DELOITTE DATA ENGINEER INTERVIEW EXPERIENCE (0-3 YOE)

1. Write a query to retrieve the top 3 highest salaries from an employee table.

```
SELECT DISTINCT salary
FROM        employee
ORDER BY      salary DESC limit 3 ;

--Alternatively, IF there are duplicate salaries AND we
need an accurate TOP 3:

SELECT salary
FROM   (
            SELECT   salary,
                        Dense_rank() OVER (ORDER BY salary DESC)
AS rnk
            FROM     employee ) ranked_salaries
WHERE  rnk<=3;
```

2. Explain the difference between a clustered and a non-clustered index.

- Clustered Index: Determines the physical order of data in a table. A table can have only one clustered index.
- Non-Clustered Index: Creates a separate structure to store index data. A table can have multiple non-clustered indexes.

3. What are window functions in SQL? Provide examples.

- Window functions perform calculations across a set of table rows related to the current row.

Example:

```
SELECT employee_id,
       department,
       salary,
       Rank()
         OVER (
                     partition BY
department
                ORDER BY salary
DESC) AS salary_rank
FROM   employee;
```

4. How would you optimize a query that takes too long to execute?

- Use Indexes on frequently queried columns.
- Avoid **SELECT * **; retrieve only required columns.
- Optimize JOINs by indexing keys.
- Use EXPLAIN PLAN to analyze query performance.
- Normalize the database structure.
- Avoid redundant subqueries and use CTEs.

5. Write a query to find duplicate records in a table.

```
SELECT column1,
    column2,
    Count(*)
FROM   table_name
GROUP  BY column1,
       column2
HAVING Count(*) > 1;
```

6. How do you handle NULL values in SQL?

- Use COALESCE() to replace NULLs:
- SELECT COALESCE(salary, 0) FROM employee;
- Use IS NULL / IS NOT NULL in conditions.
- Use IFNULL() (MySQL) or NVL() (Oracle).

7. Explain the difference between DELETE, TRUNCATE, and DROP.

- DELETE: Removes specific rows, can be rolled back, and logs each row deletion.
- TRUNCATE: Removes all rows, faster than DELETE, cannot be rolled back.
- DROP: Removes the entire table from the database.

8. What is a CTE (Common Table Expression), and how is it different from a subquery?

CTE: Temporary result set used in complex queries, improving readability.

Subquery: A nested query inside another query. Example CTE:

```
WITH salesdata
    AS (SELECT customer_id,
               Sum(amount) AS total_sales
        FROM   sales
        GROUP  BY customer_id)
SELECT *
FROM   salesdata
WHERE  total_sales > 1000;
```

9. Write a query to calculate the running total of sales for each month.

```
SELECT month,
       sales,
       Sum(sales)
         OVER (
           ORDER BY month) AS running_total
FROM   sales_table;
```

10. Explain the difference between INNER JOIN, LEFT JOIN, and FULL OUTER JOIN.

- INNER JOIN: Returns only matching records. LEFT JOIN: Returns all records from the left table and matching records from the right.
- FULL OUTER JOIN: Returns all records from both tables, filling non-matching records with NULLs. Example:

11. How would you use Python for data cleaning and transformation?

Python is widely used for data cleaning and transformation in data analysis and machine learning workflows. You can use libraries like pandas, NumPy, and re to perform various data preparation tasks efficiently.

1.Handling Missing Data

import pandas as pd

df = pd.read_csv("data.csv")

Check for missing values

print(df.isnull().sum())

Fill missing values with mean/median/mode

df["column_name"].fillna(df["column_name"].mean(), inplace=True)

Drop rows/columns with missing values

df.dropna(inplace=True)

2. Removing Duplicates

df.drop_duplicates(inplace=True)

3. Data Type Conversion df["date_column"] =

pd.to_datetime(df["date_column"]) df["numeric_column"] =

pd.to_numeric(df["numeric_column"], errors="coerce")

4. String Cleaning (Removing Special Characters, Lowercasing, etc.)

df["text_column"] = df["text_column"].str.lower().str.replace(r"[^a-zA-Z0-9]", " ",

regex=True)

5. Handling Outliers You can remove outliers or cap them

based on a threshold. # Remove rows with outliers df =

df[df['column_name'] < threshold_value]

Alternatively, cap outliers df['column_name'] =

df['column_name'].clip(lower=min_value, upper=max_value)

