Spark:

1. **Spark Integration**

   Databricks: Built completely on Apache Spark. It offers a full-featured, optimized Spark engine with advanced features like Delta Lake, MLflow, and Unity Catalog.

   Synapse: Offers Apache Spark pools as one of many compute engines. It integrates Spark but also supports T-SQL, pipelines, and Power BI integration.
2. **User Experience**

   Databricks: Has a very rich notebook interface that supports Python, Scala, SQL, and R. Great for data engineers, data scientists, and collaborative development.

   Synapse: More suited to data analysts and BI teams. It integrates SQL and Spark, but the Spark experience is more limited compared to Databricks.
3. **Cluster Management**

   Databricks: Offers auto-scaling clusters, job clusters, and interactive clusters.

   Synapse: Has fixed-size Spark pools that must be pre-created, with less flexibility in dynamic scaling.

4. **Use Cases**

    Databricks: Better for advanced data engineering, machine learning, real-time streaming, and big data processing.

    Synapse: Good for data warehousing, simple data transformations, and BI reporting.

In short, I prefer Databricks when Spark is a core part of the solution, and I choose Synapse when Spark is only needed occasionally alongside SQL and other services

80. What are the main components of the Azure Databricks architecture?

Azure Databricks architecture is made up of several components that work together to provide a scalable and collaborative data platform. The key components are:

1. **Azure Databricks Workspace**
   This is the user interface where I can access notebooks, jobs, clusters, repos, and data. It's the main environment for collaboration.

2. **Clusters (Compute)**
   Databricks uses Apache Spark clusters to process data. There are two main types: interactive clusters and job clusters. These run on Azure virtual machines and can autoscale.

3. **Databricks Runtime**
   This is an optimized Spark environment managed by Databricks. It includes built-in support for Delta Lake, ML libraries, and performance enhancements.

4. **Delta Lake Storage Layer**
   Databricks uses Delta Lake as the default storage layer to provide ACID transactions, schema enforcement, and time travel on top of data lakes like ADLS.

5. **Azure Services Integration**
   Databricks tightly integrates with Azure services like:

    Azure Data Lake Storage (ADLS) for storing data

    Azure Key Vault for secure secrets

    Azure Synapse and SQL DB for data warehousing

    Azure ML and Power BI for machine learning and reporting

6. **Control Plane and Data Plane**

    Control Plane is managed by Databricks and handles UI, job scheduling, notebooks, and metadata.

    Data Plane runs inside the customer's Azure environment and is where the actual Spark jobs execute and data is processed.

This separation ensures data security while still allowing Databricks to manage the platform efficiently.

81. What are the common use cases for Azure Databricks?

In my experience, Azure Databricks is quite flexible and supports a wide range of use cases across data engineering, analytics, and machine learning. One of the most common use cases is building ETL pipelines. I've used it to process large volumes of raw data from data lakes, clean it, and transform it into usable formats, often using Delta Lake.

Another major use case is advanced analytics and machine learning. Since Databricks supports collaborative notebooks and integrates well with MLflow, it's easy to build, train, and deploy machine learning models, all within the platform.

It's also commonly used for real-time data processing. For example, I've worked on projects where we used Databricks Structured Streaming to process live telemetry or log data and make it available almost instantly for analysis.

And finally, Databricks is often used for running large-scale SQL analytics. Databricks SQL makes it easier for analysts and business users to query data directly from the data lake without needing to move it to a warehouse.

82. What are some of the primary tools integrated with Databricks?

Databricks integrates with a lot of tools, which is one of the reasons it's so powerful. On the Azure side, it integrates really well with Azure Data Lake Storage (ADLS), which I usually use to store both raw and processed data. It also works seamlessly with Azure Key Vault for managing secrets like connection strings securely.

For data ingestion, Databricks supports tools like Kafka, Event Hubs, and Auto Loader, which is great for both batch and streaming data. On the visualization side, I often use Power BI with Databricks because there's native integration that lets me connect and build dashboards directly on top of Delta tables.

And then for version control, Databricks Repos can connect to GitHub or Azure DevOps, so my team can follow proper development workflows. MLflow is built in for experiment tracking and model management, which I find extremely helpful when working with machine learning projects.

83. What are Lakehouse architectures and how do they relate to Databricks?

So, a Lakehouse architecture is basically a combination of a data lake and a data warehouse. Traditionally, data lakes are great for storing raw, unstructured data, and data warehouses are used for structured data and analytics. But maintaining both can be expensive and complex.

The lakehouse tries to bring the best of both worlds together. With the lakehouse, we can store all kinds of data in one place (usually in a data lake like ADLS), and still run SQL queries, analytics, and even machine learning on top of it.

Databricks is actually one of the main platforms that popularized the Lakehouse concept. It uses Delta Lake as the storage format, which adds features like ACID transactions, versioning, and schema enforcement on top of the data lake. That means we can build reliable, production-grade pipelines directly on top of raw data, without needing to move it into a separate warehouse.

In short, Databricks gives us the tools to implement a lakehouse architecture effectively whether it's ETL, BI, or machine learning, it can all happen in the same unified platform.

84. How does Databricks handle data governance?

Databricks handles data governance mainly through something called Unity Catalog. From my experience, this makes managing access to data across different teams and workspaces much easier and more secure.

With Unity Catalog, we can set up fine-grained permissions at the table, column, or even row level. So if one team should only see specific data, I can enforce that with access controls. What's great is that these controls are centralized, so I don't have to set up different rules in each workspace it's all in one place.

Another big plus is data lineage. Unity Catalog tracks how data is used and transformed over time. So if someone changes a table or uses it in a report, we can trace back where it came from and who accessed it. That helps a lot for audits or debugging.

Also, Databricks integrates well with Azure Active Directory, which means I can manage users and permissions based on the company's existing roles and groups. And for sensitive data like PII, I can tag and classify columns so we know which ones need extra protection.

So overall, I'd say Databricks has matured a lot in terms of governance, and Unity Catalog plays a central role in that.

85. What is the function of the Databricks Repos?

Databricks Repos is basically the feature that lets me connect my notebooks and code directly to a Git repository like GitHub, Azure DevOps, or Bitbucket right from within the Databricks workspace.

This has been really helpful in team environments. It allows us to version control our notebooks just like we do with normal software projects. We can work on different branches, raise pull requests, and merge changes safely.

It's especially useful for collaboration. Before Repos, people used to overwrite each other's notebooks or lose track of changes. Now, we commit changes, push to Git, and everything's traceable. I've also used it in CI/CD pipelines where notebook deployments are part of an automated workflow.

And it's not limited to notebooks. We can also store .py files, configuration files like .yaml, and reuse Python modules across projects. For me, Repos bridge the gap between notebook development and proper software engineering practices.

86. What are Databricks Jobs and how do you schedule them?

Databricks Jobs are how I automate things like ETL pipelines, model training, or report generation. Instead of running a notebook manually, I can schedule it to run automatically using a job.

When I create a job, I choose what task to run it could be a notebook, a Python script, or even a JAR file. I assign a cluster for it to run on, and then set up a schedule, kind of like using cron. For example, I might schedule a job to run every hour, every night, or even trigger it through an API.

What I really like is the ability to create workflows within a job. I can have multiple tasks, each depending on another for example, load data → transform → train model → notify via email. And each task can run on its own cluster, which saves cost.

Databricks also gives detailed logs and alerting. If something fails, I can get an email or trigger a webhook. That makes it easy to track what’s going on without having to manually check.

So yeah, for any kind of recurring data processing, I always use Databricks Jobs to make it reliable and hands-off.

87. What is the function of the Databricks CLI?

The Databricks CLI is a command-line tool that I use when I want to interact with Databricks from my local machine or a DevOps pipeline.

For example, if I want to upload a notebook into the workspace, I don’t need to open the UI I just run a simple command. Same goes for downloading files, managing clusters, or triggering jobs. I’ve used it a lot in automation tasks.

One common use case for me is deploying notebooks and job definitions as part of a CI/CD process. I write the code locally, push it to Git, and then the CLI handles pushing it into Databricks automatically.

Another example is managing secrets. Using the CLI, I can create and update secret scopes or values without going into the workspace.

Overall, I find the CLI really useful when I want to script repetitive tasks or integrate Databricks actions into a broader development or deployment workflow.

88. What is the difference between interactive and automated clusters?

The main difference between interactive and automated clusters in Databricks comes down to how they’re used and who controls them.

Interactive clusters are the ones I usually spin up when I’m working inside a notebook. These are shared clusters where I can run ad-hoc queries, explore data, or test out some code manually. I usually keep these running for a while if I'm actively developing. Since they're shared, multiple notebooks or users can connect to the same cluster.

Automated clusters, on the other hand, are created automatically when a job runs. Let’s say I schedule a nightly ETL job Databricks will spin up a new cluster just for that job, run it, and then shut the cluster down afterward. These clusters are isolated, and they only exist for the duration of the job.

So in simple terms: interactive clusters are for development and exploration, while automated clusters are for production jobs that need to be scheduled or triggered programmatically. I usually prefer job clusters for scheduled tasks because they don’t stay on longer than needed, which helps save cost.

89. How does job recovery work in Databricks?

Job recovery in Databricks is actually pretty straightforward. If a job fails maybe because of an error in the code, cluster failure, or even a temporary network issue Databricks provides a way to automatically retry the job or the specific task that failed.

