**Proposed solution**

**Overview:**

To facilitate the transition from XYZ Datawarehouse to a centralized cloud-based Azure storage system, we propose utilizing Azure Data Factory (ADF) for daily incremental data ingestion, Azure Data Lake Storage (ADLS) for data storage, and Azure Databricks for data processing. This approach will ensure that the Marketing System (MS) team has seamless access to the necessary datasets for dashboard generation. Additionally, implementing a medallion architecture (Bronze, Silver, Gold layers) will ensure efficient data management and accessibility.

**Architecture:**

We will employ a three-tier architecture: Bronze, Silver, and Gold layers within Azure Data Lake Storage (ADLS). The following diagram and description outline the proposed architecture:

|  |
| --- |
|  |

**Bronze Layer:**

* **Purpose**: Staging area for raw, incremental data ingested from various on-prem systems and other sources.
* **Process**: Azure Data Factory (ADF) orchestrates the data ingestion process, transferring daily snapshots into the Bronze layer. This layer serves as the initial storage for raw data, optimizing storage costs by leveraging lower-tier storage options within ADLS.

**Silver Layer:**

* **Purpose**: Storage of preprocessed, cleaned, and transformed data.
* **Process**: Azure Databricks, utilizing PySpark, processes data from the Bronze layer, transforming it into a more structured and curated format. This layer ensures data quality and consistency, making it suitable for analytical purposes.

**Gold Layer:**

* **Purpose**: Storage of further processed, aggregated, and refined data, including fact and dimension tables.
* **Process**: Further transformations and aggregations are performed in this layer, making the data ready for reporting and dashboarding. This layer is optional and can be omitted if the Silver layer suffices for the reporting needs.

**Components:**

**Azure Data Factory (ADF):**

* **Function**: Automates the execution of data pipelines at specified intervals, ensuring timely data ingestion.
* **Capabilities**: Implements incremental loading strategies such as watermarking or Change Data Capture (CDC) to identify and transmit only new or changed data since the last load. It handles large-scale data movement, providing scalability and performance optimizations.
* **Monitoring**: ADF provides monitoring capabilities to track data loading status, delays, and failures via the Azure portal or email notifications.

**Azure Databricks:**

* **Function**: Performs data transformations and processing.
* **Capabilities**: Leveraging Apache Spark, Databricks efficiently executes ETL tasks, including complex transformations and aggregations. It ensures timely processing of transactional data, enabling the calculation of account balances and other metrics. Databricks notebooks will be integrated into ADF for monitoring and automation purposes.

**Data Sharing and Security:**

**Access Management:**

* **Role-Based Access Control (RBAC):** Implement RBAC to define and enforce user permissions and access policies. This ensures that only authorized users can access specific data and resources.
* **Microsoft Entra:** Use Microsoft Entra (formerly Azure Active Directory) to manage user identities and create security groups. Assign appropriate roles and permissions to these groups to streamline access management.
* **Data Masking:** Use of data masking technique ensures that users can only view data necessary for their role, protecting personally identifiable information (PII) and other sensitive data.

**Data Sharing:**

* **Azure Data Share:** Use Azure Data Share for secure and efficient data sharing. This service allows you to share data from your Azure Data Lake Storage (ADLS) with the MS team or other external parties while maintaining control over data access.
* **Folder-Level Access:** Provide granular access to specific folders within ADLS. Configure access permissions at the folder level to ensure that the MS team can access only the datasets relevant to their work.

**Governance and Source Control:**

* **Secret Management:** Utilize Azure Key Vault to securely store and manage sensitive information such as connection strings, API keys, and certificates.
* **Code Management:** Implement Azure DevOps for source control and CI/CD pipelines to manage code deployment and migration.

**Optional Method:**

* **Azure Blob Storage**: Consider using Azure Blob Storage as an alternative to ADLS for cost-effective storage of unstructured data, such as text files and object data. This option may be suitable if the data does not require the hierarchical namespace and advanced features of ADLS.

**Account Balance Calculation:**

Task is to calculate balance of each account number after the transaction. Used pyspark transformation to fullfill the requirement. Here is step by steps process of implimentation.

**Step 1:**

Importing necessry Libraries

from pyspark.sql import SparkSession

from pyspark.sql.functions import sum as spark\_sum, when, col

from pyspark.sql.window import Window

**Step 2:**

Create Spark session for creating DataFrame and further transformation operations

spark = SparkSession.builder \

    .appName("Transaction Balance Calculation") \

    .getOrCreate()

**Step 3:**

Creating sample data for calculation and create Data frame from it.

# Sample data

data = [

    ("2023-01-01", 100, "Credit", 1000),

    ("2023-01-02", 100, "Credit", 1500),

    ("2023-01-03", 100, "Debit", 1000),

    ("2023-01-02", 200, "Credit", 3500),

    ("2023-01-03", 200, "Debit", 2000),

    ("2023-01-04", 200, "Credit", 3500),

    ("2023-01-13", 300, "Credit", 4000),

    ("2023-01-14", 300, "Debit", 4500),

    ("2023-01-15", 300, "Credit", 1500)

]

# Creating dataframe

df = spark.createDataFrame(data, ["TransactionDate", "AccountNumber", "TransactionType", "Amount"])

**Step 4:**

Adding additional columns for debit and credit. If the TransactionType is "Debit", DebitAmount is set to Amount, otherwise it is 0. Similarly, if the TransactionType is "Credit", CreditAmount is set to Amount, otherwise it is 0.

df = df.withColumn("DebitAmount", when(col("TransactionType") == "Debit", col("Amount")).otherwise(0)) \

       .withColumn("CreditAmount", when(col("TransactionType") == "Credit", col("Amount")).otherwise(0))

**Step 5:**

A window specification is defined to partition the data by AccountNumber and order it by TransactionDate.

# This specifies the window frame as from the beginning of the partition to the current row.

window\_spec = Window.partitionBy("AccountNumber").orderBy("TransactionDate")\

      .rowsBetween(Window.unboundedPreceding, 0)

**Step 6:**

Running totals for DebitAmount and CreditAmount are calculated using the window specification. The CurrentBalance is then calculated as the difference between RunningCredit and RunningDebit.

df = df.withColumn("RunningDebit", spark\_sum("DebitAmount").over(window\_spec)) \

       .withColumn("RunningCredit", spark\_sum("CreditAmount").over(window\_spec)) \

       .withColumn("CurrentBalance", col("RunningCredit") - col("RunningDebit"))

**Step 7:**

Removing unwanted columns and print the output

df = df.drop("DebitAmount", "CreditAmount", "RunningDebit", "RunningCredit")

df.show()

**Output:**

