Week X

Capstone Project

Intermediate Data Engineer in Azure
Trainer: Balazs Balogh

CUBIX
INSTITUTE OF TECHNOLOGY

Slowly Changing Dimension

```python
from delta.tables import DeltaTable
from pyspark.sql import DataFrame, SparkSession

from cubix_data_engineer_capstone.utils.config import STORAGE_ACCOUNT_NAME


def scd1(spark: SparkSession, container_name: str, file_path: str, new_data: DataFrame, primary_key: str):
    """Slowly Changing Dimension Type 1 implementation. Compare the master Delta table with the "new_data",
    if there are changes on the given primary key, then update, if it's not found in master, then insert.

    :param spark:            SparkSession.
    :param container_name:   Name of the container holding the Delta table.
    :param file_path:        Path to the Delta table.
    :param new_data:         DataFrame with the new data.
    :param primary_key:      Primary Key to join the Master Delta table with the new data.
    """
    master_path = f"abfss://{container_name}@{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net/{file_path}"
    delta_master = DeltaTable.forPath(spark, master_path)

    (
        delta_master
        .alias("master")
        .merge(
            new_data.alias("updates"),
            f"master.{primary_key} = updates.{primary_key}"
        )
        .whenMatchedUpdateAll()
        .whenNotMatchedInsertAll()
        .execute()
    )
```

Optional – GitHub Actions CICD pipeline

CI/CD stands for **Continuous Integration** and **Continuous Delivery/Deployment**. It’s a set of practices and tools that help developers deliver code changes more frequently and reliably.

1. Continuous Integration (CI)

- **What it is**:
    - Developers frequently merge their code changes into a shared repository (e.g., GitHub).
    - Each merge triggers an automated build and test process to catch issues early.
- **Key Benefits**:
    - Early detection of bugs and conflicts.
    - Improved collaboration among team members.
    - Faster feedback loop for developers.

2. Continuous Delivery (CD)

- **What it is**:
    - Automates the process of deploying code changes to staging or production environments after CI.
    - Ensures the code is always in a deployable state.
- **Key Benefits**:
    - Reduces manual errors during deployment.
    - Enables faster and more reliable releases.
    - Allows for quick rollbacks if something goes wrong.

Optional – GitHub Actions CICD pipeline

3. Continuous Deployment (CD)

- **What it is**:
    - An extension of Continuous Delivery where every code change that passes CI is automatically deployed to production.
- **Key Benefits**:
    - Fully automated pipeline from code commit to production.
    - Enables rapid delivery of new features and fixes.

CI/CD Pipeline Workflow

1. **Code Commit**:
    - Developers push code changes to a version control system (e.g., Git).
2. **Automated Build**:
    - The CI server (e.g., Jenkins, GitHub Actions) builds the application.
3. **Automated Testing**:
    - Unit tests, integration tests, and other checks are run.
4. **Deployment to Staging**:
    - If tests pass, the code is deployed to a staging environment for further testing.
5. **Deployment to Production**:
    - In Continuous Delivery, this step is manual. In Continuous Deployment, it's automatic.

Optional – GitHub Actions CICD pipeline

```yaml
name: Deploy Capstone project

on:
  push:
    branches:
      - master
  workflow_dispatch:

jobs:
  deploy:
    runs-on: windows-latest
    env:
      DATABRICKS_HOST: ${{ secrets.DATABRICKS_HOST }}
      DATABRICKS_TOKEN: ${{ secrets.DATABRICKS_TOKEN }}
    steps:
      - name: Checkout Repository
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: "3.11"

      - name: Install Poetry
        run: |
          python -m pip install --upgrade pip
          pip install poetry

      - name: Install Dependencies
        run: |
          poetry install

      - name: Install pre-commit hooks
        run: |
          poetry run pre-commit install

      - name: Run pre-commit hooks
        run: |
          poetry run pre-commit run --all-files

      - name: Run unit tests
        run: |
          poetry run pytest tests

      - name: Install Databricks CLI
        run: |
          pip install databricks-cli

      - name: Build and Deploy to Databricks
        run: |
          pwsh -File .\cubix_data_engineer_capstone\upload_latest_whl.ps1
```

deploy.yml, goes into root/.github/workflows. This is for GitHub Actions. Contains the steps we want it to run after pushing code to the master (main) repository.
Check out the repo, set up python, install poetry and the dependencies, run the pre-commit hooks, and unit tests, then build the package and deploy to Databricks.