When setting up a job, I can define a retry policy. For example, I might say, "retry up to 3 times with a 5-minute delay between attempts." That way, if something goes wrong due to a minor glitch, the job doesn’t fail completely it just tries again.

Also, since each task inside a job is logged separately, I can go into the job run history, see where it failed, check the error logs, and re-run just that task or the entire job if needed. This makes debugging much easier, and I don't have to start everything from scratch.

So overall, job recovery in Databricks is about automatic retries and good visibility into job runs, which is very useful in production environments.

90. How do you pass parameters to a Databricks job?

There are a couple of ways I usually pass parameters to a Databricks job.

If I'm running a notebook as a job, I can use Databricks widgets. I define widgets at the top of the notebook using code like dbutils.widgets.text("input_date", ""), and then I can pass in the actual value when I configure the job. For example, I might pass input_date = 2025-07-01 when scheduling the job.

If I'm using a Python script or a JAR, then parameters can be passed through the job configuration using the "parameters" field, and I handle them in the script using something like sys.argv or Spark's built-in argument parsing.

Also, when using the Jobs REST API or the CLI, I can pass parameters programmatically as part of the job payload. This is useful for dynamic workflows where input values change depending on the scenario.

Passing parameters is a simple but powerful way to make jobs reusable for different inputs.

91. What are the steps to configure job alerts in Databricks?

Setting up job alerts in Databricks is something I usually do to stay on top of any failures or unexpected behavior in production jobs.

Here's how I do it:

1. When I create or edit a job, there's an "Alerts" section.
2. I can choose to send alerts for different job outcomes like on success, on failure, or when it's skipped.
3. Then I add the email addresses or webhook URLs where notifications should be sent. This can go to me, my team, or even to a Slack channel via webhook.
4. If needed, I can use monitoring tools like Azure Monitor or custom integrations through REST APIs for more advanced alerting.

These alerts help me react quickly if something goes wrong, especially for critical jobs that run overnight or during non-working hours.

92. How does cluster autoscaling work in Azure Databricks?

Cluster autoscaling in Databricks is a really helpful feature that automatically adjusts the number of worker nodes in a cluster depending on the workload.

So instead of manually setting a fixed number of workers, I just define a minimum and maximum range like 2 to 8 workers. When the workload increases, like during a heavy data processing task or a wide shuffle, Databricks will automatically add more nodes to speed things up. And when the load drops, it removes the extra nodes to save cost.

I’ve used this feature mostly in production ETL jobs, where the workload can change a lot depending on the time of day or the data size. It’s great because it balances performance and cost without needing constant manual intervention. And the best part is, it's built-in just check a box and set the range while creating the cluster.

93. What are the cluster termination settings in Databricks?

Cluster termination settings are basically a way to shut down idle clusters automatically so that we don’t end up paying for compute that’s not being used.

When I create an interactive cluster, I usually set an “auto termination” time, like 30 minutes or 1 hour. That means if the cluster isn’t doing anything no active commands or jobs for that time period, it will automatically terminate. This is really useful when I forget to shut down a cluster after testing or debugging something.

It’s especially important from a cost perspective, because in Databricks, you pay for compute by the minute. So these settings make sure that development or testing clusters don’t stay on longer than needed.

94. How do you monitor resource usage in Databricks?

There are several ways I keep an eye on resource usage in Databricks.

Inside the workspace, every cluster has a “Metrics” tab where I can see how much CPU and memory each node is using, how many tasks are active, and if there’s any skew. This is helpful when I want to figure out if a job is taking too long because of limited resources.

When running a job or notebook, there’s also a “Spark UI” link. I use this a lot it gives detailed info on stages, tasks, shuffle operations, and even storage usage. It’s kind of like the backend engine view of what Spark is doing.

On top of that, Databricks supports integration with tools like Azure Monitor or Ganglia for external monitoring. So in a production setup, I can send logs and metrics to a dashboard and get alerts if something looks off.

95. What are init scripts in Databricks and how are they used?

Init scripts in Databricks are shell scripts that run automatically every time a cluster starts up. I’ve used them mostly to customize the environment before the cluster is ready to run any code.

For example, sometimes I need to install extra libraries that aren’t available in Databricks by default, or I want to set some environment variables for my job. In that case, I write an init script that installs those packages or configures the settings I need.

The scripts can be stored in DBFS or even referenced from a Git repo or Azure storage. Once I attach the script to the cluster configuration, it runs at startup on each node both the driver and the workers.

They’re especially useful in production environments where consistency and repeatability are important. Instead of manually installing something every time, the init script does it automatically whenever the cluster starts.

96. What is the maximum number of concurrent jobs you can run in a Databricks workspace?

The maximum number of concurrent jobs in a Databricks workspace really depends on the pricing tier and the limits set by the platform. On the default standard plan, Databricks allows up to 1000 concurrent job runs per workspace.

But to be honest, I rarely hit that limit in real-world projects. If I ever get close, it usually means I need to either refactor how my jobs are scheduled or request a quota increase from Azure support.

Also, there are ways to control concurrency at the job level. For example, I can limit how many concurrent runs are allowed for a specific job by setting a maximum concurrency value. That way, I can make sure not to overload the cluster or the workspace with too many jobs at once.

97. What is a Databricks cluster policy?

A cluster policy in Databricks is basically a set of rules or templates that control how clusters can be configured in a workspace. I use it to enforce standardization, especially in large teams or shared environments.

For example, if I'm a platform admin, I can create a policy that says: "Only use certain instance types," or "Enable autoscaling," or "Disable public IPs." Then I assign that policy to a team or group, and whenever they create a cluster, they can only pick options allowed by that policy.

This really helps with security, cost control, and operational consistency. It also makes the cluster setup process simpler for users because they don't have to guess the right settings everything's pre-defined.

98. How can you enforce secure cluster configurations using policies?

To enforce secure cluster configurations using policies, what I usually do is define a cluster policy that locks down sensitive or risky settings.

For example, I can disable the option to attach public IPs to clusters, which helps protect against external access. I can also enforce that all clusters must use credential passthrough, which means users access data with their own identity, improving accountability.

Another thing I do is restrict cluster creation to use only certain VM types that meet our security or compliance standards like using only VMs with encrypted disks.

I can even block the use of init scripts from unknown locations or require that only Unity Catalog is used for data access, which ensures we're following governance rules.

So overall, cluster policies give me a way to make sure that all clusters created by users follow our organization's security practices automatically without relying on them to configure it correctly every time.

99. How do you configure logging for jobs in Databricks?

When it comes to job logging in Databricks, the platform automatically captures logs for each job run, which I find super helpful. These logs include standard output, error messages, and execution details.

By default, these logs are available through the “Runs” tab under each job, where I can see the stdout and stderr for every task. But if I want to persist logs for auditing or debugging later, I usually configure the job to write logs to a location in DBFS or external storage like ADLS.

For example, in my job code, I might explicitly log important messages using print() in Python or log4j in Scala, and then write the output to a defined log path.

Also, if I want to centralize the logging or integrate it with monitoring tools like Azure Monitor or Log Analytics, I can stream logs or export them using a log aggregation tool or through Databricks REST APIs.

So in short, the default logging is helpful for quick troubleshooting, but for larger environments or production jobs, I usually set up custom logging to a storage location or external monitoring tool.

100. What are DBFS root and mounts?

DBFS stands for Databricks File System. It's basically a layer on top of cloud storage that lets me read and write files using a familiar file path structure, like /dbfs/.

The DBFS root is the base path of this file system, and it’s divided into two parts: the default root (like /dbfs/) and the “local” storage attached to each driver or executor. The root area is persistent, so I can store scripts, data files, or job outputs and access them later.

Mounts, on the other hand, are a way to link external storage like Azure Data Lake Storage or Blob Storage into the DBFS namespace. When I create a mount point using dbutils.fs.mount(), I can access cloud storage just like a local path, for example: /mnt/rawdata.

I use mounts a lot because they simplify access to external data. Instead of writing long URLs or managing credentials every time, I just mount the storage once and access it like a regular folder.

101. How is access control handled in Unity Catalog?

Unity Catalog brings fine-grained access control to Databricks at the data level, and it’s a huge improvement over older methods.

It allows me to control access using a centralized governance model. So instead of setting permissions at the table or storage level using ACLs or blob permissions, I define them within Unity Catalog using familiar SQL GRANT statements.

For example, I can say something like: “GRANT SELECT ON schema.sales TO analyst_group.” That way, only users in that group can query those tables.

One of the biggest benefits is that Unity Catalog uses identity-based access, and it's integrated with Azure Active Directory. So I can manage permissions for users and groups centrally and audit everything.

It also supports row- and column-level security, and works across multiple workspaces, which is really useful when you have different teams or environments that need secure and isolated data access.

102. What is SCIM and how is it used with Azure Databricks?

SCIM stands for System for Cross-domain Identity Management. It's a standard that helps automate the user and group provisioning process between identity providers like Azure Active Directory and applications like Databricks.

In Azure Databricks, SCIM is used to automatically sync users and groups from Azure AD into the Databricks workspace. So instead of manually adding every user or maintaining roles inside Databricks, the whole thing becomes automated.

For example, if someone joins the company and gets added to a group like "Data Engineers" in Azure AD, SCIM will automatically provision them in Databricks and assign them to the correct group inside the workspace. If they leave or change roles, SCIM updates that too.