6. Data Transformation

Transform data, such as changing column names or creating new features:

Rename Columns:

```
df.rename(columns={'old_name': 'new_name'}, inplace=True)
```

Create New Columns:

```
df['new_column'] = df['column1'] + df['column2']
```

7. Filtering Data You can filter data based on certain conditions: df_filtered = df[df['column_name'] > 50] # Rows where column_name > 50

8. Handling Categorical Data Convert categorical data to numerical values using encoding techniques like Label Encoding or One-Hot Encoding.

```
# Label Encoding
df['encoded_column'] = df['category_column'].map({'Category1': 0, 'Category2': 1})
# One-Hot Encoding
df = pd.get_dummies(df, columns=['category_column'])
```

9. Normalization/Standardization You may need to scale numerical data for machine learning models.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df['scaled_column'] = scaler.fit_transform(df[['column_name']])
```

10. Date/Time Transformation Extract specific components from datetime columns or create new time-based features.

```
df['year'] = df['date_column'].dt.year
df['month'] = df['date_column'].dt.month
df['day_of_week'] = df['date_column'].dt.dayofweek
```

11. Save Cleaned Data After all transformations, save the cleaned dataset.

```
df.to_csv("cleaned_data.csv", index=False)
```

Summary

- Pandas is your main tool for data manipulation, cleaning, and transformation.
- Handle missing data, outliers, duplicates, and incorrect data types.
- Use encoding and scaling techniques for categorical and numerical features.
- Transform dates, filter data, and create new features as needed.

12.Write a Python script to connect to a database and fetch data using SQL queries.

Here's a Python script to connect to a MySQL database and fetch data using SQL queries. This script uses the mysql-connector-python library to establish the connection.

Steps in the Script:

1 . Install the required package (if not already installed):

```
pip install mysql-connector-python
```

2 .Establish a connection with the database.

3 .Execute an SQL query and fetch results.

.Handle exceptions and close the connection properly.

Python Script:

```
import mysql.connector


# Database connection details

db_config = {
   "host": "your_host",     # e.g., "localhost" or an IP address
   "user": "your_username", # e.g., "root"
   "password": "your_password",
   "database": "your_database"
}
   try:
      #Establishing connection
      conn = mysql.connector.connect(**db_config)

      if conn.is_connected():
         print("Connected to the database!")

      #Create  a  cursor  object
      cursor = conn.cursor()
```

```
#SQL query to fetch data
query = "SELECT * FROM your_table LIMIT 10;"

cursor.execute(query)


#Fetch and print results
results = cursor.fetchall()
for row in results:
    print(row)
except mysql.connector.Error as err:
   print(f"Error: {err}")
finally:
   # Close the connection
   if conn.is_connected():
      cursor.close() conn.close()
      print("Connection closed.")
```

Modifications for Different Databases:

- PostgreSQL: Use psycopg2
- SQL Server: Use pyodbc
- SQLite: Use sqlite3

13.Explain the difference between Pandas and PySpark for data manipulation.

Both Pandas and PySpark are popular Python libraries for data manipulation, but they are suited for different use cases. Here's a comparison:

Feature	Pandas	PySpark
Best For	Small to medium-sized datasets	Big data & distributed computing
Speed	Fast for small datasets	Faster for large datasets (distributed processing)
Data Size	Handles up to a few million rows efficiently	Handles terabytes of data across multiple machines
Parallelism	Single-threaded (limited by RAM)	Multi-threaded (distributed via Spark clusters)
Memory Usage	Stores all data in RAM	Uses disk storage & distributed memory
Ease of Use	Simple, intuitive API	More complex but scalable
Installation	Requires only pandas	Requires pyspark and Spark setup
Processing Engine	Works in-memory on a single machine	Uses Spark's distributed computing engine

When to Use?

Pandas → Best for small to medium datasets (Excel, CSV, databases).
PySpark → Best for large-scale data (big data, cloud-based, distributed processing).

14.How would you handle exceptions in a Python-based ETL pipeline?

In an ETL (Extract, Transform, Load) pipeline, handling exceptions is crucial for ensuring that the process runs smoothly and errors are managed properly. Here's how you can handle exceptions in a Python-based ETL pipeline.

1 . Using Try-Except Blocks

You can surround each ETL step (Extract, Transform, Load) with try-except blocks to catch specific errors and take appropriate actions like logging the error or retrying the process.

2 .General Structure for ETL Pipeline:

```
import logging
import time
# Setup logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
def extract():
    try:
        #Simulating data extraction (e.g., from a database or file)
        logging.info("Extracting data...")
        data = ["data1", "data2", "data3"]
        if not data:
            raise ValueError("No data extracted!")
        return data
    except Exception as e:
        logging.error(f"Error in extraction: {e}")
        raise # Re-raise the error after logging
def transform(data):
    try:
        #Simulating data transformation (e.g., cleaning, filtering)
        logging.info("Transforming data...")
        transformed_data = [item.upper() for item in data] # Example transformation
        return transformed_data
    except Exception as e:
        logging.error(f"Error in transformation: {e}")
        raise # Re-raise the error after logging
```

```
def load(data):
  try:
    #Simulating loading data (e.g., inserting into a database)
    logging.info("Loading data...")
    if not data:
      raise ValueError("No data to load!")
    #Assume data is successfully loaded
    logging.info(f"Data loaded: {data}")
  except Exception as e:
    logging.error(f"Error in loading: {e}")
    raise # Re-raise the error after logging
    run_etl()
    logging.info("ETL pipeline completed successfully.")
    break # Exit after successful completion
  except Exception as e:
    logging.error(f"ETL pipeline failed: {e}")
    logging.info("Retrying in 10 seconds...")
    time.sleep(10) # Retry after 10 seconds if it fails
```

Key Components of Exception Handling:

1. Logging: Use logging to capture detailed logs for debugging. It provides different levels like INFO, ERROR, WARNING, etc.
2. Specific Exceptions: Catch specific exceptions (e.g., ValueError, ConnectionError) to handle different scenarios separately.
3. Retries: If a step fails (e.g., network issue), retry it with a delay (time.sleep).
4. Raising Exceptions: After catching and logging an exception, raise it again if you want the pipeline to fail and stop, or handle it at a higher level.
5. Data Validation: Before moving to the next ETL step, check that the data is valid (e.g., non-empty).

3 . Advanced Exception Handling

For more advanced scenarios, you can:

- Use custom exception classes for specific errors.
- Integrate with a message queue (e.g., RabbitMQ) for reprocessing failed steps.
- Set up alerting mechanisms (e.g., sending an email or Slack notification) if the pipeline fails.

15.What libraries have you used for data processing in Python (e.g., Pandas, NumPy)?

1 . Pandas – Best for structured data (CSV, Excel, SQL tables)

✓ Data manipulation: DataFrame and Series
✓ Handling missing values: .fillna(), .dropna()
✓ Aggregations: .groupby(), .pivot_table()
✓ Merging & joining: .merge(), .concat()

2 . NumPy – Best for numerical computations & arrays

✓ Fast operations on large datasets
✓ Array handling: np.array(), np.reshape()
✓ Math functions: np.mean(), np.std(), np.linalg

3 . PySpark – Best for big data processing

✓ Distributed data processing with Spark

✓ Handling large datasets that don’t fit in memory

✓ Functions: DataFrame.select(), groupBy(), filter()

4 .Dask – Parallel computing for large Pandas-like datasets

✓ Works like Pandas but for larger-than-memory datasets

✓ Lazy execution for optimization

16.Describe the architecture of a cloud-based data warehouse like Snowflake or BigQuery.

A cloud-based data warehouse like Snowflake or BigQuery follows a distributed, scalable, and serverless architecture designed for high-performance analytics. Here’s a breakdown of their architectural components:

1 .Snowflake Architecture (3-Tier) Snowflake follows a multi-cluster shared data architecture with three key layers:

Compute Layer (Virtual Warehouses)

- Made up of virtual warehouses (clusters) that run queries.
- Each warehouse is independent, ensuring no resource contention.
- Supports automatic scaling (up/down based on workload).

Storage Layer

- Stores structured and semi-structured data (CSV, JSON, Parquet).
- Uses columnar storage for faster queries.
- Data is compressed, encrypted, and automatically managed.
- Decoupled from compute, allowing independent scaling.

Cloud Services Layer

- Manages query optimization, authentication, access control.
- Includes metadata management for tracking table statistics.
- Handles concurrent users and workload management.

Benefits of Snowflake:

✔ Auto-scaling & auto-suspend for cost savings
✔ Supports semi-structured data (JSON, Avro, etc.)
✔ Time travel feature for recovering past versions

2 . Google BigQuery Architecture (Serverless) BigQuery follows a serverless, columnar, and distributed architecture.

Storage Layer (Colossus)

- Uses columnar storage optimized for fast analytics.
- Supports automatic compression and partitioning.
- Data is stored in Google Cloud Storage (GCS).

Compute Layer (Dremel Execution Engine)