Optional – GitHub Actions CICD pipeline

```
param(
    [string]$ProfileName
)

# Configuration
$dbfsPath = "dbfs:/mnt/packages"
$distPath = "$PSScriptRoot\..\dist"

# Step 1: Change to the directory where the script is located (project root)
Write-Output "Changing to the project root directory..."
Set-Location -Path $PSScriptRoot

# Step 2: Build the wheel using Poetry
Write-Output "Building wheel using Poetry..."
poetry build -f wheel
if ($LASTEXITCODE -ne 0) {
    Write-Output "Failed to build the wheel. Exiting."
    exit 1
}
Write-Output "Wheel build complete."

# Prepare Databricks profile arguments if provided
$profileArgs = @()
if ($ProfileName) {
    $profileArgs = @("--profile", $ProfileName)
    Write-Output "Using Databricks profile: $ProfileName"
}
```

```
# Step 3: Clear the DBFS packages folder
Write-Output "Truncating $dbfsPath..."
databricks fs rm -r $dbfsPath @profileArgs
if ($LASTEXITCODE -ne 0) {
    Write-Output "Failed to truncate DBFS path. Exiting."
    exit 1
}

databricks fs mkdirs $dbfsPath @profileArgs
if ($LASTEXITCODE -ne 0) {
    Write-Output "Failed to create DBFS directory. Exiting."
    exit 1
}
Write-Output "Truncated $dbfsPath."
```

Modify the upload_latest_whl.ps1 to handle also local and GitHub Actions deployments.