I've found it really useful in enterprise setups where there are lots of users, and you want to be sure access is consistent, secure, and automatically managed.

103. What are service principals and how are they used in Databricks?

A service principal is basically an identity that's used by applications or automation tools not by actual people.

In Databricks, I use service principals when I want to automate something securely, like running jobs through Azure DevOps pipelines, accessing data from external systems, or using REST APIs. Instead of logging in with a personal user account (which is not secure or scalable), I authenticate the automation with a service principal.

In practice, I create a service principal in Azure AD, give it the necessary permissions, and then map it to a Databricks group or role. This lets the automation do things like trigger jobs, access Unity Catalog, or even mount storage all without involving human credentials.

It's a clean and secure way to separate human activity from automation or system-to-system communication.

104. What is the difference between workspace and account-level identities in Databricks?

This is a key distinction, especially in multi-workspace environments.

Workspace-level identities are users and groups that exist only within a specific Databricks workspace. In older setups, this was the only level of identity management. Each workspace had its own users, which meant duplication and extra management effort if you had several workspaces.

Account-level identities, on the other hand, are managed at the Databricks account level outside of any single workspace. These are part of the newer identity model that works with Unity Catalog. When I use account-level identities, I can manage users, groups, roles, and entitlements centrally and apply them consistently across multiple workspaces.

So the main difference is: workspace-level identities are limited to one workspace, while account-level identities support centralized governance and are required if I'm using Unity Catalog or managing multiple workspaces.

105. How do you integrate Azure Active Directory with Databricks?

Integrating Azure Active Directory (Azure AD) with Databricks is one of the first things I do when setting up a secure environment. This integration allows us to manage users, groups, and access through Azure AD, which makes things a lot easier for IT and ensures consistent identity management.

To set it up, I usually start by creating an enterprise application for Databricks in Azure AD. Then, I configure SCIM provisioning so that Azure AD automatically syncs users and groups into the Databricks workspace.

For authentication, Databricks uses Azure AD for single sign-on (SSO), which means users can log in to the Databricks workspace using their Azure AD credentials. This also supports multi-factor authentication and conditional access policies if those are enforced by the organization.

On top of that, once Azure AD groups are synced, I can use those groups directly in Unity Catalog to manage data permissions. So it becomes a full end-to-end identity solution from workspace access to fine-grained data control.

106. What is a catalog in Unity Catalog and how is it structured?

In Unity Catalog, a catalog is the topmost container for organizing data assets like schemas (or databases), tables, views, and functions. It acts as a logical grouping mechanism that helps in applying governance, security, and ownership.

Think of it like this: in older setups, everything was just under a database. But with Unity Catalog, I now have a three-level namespace catalog → schema → table.

So for example, I might have a catalog called finance_data, inside which I define schemas like sales, expenses, or reporting, and within those schemas, I have actual tables like monthly_sales or budget_2024.

Catalogs are especially helpful in larger organizations or multi-team environments, because I can assign different ownership and permissions at the catalog level. For example, marketing and finance can each have their own catalog with their own data and governance rules.

107. How are databases, schemas, and tables organized in Unity Catalog?

In Unity Catalog, the hierarchy is very structured and it follows this order: catalog → schema (which is like a database) → tables/views/functions.

So each data object belongs to a schema, and each schema belongs to a catalog. This means instead of accessing data with two-part names like database.table, I now use three-part names like catalog.schema.table.

Here's a practical example:

SELECT * FROM company_data.sales.transactions

In this case:

- company_data is the catalog,
- sales is the schema (which was called a database before),
- transactions is the table.

This structure is great because it gives more flexibility and control. For instance, I can give one team access only to a certain schema inside a catalog, or even restrict them to just specific tables.

It also simplifies governance across multiple Databricks workspaces, since Unity Catalog lets me apply access policies consistently across all of them, using account-level identities and Azure AD groups.

108. How do you audit access to data in Databricks?

To audit access to data in Databricks, I typically rely on Unity Catalog because it provides detailed audit logs out of the box. Every time someone accesses a table or performs a data operation, Unity Catalog logs that activity, including what was accessed, who accessed it, and when.

These audit logs are written to a secure storage location in my cloud account. In Azure, they go into a specified Azure storage container. From there, I can connect the logs to Azure Monitor, Log Analytics, or even send them to a SIEM tool for deeper analysis.

I've used this setup to track who queried sensitive data, detect unauthorized access, or just monitor general usage patterns. It's especially helpful for meeting compliance and security requirements.

Outside Unity Catalog, I can also audit workspace activity using the workspace audit logs feature, which logs notebook activity, job runs, and user login events.

So in short, between Unity Catalog's data access logs and Databricks workspace audit logs, I have full visibility into what's going on and can investigate or report on any data access behavior.

109. What is fine-grained access control in Unity Catalog?

Fine-grained access control in Unity Catalog means I can control access not just at the table level, but even more specifically down to individual rows or columns in a table.

Let's say I have a table with salary data. With fine-grained control, I can allow HR to see everything, but restrict a junior analyst to only see anonymized or partial data like just department-level averages.

I do this using SQL GRANT statements and by creating views or applying filters based on user roles or group membership. It's a much more secure and precise way to manage access, especially when working with sensitive or regulated data.

The nice thing is that it all works with Azure AD groups, so I don't have to manage permissions manually for each user just define access rules for the group, and Unity Catalog enforces them automatically.

110. How do you use row-level security in Unity Catalog?

Row-level security in Unity Catalog lets me control which rows in a table a user or group can see, based on conditions I define.

The way I usually set it up is by creating a secure view on top of the original table. In that view, I add a WHERE clause that filters data based on the user's identity or group. For example:

```
CREATE OR REPLACE VIEW finance.filtered_data AS

SELECT * FROM finance.transactions

WHERE region = current_user_region()
```

Or, more practically, I use dynamic functions like IS_MEMBER() to check if the user belongs to a specific group:

```
CREATE OR REPLACE VIEW finance.sales_view AS

SELECT * FROM finance.sales

WHERE (region = 'US' AND IS_MEMBER('us_sales_team'))

  OR (region = 'EU' AND IS_MEMBER('eu_sales_team'))
```

This way, when someone queries the view, they only see the rows they're allowed to. The real benefit is that I don't need to create multiple versions of the same table for different groups it's all handled dynamically by Unity Catalog.

It's been really useful in large organizations where multiple departments share the same data source but should only access their own slice of data.

111. How does column-level masking work in Unity Catalog?

Column-level masking in Unity Catalog allows me to hide or mask specific column values from users who shouldn't see sensitive information, like PII or financial data.

The way it works is pretty straightforward I define a dynamic view that includes logic to mask certain columns depending on the user or their group. So instead of giving users direct access to the raw table, I give them access to the masked view.

Here's a simple example I've used before:

```
CREATE OR REPLACE VIEW hr.masked_employees AS

SELECT

 name,

 CASE

  WHEN is_member('hr_team') THEN ssn

  ELSE 'XXX-XX-XXXX'

 END AS ssn,

 salary

FROM hr.employees;
```

In this case, only users in the hr_team group can see the real SSN values. Everyone else sees a masked version. I can apply this logic to emails, phone numbers, salaries, or any sensitive field.

This feature is really important when working with regulated data especially in finance or healthcare where compliance rules require tight control over what different people can see.

112. How do you share data across Databricks workspaces?

Sharing data across Databricks workspaces can be tricky in traditional setups, but Unity Catalog and Delta Sharing have made it a lot easier.

If both workspaces are under the same Databricks account and use Unity Catalog, I can simply grant permissions at the catalog, schema, or table level to users or groups in another workspace. Since Unity Catalog supports account-level identities, the access can span multiple workspaces without duplicating users or roles.

Another approach is to use Delta Sharing, which is Databricks' built-in data sharing protocol. With that, I can share live data between workspaces even across clouds or different accounts without copying it.

And finally, if Unity Catalog isn't available, I've also used DBFS or cloud storage paths like ADLS Gen2 to move data between workspaces. But that approach requires managing access policies manually, and it's not as clean or secure.

113. What is Delta Sharing and how does it work in Databricks?

Delta Sharing is an open-source protocol developed by Databricks that allows you to share live data securely and easily with other users, workspaces, or even external partners without copying the data.

The big benefit is that I can share a table, and the other side can query it directly whether they're in Databricks, another cloud platform, or even using a tool like Power BI or Python.

Here's how it works:

- As the data provider, I set up a sharing recipient, which could be another workspace or even an external client.
- I define a share, which is basically a list of tables I want to expose.
- Then I grant access to that share.

The recipient gets a URL and credentials, and they can read the data using Databricks, or connect through open-source Delta Sharing connectors.

It's secure because it supports fine-grained access control, and efficient because the data is not copied it's read in-place from the original Delta Lake files. I've used it to share data across departments and even with third parties, and it's been a game changer compared to older methods like sending CSV files or building APIs.