- Uses Dremel (Google’s query execution engine) for distributed SQL processing.
- Fully managed, auto-scaling compute.
- Queries are split into slots and executed in parallel.

Query Processing Layer

- Uses ANSI SQL with built-in machine learning (BigQuery ML).
- Supports federated queries (querying external sources like GCS, Bigtable).
- Offers BI Engine for in-memory analytics.

Benefits of BigQuery:

✔ Serverless → No infrastructure management

✔ Highly scalable and cost-effective (pay-per-query)

✔ Integration with Google AI & ML tools

Snowflake vs. BigQuery: Key Differences

Feature	**Snowflake** ❄️	**BigQuery**
Scaling	Manual & auto-scaling	Fully serverless Auto-scaling
Storage	Columnar, optimized for structured/ semi-structured data	Columnar, stored in Google Cloud
Pricing	Pay for storage + compute usage	Pay per query
Use Case	Best for high-performance, complex workloads	Best for on-demand analytics & real-time queries

Would you like a deeper comparison based on a specific use case?

17. What is the difference between OLAP and OLTP databases?

OLAP vs. OLTP

Feature	OLAP (Online Analytical Processing)	OLTP (Online Transaction Processing)
Purpose	Used for analytical querying and reporting	Used for day-to-day transactional operations
Data Structure	Typically stores large volumes of historical, summarized data	Stores real-time transactional data
Query Complexity	Complex queries, multi-dimensional analysis	Simple queries, mostly CRUD(Create, Read, Update, Delete) operations
Data Volume	Large datasets, often aggregating data over time	Small datasets, individual transaction records
Data Update Frequency	Infrequent updates (batch updates)	Frequent updates (real-time)
Examples	Data warehousing, business intelligence	Banking systems, e-commerce platforms
Performance Focus	Optimized for read-heavy operations (analysis, reports)	Optimized for write-heavy operations (transactions)
Normalization	Often denormalized for fast querying	Highly normalized to reduce data redundancy
Indexes	Less frequent, optimized for read performance	Frequent indexing for fast retrieval of transaction data
Concurrency	Lower concurrency, heavy read operations	High concurrency, many concurrent transactions

18. How do you ensure data quality during ETL processes?

Ensuring data quality during ETL (Extract, Transform, Load) processes is crucial for maintaining accuracy, consistency, and reliability. Here's how you can achieve it:

1 . Extract Phase – Validate Incoming Data
✔ Source Validation → Ensure data is extracted from trusted sources.
✔ Schema Validation → Check column names, data types, and constraints.
✔ Data Completeness → Ensure all expected records are extracted.
✔ Deduplication → Remove duplicate records at the extraction stage.

2 . Transform Phase – Cleaning & Standardization
✔ Handling Missing Values → Use imputation (fillna() in Pandas) or flag records for review.
✔ Data Type Consistency → Convert data into the correct formats (e.g., int, float, datetime).
✔ Outlier Detection → Identify and handle anomalies using statistical methods (e.g., Z-score).
✔ Business Rules Enforcement → Validate data against predefined rules (e.g., age cannot be negative).
✔ Normalization & Standardization → Convert data into a consistent format (e.g., lowercase emails).

3 . Load Phase – Integrity & Audits

✔ Primary Key Checks → Ensure uniqueness constraints are met.
✔ Referential Integrity → Validate foreign key relationships before inserting data.
✔ Row Count Validation → Compare the number of records before and after loading.
✔ Data Reconciliation → Cross-check transformed data against source data.

4 .Automated Quality Checks & Monitoring
✔ ETL Logging & Alerts → Capture errors in logs and send notifications for failures.
✔ Data Profiling Tools → Use Great Expectations, dbt, or Apache Griffin to track data quality.
✔ Unit Testing → Implement test cases using pytest or unittest for data validation.

By implementing these best practices, you can minimize errors and ensure high- quality data for analytics and decision-making!

19 .What is the role of Apache Kafka in data engineering?

Apache Kafka plays a key role in data engineering, particularly in real-time data streaming, event-driven architectures, and data integration. Here’s an overview of its role:

1 .Real-Time Data Streaming

Kafka is primarily used for building real-time data pipelines and streaming applications. It allows you to process data in motion, which is crucial for handling high-throughput, low-latency data feeds.

- Producers send data to Kafka topics.
- Consumers read from those topics in real-time.
- Useful for scenarios like IoT data, real-time analytics, and log processing.

2 .Event-Driven Architecture

Kafka enables event-driven architectures, where systems communicate through events. It decouples data producers and data consumers, allowing each to operate independently.

- Allows easy integration between various microservices.
- Ensures asynchronous communication and processing.

