ELT With Spark SQL and Python

For the `${var}` syntaxing, refer to Databricks widgets, which is a way to register and use parameters in the SQL context. [1] Similarly we can invoke regular python variables for use with PySpark.

1. Extract data from a single file and from a directory of files. [1]

   ```
   # Single file
   SELECT * FROM json.`${path}/001.json`

   # Multiple similar files in directory (takes jsons)
   SELECT * FROM json.`${path}`
   ```

2. Identify the prefix included after the FROM keyword as the data type.
   - Delta (default), json, parquet, orc, avro, json ,csv, text, binaryFile

3. Create a view, a temporary view, and a CTE as a reference to a file
   - View - saved query accessible from other notebooks. As long as the NB uses the same database and consequently the same catalog.
   - Temp view - only persists within the spark session (within the notebook). Exception is if we use %run to reference another NB, which in turn gives us access to the environment variables of that other notebook.
   - CTE - only persists within the code cell

   ```
   # VIEW, TEMP VIEW
   CREATE OR REPLACE VIEW event_view
   AS SELECT * FROM json.`${path}`

   # CTE
   WITH cte AS(
     SELECT * FROM json.`${path}`
   )
   SELECT col1, col2 FROM cte
   ```

4. Identify that tables from external sources are not Delta Lake tables. [1]
   - External tables by default are not supported by Delta format
   - Check the format using the ff. under the `key = Provider`

```
DESCRIBE TABLE EXTENDED table;
```

5. Create a table from a JDBC connection and from an external CSV file. [1]
    - For DBFS, specifying LOCATION results in an EXTERNAL table while not specifying it results in MANAGED (need to check for ABFSS, ADLS, etc.)
    - This is because specifying location means specifying a source file that has existing data.
    - As long as the data file is created outside of Databricks it is EXTERNAL, regardless of where you store it.

```
# Specify external CSV
CREATE TABLE IF NOT EXISTS sales_csv
  (${schema})
USING CSV
OPTIONS (
  header = "true",
  delimiter = "|"
)
LOCATION "${path}";

# Specify JDBC connection url

CREATE TABLE users_jdbc
USING JDBC
OPTIONS (
  url = "jdbc:sqlite:${path}",
  dbtable = "users"
);
```

6. Identify how the count_if function can be used
7. Identify how the count where x is null can be used

```
# Equivalent lines of code
SELECT count_if(email IS NULL) FROM users;
SELECT count(*) FROM users WHERE email IS NULL;
```

8. Identify how the count(row) skips NULL values.
    - `count(*)` counts all regardless of nulls
    - `count(col)` disregards nulls in that column

9. Deduplicate rows from an existing Delta Lake table.
    - `DISTINCT(*)`

10. Create a new table from an existing table while removing duplicate rows.
    - CTAS (`CREATE TABLE AS ...`) statement with DISTINCT or GROUP BY

11. Deduplicate a row based on specific columns.
    - `GROUP BY` specific columns to only dedup based on specific columns. (Need to use aggregate functions with other columns - `MAX`, `MIN`, etc)

12. Validate that the primary key is unique across all rows.
    - `GROUP BY pkey` then count if each pkey value occurs only once

13. Validate that a field is associated with just one unique value in another field.
    - `GROUP BY field 1` and count if there is only one occurence in the other field

14. Validate that a value is not present in a specific field.
    - `WHERE column = value` should return no rows

15. Cast a column to a timestamp.

    ```
    CAST(int_value AS timestamp)
    ```

16. Extract calendar data from a timestamp.

    ```
    DATE_FORMAT(timestamp, "MMM d, yyyy")
    ```

17. Extract a specific pattern from an existing string column.

    ```
    REGEXP_EXTRACT(email, "(?<=@).+", 0) AS email_domain
    ```

18. Utilize the dot syntax to extract nested data fields. [1]
    - Arrays only have 1 data type (heterogenous) while Structs can have several types (homogenous)

    ```
    # JSON strings `:`
    # Nested Structs `.`
    # Arrays `[]`
    SELECT * FROM events WHERE key:value = "value_of_interest";
    SELECT * FROM events WHERE key.value = "value_of_interest";
    SELECT items[0], items[1] FROM array_sample;
    ```

19. Identify the benefits of using array functions.

- `explode()` - array elements split into several rows
- `size()` - counts the number of elements in array
- `collect_set()` - aggregate function that collects all unique values into an array, including arrays themselves
- `flatten()` - combines an array of arrays into a single array
- `array_distinct()` - select distinct inside an array

20. Parse JSON strings into structs.
    - `schema_of_json()` - returns schema based on sample json string
    - `from_json()` - parses column with json string into a table schema. requires input schema, which can be taken from schema_of_json()

21. Identify which result will be returned based on a join query. [1, 2]
    - review basic joins (left, right, inner, outer) and complex joins (semi, cross, anti)

22. Identify a scenario to use the explode function versus the flatten function
    - explode to add rows, flatten to combine arrays into one in a single row

23. Identify the PIVOT clause as a way to convert data from wide format to a long format.
    - PIVOT uses aggregate function to convert from wide to long and vice versa

24. Define a SQL UDF.

```
# Sample Syntax
CREATE OR REPLACE FUNCTION sale_announcement(item_name STRING, item_price INT)
RETURNS STRING
RETURN concat("The ", item_name, " is on sale for $", round(item_price * 0.8, 0));

SELECT *, sale_announcement(name, price) AS message FROM item_lookup
```

25. Identify the location of a function.

```
# Location is under `Function` key, which shows catalog.database.{function}
DESCRIBE FUNCTION EXTENDED sale_announcement
```

26. Describe the security model for sharing SQL UDFs. [1, 2]
    - Security model follows that of Unity Catalog and Delta Sharing

- SQL SECURITY `DEFINER` option for allowing access to use function via authorization of the function owner.

```
CREATE OR REPLACE FUNCTION
    from_rgb(rgb STRING COMMENT 'an RGB hex color code')
RETURNS TABLE(name STRING COMMENT 'color name')
READS SQL DATA SQL SECURITY DEFINER
COMMENT 'Translates an RGB color code into a color name'
RETURN SELECT name FROM colors WHERE rgb = from_rgb.rgb;
```

27. Use CASE/WHEN in SQL code.
    - IF/ELSE but in SELECT clause

28. Leverage CASE/WHEN for custom control flow.
    - Usable in SELECT clause
    - Can define CASE WHEN statement inside UDF