114. What are the components of a Lakehouse platform in Databricks?

A Lakehouse combines the best of data lakes and data warehouses, and Databricks is built exactly on that idea. The key components of the Lakehouse platform in Databricks are:

1. **Delta Lake –** This is the foundation. It brings ACID transactions, schema enforcement, time travel, and performance optimization to your data lake. Basically, it makes your data lake behave like a database.
2. **Unity Catalog –** This handles governance. It manages data access control, lineage, and auditing across workspaces, which is really useful for enterprise setups.
3. **Notebooks and Workflows –** These are where users do development, data science, and run automated jobs. Notebooks support multiple languages like Python, SQL, Scala, etc., and can be scheduled with jobs.
4. **MLflow –** This is for managing the machine learning lifecycle. I've used it for tracking experiments, packaging models, and even deploying them.
5. **Structured Streaming –** For real-time use cases, Databricks supports stream processing natively using Spark structured streaming. You can build pipelines that update tables in near real-time.
6. **Data Engineering, BI, and ML Integration –** Databricks supports full workflows, whether you're running ETL, generating reports in Power BI, or training ML models all from the same platform.

So with Databricks Lakehouse, I don't need separate tools for a data warehouse, data lake, and ML platform. Everything is unified in one place, which makes data sharing, governance, and performance tuning much easier.

115. How does Databricks handle metadata management?

Metadata management in Databricks is handled through **Unity Catalog**. It's a centralized system that stores metadata about tables, schemas, columns, ownership, permissions, and lineage across all workspaces.

What I like about Unity Catalog is that it brings a consistent and secure way to manage metadata just like a traditional catalog in a database, but extended for the lakehouse model. So instead of each workspace having its own view of the metadata, everything is shared and managed from a central account-level layer.

Unity Catalog also integrates with audit logs and access policies, so I can track who accessed what data and when, and apply fine-grained controls over data access even at the row and column level.

Before Unity Catalog, metadata was mostly tied to individual Spark catalogs or Hive metastores, which made cross-workspace governance really messy. But now, I can manage metadata and enforce policies at the catalog level across my whole organization.

116. What is Apache Hive compatibility in Databricks?

Databricks offers compatibility with Apache Hive mainly to support teams and legacy systems that rely on Hive-based queries, metadata, and formats.

From my experience, this includes support for:

- Hive metastore (if Unity Catalog is not yet enabled),
- Hive-compatible SQL syntax,
- Hive SerDes for reading and writing data in Hive-supported formats like ORC or text,
- and even using Hive UDFs (User Defined Functions) with some setup.

This compatibility is useful when migrating workloads from older Hadoop or Hive-based systems to Databricks. So I've been able to lift and shift some legacy ETL jobs into Databricks without rewriting everything from scratch.

However, for most modern use cases, I prefer working with Delta Lake and Unity Catalog because they're more robust, cloud-native, and offer better performance and governance. But it's good to know that Hive support is there when needed for integration or migration.

117. How do you connect BI tools (Power BI, Tableau) with Databricks?

Connecting BI tools like Power BI or Tableau to Databricks is actually pretty straightforward. Databricks provides built-in connectors for both.

For Power BI, I usually use the Azure Databricks connector, which is available out of the box in Power BI Desktop. I just need the Server Hostname and HTTP Path from the Databricks SQL endpoint or cluster. I also use my Azure Active Directory credentials for secure authentication.

Once connected, I can query Databricks tables directly using SQL, and create dashboards just like I would with a regular database. There's also a Power BI integration in the Databricks workspace itself that lets me generate a .pbids file to launch Power BI with the connection pre-configured.

For Tableau, it's a similar process. I use the Databricks JDBC connector or the Databricks Tableau connector. I provide the server details, personal access token, and HTTP path, and then I'm good to go.

Both tools support live queries or extracts. I usually prefer live connections when I want real-time dashboards, and extracts when performance or data volume is a concern.

118. How do you secure external access to Databricks SQL endpoints?

Securing external access to SQL endpoints in Databricks mainly involves a few layers of protection:

1. **Authentication** – First, I make sure users authenticate using Azure Active Directory. That way, access is managed centrally and tied to corporate identities.
2. **Personal Access Tokens** or **OAuth** – If I'm using tools like Tableau or Power BI, I generate personal access tokens or use OAuth so the tool can connect securely without exposing credentials in plain text.
3. **IP Access Lists** – I can restrict which IP addresses are allowed to connect to my Databricks workspace or SQL endpoints. This helps ensure only trusted networks or locations can access the endpoint.

4. **SQL Permissions –** On top of network security, I use Unity Catalog or workspace access controls to define exactly who can query which tables or views.
5. **Cluster Policies –** For SQL endpoints, policies can be applied to restrict compute settings like instance types, scaling limits, or query timeouts, ensuring the endpoint is both secure and cost-efficient.

So with these layers identity, networking, tokens, and permissions I make sure only the right users have access and that the data stays protected.

119. How does query federation work in Databricks SQL?

Query federation in Databricks SQL allows me to write a single SQL query that pulls in data from multiple external sources like SQL Server, Oracle, PostgreSQL, or even Snowflake along with data in Delta tables on the lake.

Let's say I have sales data in a Delta Lake and customer master data in an external SQL Server. Instead of moving data around or creating pipelines, I can use query federation to directly query both sources in a single SQL statement, like:

SELECT a.customer_id, b.name, a.total_purchase

FROM delta.sales a

JOIN sqlserver.customers b

ON a.customer_id = b.id

Under the hood, Databricks uses connectors that push down as much computation as possible to the external data source to optimize performance.

I've used this feature to simplify reporting pipelines, especially when some data lives in cloud databases and the rest in the lakehouse. It's great because it saves a lot of time and avoids the complexity of ETL jobs just for joining data across systems

120. What are dashboards in Databricks SQL?

Dashboards in Databricks SQL are visualizations built directly on top of SQL queries that run within the Databricks environment. They let me turn raw query results into charts, tables, and KPIs that are easy to understand and share with others.

I usually create a dashboard by first writing a SQL query in the SQL editor, then clicking "Add to Dashboard." From there, I can choose different chart types like bar charts, pie charts, line graphs, or even just a simple table. I can also customize filters, add text boxes for context, and organize the layout.

One thing I really like is that dashboards are interactive. So I can add filters for things like date ranges, regions, or product categories and the visuals update based on those inputs. I've used dashboards to give business users a quick view into sales trends, data quality checks, and pipeline monitoring.

They're easy to share too. I can generate a link or set permissions so teams across the company can view the latest insights without needing to know Spark or SQL deeply.

121. How do you schedule dashboard refreshes in Databricks?

To keep dashboards up to date, Databricks lets me schedule the underlying SQL queries to run automatically at set intervals.

Here's how I usually do it:

- I go to the dashboard or the specific query that's part of it.
- Then I set a refresh schedule it could be every hour, once a day, or at specific times using cron syntax.
- Once scheduled, the query runs on a SQL endpoint and updates the data behind the visualizations.

There's also an option to set alerting rules for example, if a query returns a certain threshold value (like low inventory or failed pipeline count), it can send an email or Slack message.

So this scheduling helps me ensure the business always sees up-to-date metrics without someone manually running the queries every time.

122. What is the difference between Databricks SQL and traditional SQL engines?

Databricks SQL is similar in syntax to traditional SQL engines like SQL Server or MySQL, but it's built on a much more scalable and distributed backend using Apache Spark.

Here are a few key differences I've noticed:

1. **Scalability –** Traditional SQL engines usually run on a single server or a small cluster, which limits how much data they can handle. Databricks SQL runs on Spark, which is distributed, so it can process terabytes or even petabytes of data.
2. **Storage –** Traditional databases store data in their own proprietary formats and structures. In Databricks SQL, I work directly on cloud object storage (like ADLS or S3) using Delta Lake tables, which means there's no need to import data into a separate system.
3. **Performance –** Thanks to features like query caching, data skipping, and Delta Lake optimizations (like ZORDER and file compaction), Databricks SQL can run complex queries surprisingly fast even on massive datasets.
4. **Elastic Compute –** Unlike fixed SQL servers, Databricks SQL uses SQL warehouses (formerly endpoints) that scale up or down based on demand. I don't have to worry about hardware capacity or provisioning.
5. **Open Format and Lakehouse Model –** Instead of locking data into a warehouse, Databricks SQL works on open formats like Parquet and Delta. This makes it more flexible for data science and machine learning workloads alongside BI.

So overall, Databricks SQL gives me the familiar feel of traditional SQL but with cloud-scale performance and much more flexibility. It's a great fit when working with big data in a modern environment.

123. How do materialized views work in Databricks?

In Databricks, materialized views let me store the results of a query physically, so that the data doesn't have to be recomputed every time I query it. It's similar to a cached table, but it stays updated when the underlying data changes depending on how I set it up.

They're useful when I have expensive or frequently used queries especially aggregations or joins that don't need real-time data. For example, I might create a materialized view to show daily sales by product and region. Instead of recalculating that every time someone runs a report, the materialized view holds the latest snapshot of the data.

Databricks keeps the materialized view in sync with the source tables based on the refresh policy I define. I can choose to refresh manually, on a schedule, or automatically.

I've used them mostly in Databricks SQL for dashboard performance or reporting use cases, where reducing query time really matters. It's not something I'd use for real-time pipelines, but for BI workloads, it can save a lot of time and cost.

124. How do you perform cost analysis in Databricks?

There are a few ways I perform cost analysis in Databricks:

1. **Billing Dashboard –** Inside the Databricks admin console (especially in the account console if I'm using multiple workspaces), there's a billing dashboard where I can see DBU consumption over time, broken down by cluster, job, user, or workspace.
2. **Cluster Tags –** I tag clusters with things like project=xyz or team=dataeng, which lets me track costs by team or project. These tags show up in Azure Cost Management too, so it's easier to reconcile Databricks spending with cloud billing.
3. **Audit Logs + Usage Logs –** I enable audit and usage logging in Databricks, which lets me export logs of who ran what job, how long it took, and what resources were used. I can analyze this in my own reporting layer.
4. **SQL Warehouse Monitoring –** For SQL workloads, I check the SQL warehouse (formerly endpoint) usage metrics like query run time, number of queries, and active users. This helps me understand which dashboards or users are consuming the most compute.