3 .Data Integration and Data Pipelines

Kafka acts as a central messaging layer in complex data engineering pipelines. It facilitates the integration of multiple data sources, including databases, third-party systems, and internal applications.

- Data can be sent from Kafka to data warehouses (e.g., Snowflake, BigQuery) for batch processing.
- Integrates seamlessly with ETL tools like Apache Flink, Apache Spark, and Kafka Streams for processing.

4 .Fault Tolerance and Scalability

Kafka provides built-in fault tolerance and scalability, ensuring high availability and reliability for data flows.

- Data is replicated across multiple brokers for fault tolerance.
- Kafka can scale horizontally by adding more brokers to the cluster.

5 .High Throughput and Low Latency

Kafka handles high-throughput and low-latency data streams, making it ideal for applications where speed is critical (e.g., financial transactions, recommendation engines).

- It can handle millions of messages per second with low latency, ensuring fast processing.

6 .Data Storage and Durability

Kafka offers durability by persisting data to disk, enabling long-term storage of messages. Unlike traditional message queues, Kafka can retain messages for configurable retention periods, allowing consumers to reprocess them as needed.

- Kafka's log-based storage allows for scalable retention policies, useful for audit logs, reprocessing data, or data archiving.

Common Use Cases of Apache Kafka:

- Log aggregation: Collect logs from various systems for centralized analysis.
- Metrics collection: Real-time metrics and monitoring of application performance.
- Real-time analytics: Real-time dashboards, fraud detection, and recommendation systems.
- Data synchronization: Synchronizing data across various systems (databases, applications).

In summary, Kafka is a powerful tool in data engineering for managing high-volume, real-time, and fault-tolerant data streams across distributed systems. It plays a critical role in building modern data architectures, particularly in streaming analytics, event sourcing, and data integration pipelines.

20.What is ETL? Explain its phases and tools you have worked with.

ETL (Extract, Transform, Load) ETL is a data integration process used to move data from various sources into a centralized data warehouse or data lake. It consists of three main phases: Extract, Transform, and Load.

1. Extract Phase

The Extract phase involves retrieving raw data from various source systems, which could include databases, APIs, flat files, or third-party services.

Key Steps:

- Connect to Source Systems: Data is pulled from multiple sources like relational databases, web services, cloud platforms, etc.
- Data Extraction: The raw data is captured, usually in a format like CSV, JSON, or XML.

Tools for Extraction:

- Python libraries (e.g., pandas, requests, pyodbc) for pulling data from APIs, databases.
- Apache Kafka for streaming real-time data.
- AWS Glue for serverless extraction from cloud storage.
- Talend for data extraction from different sources.

2 .Transform Phase

The Transform phase is where the raw data is cleaned, enriched, and converted into a format suitable for analysis. This is the most complex phase, as it involves applying business rules, data validation, and restructuring.

Key Steps:

- Data Cleaning: Handle missing values, duplicates, and outliers.
- Data Enrichment: Add additional data or attributes from other sources.
- Data Standardization: Convert data to a standard format (e.g., date formats, currency conversions).
- Data Aggregation: Summarize data for analytical purposes.
- Data Validation: Ensure data integrity and consistency.

Tools for Transformation:

- Pandas for Python-based data manipulation.
- Apache Spark for large-scale data transformations.
- dbt for SQL-based transformations in data warehouses.
- Talend for visual data transformation workflows.
- Airflow for orchestrating transformation tasks.

3 .Load Phase

The Load phase involves writing the transformed data into the destination, typically a data warehouse or data lake for further analysis.

Key Steps:

Bulk Load: Insert transformed data in large batches.

Incremental Load: Only insert new or updated data to improve efficiency.

Data Indexing: Create indices to speed up query performance.

Tools for Loading:

AWS Redshift or Google BigQuery for loading data into cloud data warehouses.

SQL Server or Oracle Database for traditional relational databases.

Apache Hive for storing data in Hadoop-based data lakes.

Pandas and SQLAlchemy for Python-based data loading.

Apache Nifi for automated ETL pipelines.

ETL Process Example:

1. Extract data from an API containing customer transaction records.
2. Transform the data by cleaning out null values, standardizing date formats, and enriching with geographic information.
3. Load the transformed data into a PostgreSQL database or AWS Redshift for analysis.