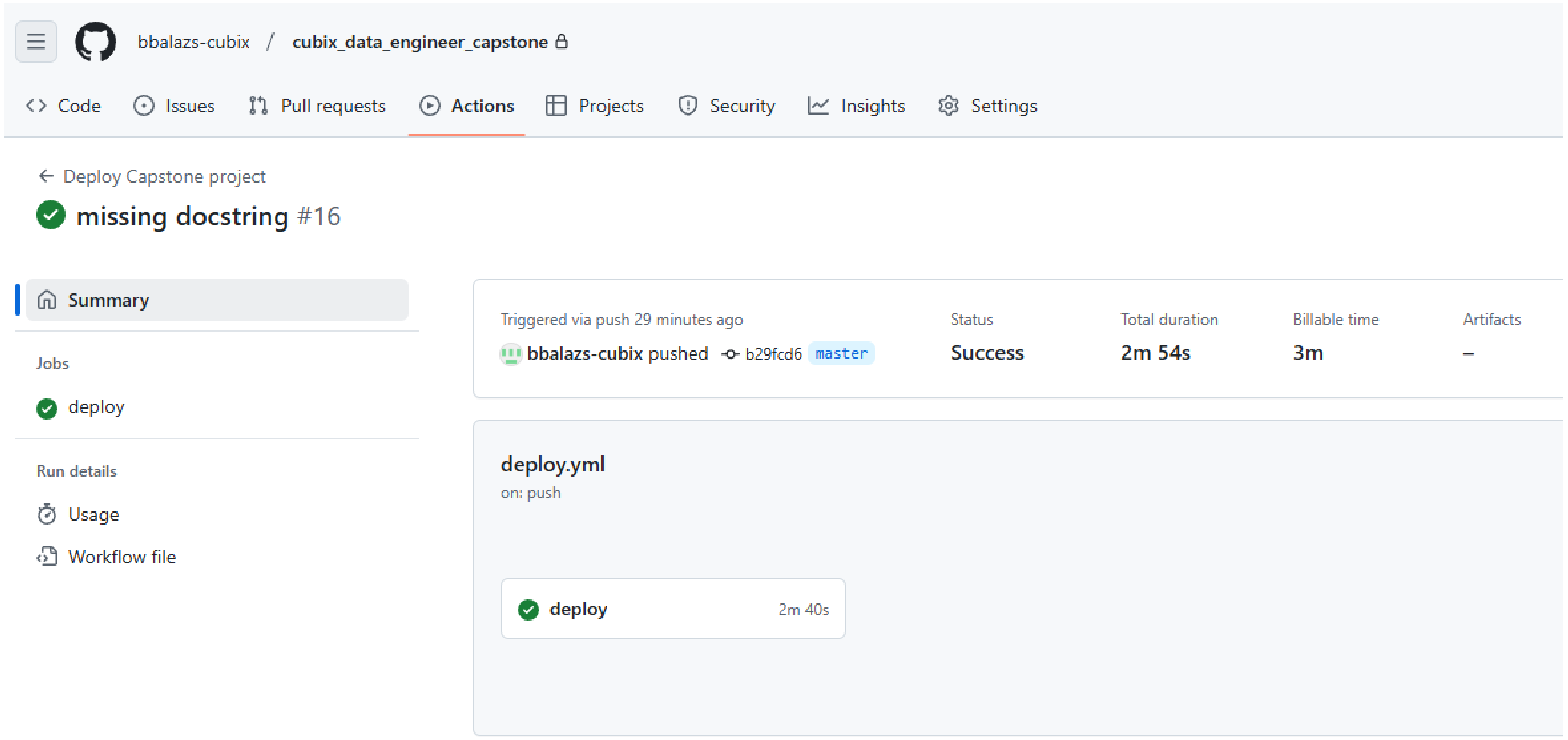
Optional – GitHub Actions CICD pipeline

```
# Step 4: Find the latest .whl file in the dist folder
Write-Output "Finding the latest .whl file in $distPath..."
$latestWhl = Get-ChildItem -Path $distPath -Filter *.whl | Sort-Object LastWriteTime
-Descending | Select-Object -First 1
if (-not $latestWhl) {
    Write-Output "No .whl files found in $distPath. Exiting."
    exit 1
}
Write-Output "Latest .whl file found: $($latestWhl.FullName)"

# Step 5: Copy the latest .whl file to DBFS
Write-Output "Copying $($latestWhl.FullName) to $dbfsPath..."
databricks fs cp $latestWhl.FullName "$dbfsPath/" --overwrite @profileArgs
if ($LASTEXITCODE -ne 0) {
    Write-Output "Failed to copy the .whl file to DBFS. Exiting."
    exit 1
}
Write-Output "Copied $($latestWhl.FullName) to $dbfsPath."

# Completion message
Write-Output "Latest .whl file built and uploaded to $dbfsPath successfully."
```

CUBIX
INSTITUTE OF TECHNOLOGY

Optional – GitHub Actions CICD pipeline

After either a push or a manual trigger the deployment can be followed on the repository's page (the "workflow_dispatch" parameter in the deploy.yml needed for the ability of manual deployment).