By combining these, I get a clear picture of which jobs or users are driving cost, and I can optimize by right-sizing clusters, shutting down idle ones, or using autoscaling and job clusters efficiently.

125. How do DBUs relate to pricing in Databricks?

DBU stands for Databricks Unit, and it's the main way Databricks charges for compute usage. One DBU represents an abstract unit of processing power per hour, and the total cost is calculated based on how many DBUs are consumed and for how long.

Each type of cluster or SQL warehouse has a different DBU rate. For example:

- A standard all-purpose cluster might consume 2 DBUs per worker per hour.
- A job cluster might be more cost-efficient with 1 DBU per worker.
- SQL warehouses have their own DBU pricing depending on size (like Small, Medium, etc.).

So the formula is roughly:
Cost = DBUs used × DBU rate ($) × time (hours)

I pay for both driver and worker nodes, and the more nodes I use or the longer they run, the more DBUs I consume. That's why it's important to use autoscaling, cluster termination settings, and job clusters wisely so I'm not paying for idle time.

Also, using features like Photon (Databricks' high-performance engine) can reduce runtime and, in turn, lower DBU usage for the same workload.

126. What are the Databricks pricing tiers?

Databricks offers a few pricing tiers, and each one includes different sets of features based on the use case. The main ones I've worked with are:

1. **Standard Tier –** This is more for basic use cases. It includes core features like notebooks, jobs, basic cluster management, and Delta Lake. It's fine for small teams or individual projects where advanced security or collaboration isn't required.
2. **Premium Tier –** This is the one I see most often in enterprise setups. It adds important things like role-based access control (RBAC), credential passthrough for accessing Azure Data Lake with user identity, audit logs, and job access control. It's better for teams that need to manage users and data securely.
3. **Enterprise Tier –** This is the highest tier and includes everything in Premium, plus features like Unity Catalog, enhanced security (like customer-managed keys), network isolation, and better governance. It's ideal for regulated industries or companies with strict compliance needs.

Each tier adds more capabilities, but also increases the DBU cost. So when I help teams choose, I usually weigh what security, compliance, and governance features are actually required for the workload.

127. What is the cost of a job cluster versus an all-purpose cluster?

The cost mainly comes down to how many DBUs each cluster type consumes per hour.

- Job clusters are designed to run automated workloads like scheduled ETL jobs or notebooks triggered by workflows. They're more cost-efficient because they spin up just for the job and terminate right after. They generally use fewer DBUs per hour compared to all-purpose clusters. For example, they might cost around 1 DBU per node per hour.
- All-purpose clusters, on the other hand, are used for interactive analysis and collaboration in notebooks. Since they stay running for longer and support multiple users and sessions, they're priced higher maybe around 2 DBUs per node per hour.

So for production pipelines, I always prefer job clusters. They start fast, cost less, and shut down automatically. But for ad hoc analysis or development work, all-purpose clusters are more convenient, even though they cost more.

128. What are the primary billing components in Azure Databricks?

There are a few key components that contribute to the billing in Databricks:

1. **DBU Consumption –** This is the biggest part. It's the cost of compute usage, based on how many DBUs are used by clusters or SQL warehouses. It varies by cluster type and pricing tier.
2. **Compute Resources (VMs) –** Azure also charges separately for the underlying virtual machines (like Standard_D4s_v3). So I'm paying both Databricks (for DBUs) and Azure (for VM time).
3. **Storage –** While most storage is in Azure Data Lake or Blob, I might also get charged for storage used in DBFS (Databricks File System) or delta tables if they're in premium-tier storage accounts.
4. **Networking –** There could be some cost associated with VNET peering, data egress, or NAT gateway usage especially in enterprise setups where network architecture is locked down.
5. **Features Used –** Some features like Unity Catalog, SQL Pro vs SQL Classic, Photon acceleration, or Delta Live Tables may carry different DBU pricing or requirements for premium tiers.

To keep billing under control, I usually recommend setting up cluster policies, idle timeouts, autoscaling, and using the cost and usage reports available in both Azure and the Databricks workspace.

129. What are the ways to reduce cost in Azure Databricks?

There are actually several strategies I use to reduce cost in Azure Databricks, especially when dealing with production workloads or larger teams:

1. **Use job clusters instead of all-purpose clusters –** Job clusters are cheaper because they spin up just for the job and terminate afterward. All-purpose clusters stay up longer and cost more per DBU.
2. **Enable autoscaling –** I always turn on autoscaling so that clusters grow only when needed and shrink during idle time. That helps avoid overprovisioning.

3. **Set idle termination timeouts –** I configure clusters to shut down automatically after a period of inactivity like 10 or 15 minutes so I'm not charged when no one's using them.
4. **Use cluster policies –** Policies help enforce things like max workers, auto-termination, or disallowing all-purpose clusters for certain users, which keeps costs under control.
5. **Photon engine –** When using Databricks SQL or Delta Lake, enabling Photon can dramatically reduce query times, which means fewer DBUs consumed overall.
6. **Use Delta tables –** Delta Lake improves performance with caching, indexing, and data skipping. Faster jobs mean less compute time and lower cost.
7. **Tag clusters for visibility –** By tagging clusters with project or team names, I can track usage and identify where the most cost is coming from.
8. **Schedule jobs during off-peak hours –** In some organizations, spot pricing or less load during off-hours helps reduce cost too.

So in short, it's about choosing the right cluster type, using automation to scale and shut things down, and continuously monitoring usage patterns.

130. What is cluster pooling and how does it save cost?

Cluster pooling is a feature in Databricks that helps speed up job start times and save cost at the same time. Normally, when I run a job, it takes some time to spin up new VMs, start the cluster, and get it ready that's often a few minutes, and I'm paying for that time.

With a cluster pool, Databricks keeps a few ready-to-use instances (like pre-warmed VMs) available. So when a job or notebook runs, it can borrow a machine from the pool instantly instead of waiting for a new one to start from scratch.

Here's how it saves cost:

- The startup time is much lower, so I'm not wasting DBU or VM charges during job startup.
- Multiple jobs can reuse nodes from the same pool, which means fewer idle VMs sitting around.
- Since the pool automatically scales down when not in use, I avoid unnecessary charges.

I've used it a lot in teams where short jobs run frequently like hourly ETLs. It keeps things fast and reduces the "waiting time" cost that adds up across the day.

131. What is spot instance usage in Databricks?

Spot instances (also called low-priority VMs in Azure) are virtual machines that are much cheaper than regular ones sometimes up to 80% cheaper. The catch is that Azure can take them back at any time when resources are needed elsewhere.

In Databricks, I can choose to run clusters using spot instances for the worker nodes (not the driver), which helps save a lot of cost for non-critical or retry-safe workloads.

I usually use spot instances for:

- Batch ETL jobs that can restart if interrupted
- Data exploration or training ML models where it's okay to retry
- Scenarios where cost matters more than 100% uptime

Databricks even lets me set a fallback to on-demand VMs if spot capacity isn't available, so the job still runs but at a higher cost. I always recommend using spot instances for dev/test or fault-tolerant pipelines where a little risk is acceptable in exchange for lower cost.

132. How do you monitor cluster cost over time?

To monitor cluster cost over time in Azure Databricks, I usually combine several tools and approaches:

1. **Cluster Tags –** First, I make sure every cluster is tagged properly with things like project, team, or environment. These tags flow through to Azure Cost Management, so I can break down the cost by team or workload.
2. **Azure Cost Management –** This is my go-to tool to view cost trends over time. Since Databricks charges are broken down by DBUs and Azure VM usage, I can track how much is being spent daily or monthly per resource group or tag.
3. **Databricks Usage Logs –** Inside the Databricks workspace, I enable usage logging. These logs tell me which user ran what, on which cluster, for how long. I usually export these logs to a storage account or to a monitoring tool for analysis.
4. **Audit + Usage Log Dashboards –** In some cases, I build a custom dashboard using Power BI or Azure Monitor to visualize costs over time broken down by cluster type, user, or time of day. This helps identify things like overused all-purpose clusters or clusters left running idle.

So in short, I rely on tagging, usage logs, and Azure's native tools to get a full picture of where costs are coming from, and how to optimize them over time.

133. What are the audit logging options available in Azure Databricks?

Audit logging is critical, especially in enterprise environments, and Databricks provides a couple of ways to do this:

1. **Workspace-Level Audit Logs –** In the Premium or Enterprise tier, I can enable audit logs that track user activity like login attempts, notebook edits, job runs, cluster changes, and permission changes. These logs can be sent to an Azure storage account for central collection.
2. **Unity Catalog Audit Logs –** If I'm using Unity Catalog, it adds even more detailed logs related to data access like which user queried what table, what operation they performed, and whether access was allowed or denied.
3. **Azure Diagnostic Settings –** I can also route logs directly from Databricks to **Azure Monitor**, **Log Analytics**, or **Event Hubs** using diagnostic settings. This helps integrate logging into broader enterprise monitoring pipelines.
4. **SQL Audit Logs –** For SQL workloads, I get logs that track who ran which SQL queries and their performance metrics, which is great for both security and performance monitoring.