ETL Tools I Have Worked With:

- Python (with Pandas, NumPy, and SQLAlchemy) for handling small to medium ETL tasks and custom transformations.
- SQL for extracting and transforming data within relational databases.
- Apache Kafka for real-time data streaming and integration.
- Apache Airflow for orchestrating ETL pipelines and scheduling tasks.
- AWS Glue for serverless ETL jobs in the cloud.
- Power BI for transforming data within its in-built tools before loading it to dashboards.

ETL in Modern Data Engineering:

- In modern data engineering, ETL processes have become more automated, with tools like Apache Airflow for scheduling, dbt for transformation, and cloud-based solutions like AWS Glue or Google Cloud Dataflow for scalable data processing.

ETL is critical in building efficient data pipelines, ensuring data is clean, accurate, and available for downstream analytics.

Azure Data Factory (ADF)

Scenario:

Your organization wants to integrate ADF with an Event Hub to process real-time streaming data.

Questions:

- How would you set up ADF to process data from an Event Hub?
- What are the limitations of using ADF for real-time processing?
- How does ADF integrate with other Azure services like Stream Analytics for real-time use cases?

Scenario: You have a large dataset stored in Azure Data Lake that needs to be processed by date partitions.

Questions:

- How would you design a pipeline to process data in partitions?
- What are the advantages of data partitioning in ADF?
- How would you use dynamic expressions to process each partition dynamically?

Scenario: You have multiple pipelines, and one pipeline should only start after the successful completion of another.

Questions:

- How would you implement dependencies between pipelines in ADF?
- What are the pros and cons of using execute pipeline activity versus trigger chaining?
- How would you handle scenarios where one pipeline fails but others should continue?

Scenario: You have a master pipeline that orchestrates the execution of multiple child pipelines. Some child pipelines are dependent on the output of others.
Questions:

- How would you design the master pipeline to handle dependencies between child pipelines?
- What are the differences between the "Wait" and "If Condition" activities in this context?
- How would you monitor and troubleshoot issues in a complex pipeline execution?

Scenario: Your pipeline needs to be triggered whenever a file is uploaded to a specific container in Azure Blob Storage.
Questions:

- How would you set up an event-based trigger in ADF?
- What are the advantages and limitations of using event-based triggers?
- How would you handle scenarios where multiple files are uploaded simultaneously?

Scenario: Your pipeline frequently encounters transient network issues when copying data from an on-premises database to Azure SQL Database.
Questions:

- How would you implement retry logic to handle transient errors?
- What settings in ADF activities allow for retries and delays?
- How would you monitor and alert on excessive retries in pipeline executions?

Scenario: You are required to process only the files uploaded in the last 24 hours from Azure Blob Storage.
Questions:

- How would you filter files based on their upload timestamp?
- What expressions or functions would you use to calculate time-based conditions?
- How would you handle time zone differences when filtering files?

Scenario: Your pipeline processes files daily from Azure Blob Storage and loads them into Azure SQL Database. Duplicate files occasionally appear in the storage container.
Questions:

- How would you design a pipeline to identify and skip duplicate files?
- How would you use metadata (e.g., file names or hashes) to track processed files?
- How would you recover if a duplicate file causes partial data corruption?

Scenario: Your pipelines process a daily batch of files, and you need to ensure that if a pipeline fails, it can resume from the last successfully processed file.
Questions:

- How would you implement state management to track processed files?
- What role do control tables play in this scenario?
- How would you ensure idempotency in pipeline executions?

Scenario: You need to process data from multiple Azure regions and consolidate it into a central Azure Data Lake in a cost-efficient manner.
Questions:

- How would you design a pipeline to handle geo-distributed data?
- What are the cost and performance considerations for cross-region data transfers?
- How can you use regional Integration Runtimes to optimize performance?

Scenario: You are responsible for ensuring data quality before loading it into the destination system. This includes null checks, duplicate checks, and threshold-based validations.
Questions:

- How would you implement data quality checks in ADF?
- What role do Data Flow transformations like Filter, Aggregate, and Exists play in these checks?
- How would you handle rows that fail quality checks?