bbalazs-cubix / cubix_data_engineer_capstone

← Deploy Capstone project

missing docstring #16

Triggered via push 29 minutes ago	Status	Total duration	Billable time	Artifacts
bbalazs-cubix pushed -o- b29fcd6 master	Success	2m 54s	3m	–

deploy.yml
on: push

deploy 2m 40s

Medallion Architecture III. – Gold Layer – Wide Sales

```python
import pyspark.sql.functions as sf
from pyspark.sql import DataFrame


def _join_master_tables(
        sales_master: DataFrame,
        calendar_master: DataFrame,
        customers_master: DataFrame,
        products_master: DataFrame,
        product_subcategory_master: DataFrame,
        product_category_master: DataFrame
) -> DataFrame:
    """Join the master DataFrames to the Sales Master.
    Drop Date, ProductSubCategoryKey and ProductCategoryKey to avoid duplicate columns.

    :param sales_master:                Master DataFrames for Sales.
    :param calendar_master:             Master DataFrames for Calendar.
    :param customer_master:             Master DataFrames for Customer.
    :param products_master:             Master DataFrames for Products.
    :param product_subcategory_master:  Master DataFrames for Product Subcategory.
    :param product_category_master:     Master DataFrames for Product Category.
    :return:                            Sales Master with all the joined DataFrames.
    """
```

Wide Sales, as the name suggests is a wide table, or One Big Table (OBT).

OBT is a data modeling approach often used in data warehousing and analytics. It involves consolidating data from multiple sources into a single, denormalized table to simplify querying (by reducing joins) and improve performance.

Medallion Architecture III. – Gold Layer – Wide Sales

```
    return (
        sales_master
        .join(calendar_master, sales_master["OrderDate"] == calendar_master["Date"], how="left")
        .drop(calendar_master["Date"])
        .join(customers_master, on="CustomerKey", how="left")
        .join(products_master, on="ProductKey", how="left")
        .join(
            product_subcategory_master,
            products_master["ProductSubCategoryKey"] ==
product_subcategory_master["ProductSubCategoryKey"],
            how="left"
        )
        .drop(product_subcategory_master["ProductSubCategoryKey"])
        .join(
            product_category_master,
            product_category_master["ProductCategoryKey"] ==
product_subcategory_master["ProductCategoryKey"],
            how="left"
        )
        .drop(product_category_master["ProductCategoryKey"])
    )
```

Medallion Architecture III. – Gold Layer – Wide Sales

```
def get_wide_sales(
    sales_master: DataFrame,
    calendar_master: DataFrame,
    customers_master: DataFrame,
    products_master: DataFrame,
    product_subcategory_master: DataFrame,
    product_category_master: DataFrame
) -> DataFrame:
    """
    1. Join the Master tables.
    2. Convert the MaritalStatus and Gender to human readable format.
    2. Calculate SalesAmount, HighValueOrder, Profit.

    :param sales_master:                Master DataFrames for Sales.
    :param calendar_master:             Master DataFrames for Calendar.
    :param customer_master:             Master DataFrames for Customer.
    :param products_master:             Master DataFrames for Products.
    :param product_subcategory_master:  Master DataFrames for Product Subcategory.
    :param product_category_master:     Master DataFrames for Product Category.
    :return:                            The joined DataFrame with the additional columns.
    """
```

Medallion Architecture III. – Gold Layer – Wide Sales

```
wide_sales_df = _join_master_tables(
    sales_master,
    calendar_master,
    customers_master,
    products_master,
    product_subcategory_master,
    product_category_master
)

calculate_sales_amount = sf.col("OrderQuantity") * sf.col("ListPrice")
calculate_high_value_order = sf.col("SalesAmount") > 10000
calculate_profit = sf.col("SalesAmount") - (sf.col("StandardCost") * sf.col("OrderQuantity"))

return (
    wide_sales_df
    .withColumn(
        "MaritalStatus",
        sf.when(sf.col("MaritalStatus") == 1, "Married")
        .when(sf.col("MaritalStatus") == 0, "Single")
        .otherwise(None)
    )
    .withColumn(
        "Gender",
        sf.when(sf.col("Gender") == 1, "Male")
        .when(sf.col("Gender") == 0, "Female")
        .otherwise(None)
    )
    .withColumn("SalesAmount", calculate_sales_amount)
    .withColumn("HighValueOrder", calculate_high_value_order)
    .withColumn("Profit", calculate_profit)
)
```