So, overall, there are quite a few audit logging options depending on what I want to track user behavior, data access, cluster activity, or SQL queries.

134. How do you integrate Databricks with Azure Monitor?

Integrating Databricks with Azure Monitor allows me to send logs, metrics, and alerts from Databricks into the wider Azure ecosystem. Here's how I typically do it:

1. **Enable Diagnostic Logs –** First, I enable diagnostic logging on the Databricks workspace using Azure Portal or CLI. I can choose to send the logs to:

   Log Analytics Workspace

   Azure Storage Account

   Event Hubs
2. **Log Analytics for Monitoring –** If I choose Log Analytics, I can then query logs using KQL (Kusto Query Language), create alerts, and build dashboards in Azure Monitor or Azure Dashboard.
3. **Metrics Integration –** For clusters, I monitor metrics like CPU, memory usage, and DBU consumption using **Ganglia** (built into Databricks) or export those metrics to Azure Monitor if deeper integration is needed.
4. **Alerts and Automation –** Once the logs are in Azure Monitor, I set up alerts for example, alert me if a cluster runs more than X hours or if a job fails too often. I can also use Logic Apps to automate responses.

This kind of integration is really valuable in production environments where I need centralized observability and alerts across multiple services, not just Databricks alone.

135. What metrics are available through the Ganglia UI in Databricks?

In Databricks, Ganglia is a built-in monitoring tool that gives me detailed metrics at the cluster level. It's super useful when I want to understand how my Spark jobs are using the underlying compute resources.

Some of the key metrics I usually look at in the Ganglia UI include:

- **CPU usage**: This shows how much processing power is being used per node. It helps me identify if nodes are under- or over-utilized.
- **Memory usage**: I monitor how much memory is used and how much is free. If memory usage is high, that might lead to spilling or out-of-memory errors.
- **Disk I/O**: This tracks read/write activity on local disks. It's useful to know if my job is doing a lot of disk operations (like shuffles or caching).
- **Network I/O**: I use this to track the amount of data being transferred between nodes, especially during operations like joins and shuffles.
- **JVM Garbage Collection (GC) activity**: If there's a lot of GC happening, that usually means the job is memory-intensive, and I may need to optimize the code or increase executor memory.
- **Spark Executor metrics**: I can also dig into the performance of individual Spark executors how long they run, how much data they process, etc.

I usually use Ganglia during development or troubleshooting to get a quick feel for how well the cluster is handling the workload.

136. What is Databricks REST API used for?

The Databricks REST API is basically how I can programmatically interact with the Databricks workspace. It gives me the ability to automate tasks that I would otherwise do manually through the UI.

Some common use cases I've worked with include:

- **Launching jobs**: I can use the API to trigger a job or notebook automatically from an external system like Azure Data Factory or a CI/CD pipeline.
- **Managing clusters**: I can start, stop, or resize clusters using API calls.
- **Importing/exporting notebooks**: This is helpful for version control or automated deployments across environments.
- **Managing DBFS (Databricks File System)**: I can upload files, create directories, or list file contents through the API.
- **User and permissions management**: The API allows me to assign users, set permissions, or configure access controls although this part is mostly done through SCIM or Unity Catalog now.

In short, it's super powerful when I want to integrate Databricks into an automated data pipeline or workflow.

137. How do you authenticate against the Databricks REST API?

To authenticate against the Databricks REST API, I usually use a personal access token (PAT). It's the most straightforward and commonly used method, especially in automation scenarios.

Here's how it works in practice:

1. I generate a token from my Databricks user settings page.
2. Then I include the token in the header of my API call, like this:

Authorization: Bearer <your-token-here>

For example, using curl:

```
curl -X GET https://<databricks-instance>/api/2.0/clusters/list \

-H "Authorization: Bearer dapi1234567890abcdef"
```

In enterprise environments, there are a couple of other ways I've used too:

- **Azure AD authentication** using service principals and OAuth2, especially when integrating with tools like Azure DevOps or Microsoft Entra.
- **SCIM token** for user provisioning when working with identity providers.

But for most REST API calls, PAT tokens are the easiest and quickest way to get started.

138. What is a personal access token in Databricks and how is it used?

A personal access token, or PAT, in Databricks is basically a secure way to authenticate yourself when you're using the REST API or other tools outside the Databricks UI.

Instead of using a username and password, I generate a token from my Databricks account settings. Once created, the token acts like a password that I can pass in the header of my API requests to prove my identity.

Here's how I typically use it:

- I generate the token from the User Settings page.
- Then in my API request, I include it in the Authorization header like this:

Authorization: Bearer <your-token>

For example, if I want to list all clusters using curl, I'd write:

```
curl -X GET https://<databricks-instance>/api/2.0/clusters/list \

-H "Authorization: Bearer dapi1234567890abcdef"
```

It's really helpful when I want to automate things like triggering jobs, uploading notebooks, or managing clusters. One thing to note is that tokens do expire (or can be revoked), so it's good practice to rotate them regularly and not hardcode them in scripts usually I store them in Azure Key Vault or environment variables.

139. How do you automate job deployment using REST API?

Automating job deployment in Databricks using the REST API is something I've done when integrating with CI/CD tools like Azure DevOps or GitHub Actions.

Here's the basic flow I follow:

1. **Prepare a job definition**: I create a JSON payload that defines the job. This includes the cluster spec, the task (notebook or script), libraries, schedule, and any parameters. Example:

```
{
 "name": "daily_etl_job",
 "existing_cluster_id": "abc-123",
 "notebook_task": {
  "notebook_path": "/Repos/team/data_pipeline"
 }
}
```

2. **Make the API call**: I use the 2.1/jobs/create endpoint and pass my JSON payload using curl or from a CI/CD pipeline.

```
curl -X POST https://<databricks-instance>/api/2.1/jobs/create \
-H "Authorization: Bearer <token>" \
-H "Content-Type: application/json" \
-d @job_config.json
```

3. **Versioning**: To manage updates, I either use jobs/reset if the job already exists, or keep a mapping of job IDs for different environments (like dev, test, prod).

This setup helps me maintain repeatable deployments of jobs without needing to go into the UI, and it makes my workflow more reliable and scalable.

140. How do you automate notebook execution using REST API?

To automate the execution of a notebook using the Databricks REST API, I use the jobs/run-now endpoint. This is really useful when I want to trigger notebooks on demand for example, from a webhook or as part of a larger pipeline.

Here's how I usually do it:

1. **Create the job first**: Either through the UI or via API, I register the notebook as a job. It can be a one-time setup.
2. **Run the job via API**: I call the 2.1/jobs/run-now endpoint, passing the job ID and any parameters.

Example:

```
curl -X POST https://<databricks-instance>/api/2.1/jobs/run-now \
-H "Authorization: Bearer <token>" \
-H "Content-Type: application/json" \
-d '{
 "job_id": 123,
 "notebook_params": {
  "table": "sales_data",
  "date": "2025-07-20"
 }
}'
```

3. **Monitor the run**: The response includes a run_id, which I can use to check the status using the jobs/runs/get endpoint.

This approach has worked well for me when integrating with tools like Power Automate, Azure Data Factory, or when setting up manual triggers with parameters.

141. How do you export and import Databricks notebooks programmatically?

Exporting and importing notebooks programmatically is something I've done a lot when setting up CI/CD pipelines or moving code between environments.

To export a notebook, I use the Databricks REST API. The endpoint I call is 2.0/workspace/export. It lets me download a notebook in formats like SOURCE, HTML, or JUPYTER.

Here's an example using curl to export a notebook in source format (Python, Scala, etc.):

```
curl -X GET https://<databricks-instance>/api/2.0/workspace/export \
 -H "Authorization: Bearer <token>" \
 --data-urlencode 'path=/Users/abc/my_notebook' \
 --data-urlencode 'format=SOURCE' \
 -o my_notebook.py
```

To import a notebook, I use the 2.0/workspace/import endpoint. I pass the path where I want the notebook to land, along with the file contents encoded in base64.

Here's an example:

```
curl -X POST https://<databricks-instance>/api/2.0/workspace/import \
 -H "Authorization: Bearer <token>" \
 -H "Content-Type: application/json" \
 -d '{
  "path": "/Users/abc/my_new_notebook",
  "format": "SOURCE",
  "language": "PYTHON",
  "content": "<base64-encoded-file-content>",
  "overwrite": true
 }'
```

So, I typically use these API calls in automation scripts, especially when syncing notebooks from Git repos or promoting notebooks from dev to test/prod environments.

142. What is MLflow and how does it integrate with Databricks?

MLflow is an open-source platform for managing the machine learning lifecycle, and the cool part is that it's built into Databricks. So when I'm training or tuning models, I can use MLflow to track everything from parameters and metrics to models and artifacts.

Here's how I usually use MLflow inside Databricks:

- **Tracking experiments**: With just a few lines of code, I can log parameters, accuracy, loss, and other metrics using mlflow.log_param() and mlflow.log_metric(). This is super helpful when I'm comparing models or runs.
- **Model versioning**: Every time I train a new model, MLflow can save it as a versioned artifact. Later I can register it in the MLflow Model Registry directly from the notebook.
- **Reproducibility**: Since MLflow logs code, data, and configs, I can always reproduce a model training run exactly as it was, even months later.
- **Deployment**: After training, I can use MLflow to deploy the model right from Databricks to a REST endpoint or export it to a production environment.

