Incremental Data Processing

Data Pipelines with Delta Live Tables

1. Identify the components necessary to create a new DLT pipeline.
    - The live table defined via Python/SQL in a notebook
    - The pipeline defined via Workflows
    - config (params, etc.)
    - cluster (autocreated when first starting your pipeline)

2. Identify the purpose of the target and of the notebook libraries in creating a pipeline.
    - target is the schema/database you want to put the tables in
    - notebook libraries for specifying and/or defining the table select statements

3. Compare and contrast triggered and continuous pipelines in terms of cost and latency
    - triggered pipelines are cheaper since they aren't continually run.
    - latency is higher for triggered since we have to spin up the cluster first.

4. Identify which source location is utilizing Auto Loader.
   - cloud files uses auto loader

```
@dlt.table
def orders_bronze():
    return (
        spark.readStream
            .format("cloudFiles")
            .option("cloudFiles.format", "json")
            .option("cloudFiles.inferColumnTypes", True)
            .load(f"{source}/orders")
            .select(
                F.current_timestamp().alias("processing_time"),
                F.input_file_name().alias("source_file"),
                "*"
            )
    )
```

5. Identify a scenario in which Auto Loader is beneficial. [1]
    - optimised for file discovery in cloud native storage

- has schema evolution

6. Identify why Auto Loader has inferred all data to be STRING from a JSON source 1
    - JSON inputs do not enforce schemas natively. To avoid schema mismatch, it is initially inferred as string. Same case with CSVs. For Avro/Parquet, schema is included with the input file.

7. Identify the default behavior of a constraint violation
    - Default behavior is to record the violating records ONLY (`dlt.expect`)
8. Identify the impact of ON VIOLATION DROP ROW and ON VIOLATION FAIL UPDATE for a constraint violation
    - `dlt.expect_or_fail` fails the pipeline at this step
    - `dlt.expect_or_drop` drops the row with violation

```
@dlt.table
@dlt.expect_or_fail("valid_id", "customer_id IS NOT NULL")
@dlt.expect_or_drop("valid_operation", "operation IS NOT NULL")
@dlt.expect("valid_name", "name IS NOT NULL or operation = 'DELETE'")
@dlt.expect("valid_adress", """
    (address IS NOT NULL and
    city IS NOT NULL and
    state IS NOT NULL and
    zip_code IS NOT NULL) or
    operation = "DELETE"
    """)
@dlt.expect_or_drop("valid_email", """
    rlike(email, '^([a-zA-Z0-9_\\\\-\\\\.]+)@([a-zA-Z0-9_\\\\-
\\\\.]+)\\\\.([a-zA-Z]{2,5})$') or
    operation = "DELETE"
    """)
def customers_bronze_clean():
    return (
        dlt.read_stream("customers_bronze")
    )
```

9. Explain change data capture and the behavior of APPLY CHANGES INTO 1
    - changes in source table data will consequently apply changes into target table
    - default is SCD type 1 (simply changing records)

```
dlt.create_target_table(
    name = "customers_silver")

dlt.apply_changes(
    target = "customers_silver",
    source = "customers_bronze_clean",
    keys = ["customer_id"],
    sequence_by = F.col("timestamp"),
    apply_as_deletes = F.expr("operation = 'DELETE'"),
    except_column_list = ["operation", "source_file", "_rescued_data"])
```

10. Query the events log to get metrics, perform audit logging, examine lineage. 1

    ```
    display(
            spark.sql(f"SELECT * FROM delta.`{D
    A.paths.storage_location}/system/ events`"))
    ```

11. Troubleshoot DLT syntax: Identify which notebook in a DLT pipeline produced an error, identify the need for LIVE in create statement, identify the need for STREAM in from clause.
    - for SQL, we need to use LIVE keyword to specify. for Python, we need to use the @dlt.table() decorator
    - for SQL, we use `cloud_files()` or `STREAM()` in the FROM clause. for Python, we use `readStream()` and `dlt.read_stream()`
        - Kung magbabasa palang sa source, we use `cloud_files()` and `readStream()`. Pero kung kapwa live table na ang babasahin, we use `STREAM()` and `dlt.read_stream()`.

Info on this topic is slightly lacking, kasi GCP quota does not allow me to actually provision a workflow compute cluster (job compute). Parang bawal kasi na single node and gamitin, eh isa isa lang node na binibigay ni Google Kubernetes Engine (GKE).