Scenario: Your pipeline processes sensitive financial data that needs to be encrypted during transit and at rest.
Questions:

- How would you ensure end-to-end encryption for sensitive data in ADF?
- How can you use Azure Key Vault for managing credentials and encryption keys?
- What security best practices would you follow to secure data pipelines?

Scenario: You need to process data stored in partitions (e.g., year/month/day folders), but only for specific time ranges based on runtime parameters.
Questions:

- How would you configure the Copy Activity or Data Flow to read specific partitions dynamically?
- What functions or expressions would you use to skip unnecessary partitions?
- How can you optimize pipeline performance when dealing with highly partitioned data?

Scenario: You are tasked with processing unstructured data like log files or free-form text stored in Azure Blob Storage.
Questions:

- How would you handle unstructured data in ADF?
- What external tools (e.g., Databricks, Cognitive Services) can you integrate with ADF for parsing or extracting insights?
- How would you transform this data into a structured format for downstream processing?

Scenario: You have a pipeline with multiple parallel activities, and one of the activities fails intermittently due to source system issues.
Questions:

- How would you implement exception handling for individual activities in ADF?
- How can you ensure that the pipeline continues processing unaffected branches?
- What strategies would you use to retry or log failed activities?

Scenario: Your pipeline needs to process files dynamically based on folder structure and file patterns in Azure Data Lake.
Questions:

- How would you use wildcard file paths in ADF to process specific files?
- How can you create folders dynamically based on runtime parameters?
- What are the challenges of managing large numbers of folders and files, and how would you address them?

Scenario: Your team needs to collaborate with another team to build pipelines that share dependencies and datasets.
Questions:

- How would you manage shared resources (e.g., Linked Services, Datasets) across teams?
- What strategies would you use to avoid conflicts in pipeline development?
- How can Git integration help streamline collaboration between teams?

Scenario: You need to notify stakeholders immediately when a pipeline or activity fails, including error details.
Questions:

- How would you implement real-time error notifications using Azure Monitor or Logic Apps?
- How can you configure email or SMS alerts for pipeline failures?
- What are the key metrics and logs to monitor for proactive issue detection?

Scenario: You are part of a large organization with multiple teams working on separate ADF projects. Central governance is required for Linked Services, triggers, and naming conventions.
Questions:

- How would you implement centralized governance for ADF projects?
- How can Azure Policy or Resource Manager templates enforce naming conventions?
- What strategies would you use to manage shared Linked Services across teams?

Scenario: You need to load data from multiple sources into corresponding tables in a destination, with dynamic schema mapping.
Questions:

- How would you configure dynamic sink mapping in a Copy Activity?
- How can parameterization help in automating schema mapping?
- What challenges might you encounter when handling mismatched schemas?

Scenario: You need to implement data retention policies for your pipelines, ensuring that data older than a certain period is deleted or archived.
Questions:

- How would you automate data retention policies in ADF?
- What role does the Delete Activity play in this process?
- How can you monitor and validate the successful execution of retention policies?

Scenario: You need to process semi-structured data (e.g., JSON files with varying schemas) stored in Azure Blob Storage.
Questions:

- How would you handle schema variability while processing semi-structured data in ADF?
- What transformations would you use in Mapping Data Flows to parse JSON data?
- How can you flatten hierarchical data structures for downstream consumption?

Scenario: Your pipeline processes files in batches based on their upload time, dynamically creating batches for every 24-hour period.
Questions:

- How would you design a pipeline to identify and process dynamic file batches?
- How can you use metadata from Azure Blob Storage to determine batch boundaries?
- What challenges might arise in handling late-arriving files, and how would you address them?

Scenario: Your pipelines need to adapt dynamically based on metadata, such as file names, schema definitions, or transformation rules stored in a database.
Questions:

- How would you design a metadata-driven pipeline in ADF?
- How can Lookup and ForEach Activities be used to retrieve and apply metadata?
- What are the advantages of a metadata-driven approach in large-scale ETL processes?