Overall, MLflow saves a ton of manual effort and makes collaboration across data science and engineering teams much easier.

143. How do you perform hyperparameter tuning in Databricks?

In Databricks, I usually perform hyperparameter tuning using either native Python techniques (like GridSearchCV or RandomizedSearchCV from sklearn) or more advanced methods using Hyperopt with MLflow integration.

The easiest way I've found to do it at scale is by using MLflow + Hyperopt together. Here's how the flow usually works:

1. **Define the search space** for parameters I want to tune. For example, learning rate, depth, etc.

```
from hyperopt import hp


search_space = {
  'max_depth': hp.choice('max_depth', [3, 5, 7]),
  'learning_rate': hp.uniform('learning_rate', 0.01, 0.3)
}
```

2. **Define an objective function** that trains a model and returns a loss. I also log metrics using mlflow.log_metric() inside this function.
3. **Run tuning with Hyperopt's fmin()**, and optionally use SparkTrials to parallelize across the Databricks cluster:

```
from hyperopt import fmin, tpe, SparkTrials
best_params = fmin(
  fn=objective_function,
  space=search_space,
    algo=tpe.suggest,
    max_evals=20,
    trials=SparkTrials()
)
```

4. **Track everything in MLflow**, which automatically logs each run and its outcome. This helps me compare and pick the best model visually.

This approach is great because it scales well on large datasets, and I don't have to do trial-and-error manually. It's especially powerful when combined with MLflow, since I get full experiment tracking along with tuning.

144. How does distributed training work in Databricks?

Distributed training in Databricks means training machine learning models using the power of multiple machines (or nodes) at once. Instead of training on a single machine, the data and computation are split across a cluster, which really speeds things up especially for large datasets or complex models.

In Databricks, distributed training can be done in a few ways depending on the framework:

1. **Spark MLlib –** Since Spark is naturally distributed, any ML model I train using MLlib (like logistic regression or decision trees) automatically gets trained in a distributed way without me having to manage the low-level details.
2. **Horovod –** For deep learning with frameworks like TensorFlow or PyTorch, I use HorovodRunner in Databricks. It's an abstraction that lets me train deep learning models across multiple GPUs or machines using just a few code changes. It supports parameter synchronization and scaling automatically.
3. **Petastorm and Spark Dataset Integration –** When I need to preprocess huge datasets before training, I can use Petastorm to stream Spark DataFrames directly into PyTorch or TensorFlow while keeping the training distributed.

So basically, Databricks makes distributed training very accessible. I don't need to set up MPI clusters or manage GPU infrastructure manually it handles all that behind the scenes.

145. What are feature stores and how does Databricks support them?

A feature store is a centralized place to store, manage, and reuse features (basically the inputs for ML models). It helps keep features consistent between training and inference, and also lets different teams collaborate better by reusing existing features.

In Databricks, there's a built-in Feature Store that I've found super useful. Here's how I usually work with it:

- **Create and log features**: I define features in a notebook or pipeline, and then log them to the feature store using simple APIs. These features are stored along with metadata like descriptions, owners, and timestamps.
- **Use in training**: When I'm training a model, I can load features directly from the store. Databricks ensures that the same transformation logic is applied every time, so I don't have data leakage or inconsistencies.

- **Use in batch or real-time inference**: The feature store also supports fetching features in real-time or through batch jobs, so I don't have to re-engineer the feature logic during deployment.

It solves one of the most common issues in ML projects keeping training and production data consistent. Plus, it encourages reuse, so I don't have to keep building the same features over and over again.

146. What are the data validation options in Databricks workflows?

When I'm working with data pipelines or ML workflows in Databricks, validating data at different stages is really important. Databricks gives me several ways to do this:

1. **Expectations in Delta Live Tables (DLT) –** This is my go-to method for validating streaming or batch data. I can define expectations like:

EXPECT column_name IS NOT NULL

EXPECT column_age > 0

These rules are enforced automatically as the data flows through the pipeline. I can also choose what to do with records that fail either drop them, send them to quarantine, or just log them.

2. **Using Great Expectations –** I sometimes integrate Great Expectations into my notebooks or workflows. It's a Python-based library that lets me write detailed checks and test data quality. I usually use it in batch pipelines for more customized validations.
3. **Custom validation code –** For ad-hoc checks, I write simple PySpark or Pandas logic to test nulls, data ranges, duplicates, and schema mismatches. This works well when I want more flexibility or logic that isn't just column-level.
4. **Alerts and logging –** I also set up alerts in jobs if the data fails certain validation thresholds, like too many missing values or outliers. This helps in proactively catching issues before they move downstream.

Overall, Databricks gives me a lot of flexibility depending on how critical the validation is and how automated I want it to be.

147. How do you use Great Expectations with Databricks?

I've used Great Expectations in Databricks to do automated data quality checks, and it fits in really well with the data pipeline workflows.

To use it, I usually install the Great Expectations package directly in the notebook using %pip install great_expectations. After that, I initialize a new Great Expectations project in DBFS or local mode using DataContext.create().

Once set up, the main steps are:

1. **Create Expectations –** I profile the data or define my own expectations like column value ranges, null checks, uniqueness, etc.
   For example:

```
suite = context.create_expectation_suite("my_suite", overwrite=True)

batch = context.get_batch(batch_request)

batch.expect_column_values_to_not_be_null("user_id")
```

2. **Validate data –** Before loading or transforming data, I run these expectations to catch issues early. I can also configure alerts or logging to flag failed validations.

3. **Integrate in workflows –** I usually wrap these validations inside Databricks jobs or Delta Live Tables so the pipeline stops or reroutes bad data automatically.

One cool thing is that Great Expectations also generates clean HTML reports for each validation, which is great for audits or sharing with the team.

148. What is the difference between interactive notebooks and workflows in Databricks?

This is a common question and one I've experienced first-hand. So, interactive notebooks and workflows serve different purposes in Databricks:

- Interactive notebooks are what I use during development or exploration. They let me write and test code step-by-step, visualize results, and tweak logic on the fly. These are great when I'm experimenting with data, building a model, or debugging a transformation.
- Workflows, on the other hand, are for automation and scheduling. Once my logic in the notebook is solid, I schedule it as a job (workflow). Workflows can include one or multiple tasks (like notebooks, Python scripts, or SQL queries), and they can be triggered on a schedule, based on events, or manually.

So basically, I use interactive notebooks for building and testing, and then move to workflows when I'm ready to run that logic repeatedly or in production.

149. How do you test notebooks before production deployment?

Before I deploy a notebook to production in Databricks, I make sure to test it thoroughly. Here's how I usually approach it:

1. **Use smaller test datasets –** Instead of running on full production data, I run the notebook on a smaller or sampled version of the data. This helps me test faster and safely.
2. **Validate logic with asserts –** I add assertions or validations inside the notebook to check assumptions, like:

assert df.filter("status IS NULL").count() == 0

3. **Dry-run with dummy configs –** If the notebook depends on parameters or paths, I test it with test configs using widgets or dbutils.widgets.get() so I can simulate different runs.
4. **Run in a separate environment –** I typically test in a staging or dev workspace, not directly in production. If I'm using Repos, I create a feature branch and test everything there before merging.
5. **Log results and edge cases –** I log metrics or outputs using mlflow.log_metric() or custom logging, especially when I want to track what happens in edge cases.
6. **Convert it to a job and test the job run –** Even after testing manually, I like to run it as a scheduled or triggered job to ensure that cluster configuration, dependencies, and parameters are working as expected.

This kind of end-to-end testing helps me catch issues before they go live and ensures that production runs are reliable and predictable.

150. How do you orchestrate pipelines in Databricks?

To orchestrate pipelines in Databricks, I typically use Databricks Workflows. It's a built-in feature that lets me define a sequence of tasks, like notebooks, Python scripts, JARs, or SQL commands, and run them in a defined order.

Here's how I usually do it:

- I start by creating a new job in the Jobs tab.
- Then I add multiple tasks, where each task can be a notebook or a script. I define dependencies between them, like "Task B should only run after Task A completes."
- I can pass parameters between tasks, schedule jobs, and even include retries if a task fails.
- I also use task conditions, so some tasks run only if others succeed or fail, which is useful for error handling and branching logic.

For more advanced orchestration or when I want to trigger jobs across tools, I sometimes use Azure Data Factory or Apache Airflow, depending on the project. But for most Databricks-native workflows, the built-in job orchestration works well.

151. What is the difference between Databricks and Data Factory?

Databricks and Azure Data Factory are both used in data engineering, but they have very different purposes.

- Databricks is a big data analytics and machine learning platform. I use it mainly for heavy data processing, building machine learning models, running Spark jobs, and doing advanced transformations. It's built for data engineers, data scientists, and analysts who need powerful compute and code flexibility.
- Azure Data Factory (ADF) is more of an orchestration tool. I use it to move and orchestrate data between different sources. It's great for ETL/ELT pipelines that don't need complex transformations. It's a low-code tool with drag-and-drop features, ideal for scheduling, copying, and moving data.

So, I usually combine them like this: ADF handles the movement and scheduling, and Databricks handles the heavy processing or transformation parts.

152. How do you integrate Databricks into Azure Data Factory pipelines?

Integrating Databricks into ADF pipelines is pretty straightforward. Here's how I usually do it:

1. In Azure Data Factory, I create a Databricks linked service by providing the workspace URL and either a personal access token or using Azure Managed Identity.
2. Then, in my pipeline, I add a Databricks Notebook activity or a Python script activity, depending on what I want to run.
3. I select the linked Databricks workspace and choose the cluster or create a new job cluster if needed.
4. I can pass parameters from ADF to the Databricks notebook, which is useful for making the notebook dynamic for example, processing data for a specific date.
5. Finally, I monitor the pipeline through ADF's monitoring tab, where I can see the execution status and logs.

This integration lets me schedule complex Databricks logic from ADF while managing the end-to-end pipeline centrally.

153. How does Azure Purview integrate with Databricks?

Azure Purview (now also called Microsoft Purview) integrates with Databricks to help with data discovery, cataloging, and governance across the entire data estate.

In my experience, the integration happens mainly at two levels:

1. Metadata Scanning: Purview can scan Databricks Unity Catalog. This means that all the catalogs, schemas, tables, views, and their metadata like column names, data types, and descriptions are pulled into Purview. That helps data stewards and analysts find and understand the data stored in Databricks.
2. Lineage and Classification: If I'm using Unity Catalog, it becomes even better because Purview can detect and trace data lineage so I can see where data came from, how it was transformed, and where it's going. It also helps in tagging sensitive data using built-in classifiers, like detecting emails, credit card numbers, or other PII.

To connect it, I usually create a new Databricks (Unity Catalog) source in Purview, authenticate it using a service principal or managed identity, and schedule regular scans.

154. How do you publish lineage information from Databricks?

When I want to publish data lineage from Databricks, the easiest and most reliable way is to use Unity Catalog. If I use Unity Catalog-managed tables and run queries using Databricks SQL or notebooks, the lineage is automatically captured.

For example, if I create a table from another using SQL or Spark commands, Unity Catalog logs:

- The source table
- The transformation logic (like a join or filter)
- The target table

This lineage can then be viewed directly in the Databricks UI under the "Lineage" tab for each table. But more importantly, Unity Catalog also exposes that lineage to external tools, including Microsoft Purview, which can then centralize it.

If I'm using tools like MLflow or Delta Live Tables, those also contribute to lineage metadata. And for custom or advanced use cases, Databricks provides REST APIs to programmatically capture and publish lineage events.

155. How does workspace migration work in Databricks?

Workspace migration usually comes up when moving from one environment to another like from dev to prod, or from one region or subscription to another.

Here's how I generally handle it:

1. **Export Notebooks and Jobs**: I export notebooks using the **Databricks CLI**, or I use the databricks workspace export_dir command to download entire directories. Jobs can also be exported as JSON using the CLI or REST API.
2. **Migrate Clusters and Libraries**: Cluster configurations (like node types, autoscaling, init scripts) can be captured and recreated in the target workspace. I sometimes automate this with Terraform if it's part of infrastructure as code.
3. **Copy Data and DBFS**: If I'm using **DBFS**, I copy data using the Databricks CLI (dbfs cp) or mount cloud storage (like ADLS Gen2) to avoid copying altogether. For Delta tables stored in external storage, I usually just reconfigure the mounts.
4. **Recreate Permissions**: ACLs on notebooks, clusters, jobs, and data need to be re-applied. If I'm using Unity Catalog, permissions can be managed at the account level and synced across workspaces using SCIM and Azure AD.
5. **Validate**: Once everything's moved, I test jobs, cluster performance, and security policies in the new workspace to make sure everything behaves as expected.

For larger or enterprise-scale migrations, Databricks also offers a **migration toolset** or professional services to help automate and validate this process.

156. What are the tools used for workspace backup and recovery?

So, Databricks doesn't have a full "one-click" backup and restore feature like some traditional platforms, but there are definitely ways I manage backup and recovery using available tools.

Here's what I use:

1. **Databricks CLI or REST API** – These are great for automating the export of notebooks, job definitions, and even DBFS files. I usually schedule regular exports of notebooks and configurations so I have versioned backups stored in something like Azure Blob Storage or Git.
2. **Repos (Git Integration)** – This is super helpful. I connect notebooks to Git repos like GitHub or Azure DevOps. That way, version control and backup happen naturally as part of my development workflow.
3. **DBFS and Delta Lake** – For data, if I'm using **Delta Lake on external storage** (like ADLS Gen2), the data is already versioned thanks to Delta's time travel features. So I can roll back to previous versions easily using Delta's VERSION AS OF or TIMESTAMP AS OF.
4. **Terraform or ARM Templates** – I also use Terraform scripts to manage infrastructure (like cluster configs, jobs, permissions). This helps me re-create environments if something goes wrong.

So even though there's no built-in "snapshot" for the whole workspace, I combine these tools to build a reliable backup and recovery setup.

157. What is a metastore in Databricks?

The metastore in Databricks is like the database for your metadata. It stores information about all the structured data like what tables exist, what columns they have, their data types, where the data is stored, and so on.

There are actually two types of metastores I've worked with:

1. **Hive Metastore (legacy)** – This is the default metastore that used to come with each workspace. It stores metadata locally per workspace, but it can also be externalized to a centralized Hive metastore if needed.
2. **Unity Catalog Metastore** – This is the newer, more powerful option. It's centralized and can span multiple workspaces. I prefer this because it supports features like fine-grained access control, lineage, and data governance.

The metastore is essential because every time I run a query or load a table, Spark reads the metadata from the metastore first before actually accessing the data.

158. How do you use external metastores (like Hive) in Databricks?

Using an external Hive metastore in Databricks is something I've done when companies want to centralize metadata across multiple Databricks workspaces or even share it with other tools.

Here's how I do it:

1. First, I set up a Hive metastore backed by a relational database like Azure SQL DB or MySQL.
2. Then, in Databricks, I configure the cluster to point to this external Hive metastore by setting the right Spark configuration properties:
    - spark.hadoop.javax.jdo.option.ConnectionURL
    - spark.hadoop.javax.jdo.option.ConnectionDriverName
    - spark.hadoop.javax.jdo.option.ConnectionUserName
    - and so on.

These properties tell Databricks to talk to that external Hive metastore instead of the built-in one.

I also make sure the metastore database is accessible over the network from the cluster and has the right firewall or private endpoint settings.

However, these days, unless there's a legacy system involved, I recommend using Unity Catalog instead, because it's more secure, easier to manage, and better integrated with things like Purview, RBAC, and data lineage.

159. What are Unity Catalog privileges and how do you assign them?

Unity Catalog privileges are basically permissions that control what users or groups can do with data objects like catalogs, schemas, tables, views, and functions.

There are different types of privileges depending on the object. For example:

- On a catalog, you can grant USE CATALOG, CREATE SCHEMA
- On a schema, you might grant USE SCHEMA, CREATE TABLE
- On a table or view, you can give SELECT, INSERT, UPDATE, DELETE

To assign these privileges, I typically use SQL GRANT statements. For example:

GRANT SELECT ON TABLE sales.transactions TO `data_analyst_group`;

In Databricks, the Unity Catalog supports assigning privileges to:

- Individual users
- Groups synced from Azure Active Directory
- Service principals (for automation or APIs

Also, the permissions are hierarchical. So, if someone doesn't have access to the catalog or schema, they won't be able to access the table even if they have SELECT on the table itself.

For ease of management, I usually create groups (like data_engineers, analysts, etc.) in Azure AD and then manage privileges at the group level. It's cleaner than assigning to individual users.

160. What are grants and roles in Databricks security model?

In Databricks, "grants" are the actual assignments of privileges to users, groups, or service principals. So when I use a GRANT command, I'm explicitly allowing someone to perform an action on a data object.

For example:

GRANT SELECT ON TABLE sales.customer_data TO `marketing_team`;

This is a grant it's saying: "Give the marketing team access to read this table."

Now, roles are a bit different in Databricks. Unlike some traditional databases, Unity Catalog doesn't use role-based access in the same way (like DBA, reader, etc.), but there are account-level roles such as:

- Account Admin – full control over the entire Databricks account
- Workspace Admin – can manage clusters, users, and settings in a workspace
- Data Steward / Owner – responsible for data access and quality

In the Unity Catalog, there is also an object ownership concept. If I create a table, I become the owner by default, and I can then grant or revoke access to others. Ownership itself can be transferred.

So, in practice, I use grants to assign permissions and rely on group memberships and ownership to control access in a clean and scalable way.

161. How do you revoke access or audit user actions in Unity Catalog?

To revoke access in Unity Catalog, I use the REVOKE command. It's pretty straightforward.

For example:

REVOKE SELECT ON TABLE sales.customer_data FROM `contractor_team`;

This will remove read access from that group. If someone tries to run a query after that, they'll get a permission denied error.

Now, when it comes to auditing, Unity Catalog supports audit logs, and these can be forwarded to tools like Azure Monitor, Log Analytics, or even SIEM systems.

With audit logging enabled, I can track:

- Who accessed what data
- What queries they ran
- When and from where they did it

These logs help with compliance (like GDPR or HIPAA), and they also make it easier to investigate issues or unauthorized access. Databricks provides these logs at the account level, and they can be sent to Azure storage accounts or Event Hubs.

I usually recommend enabling audit logging early in a project, especially for production environments. It's something security teams expect during audits.