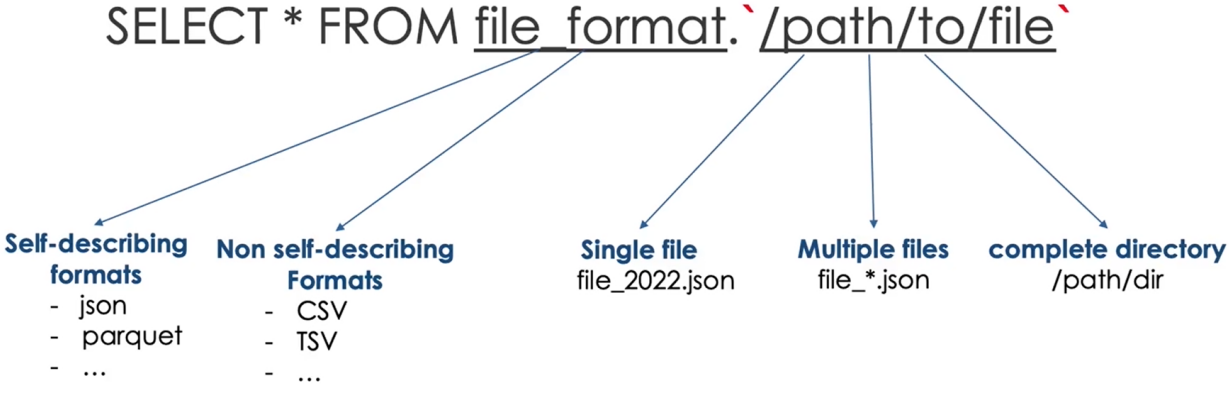
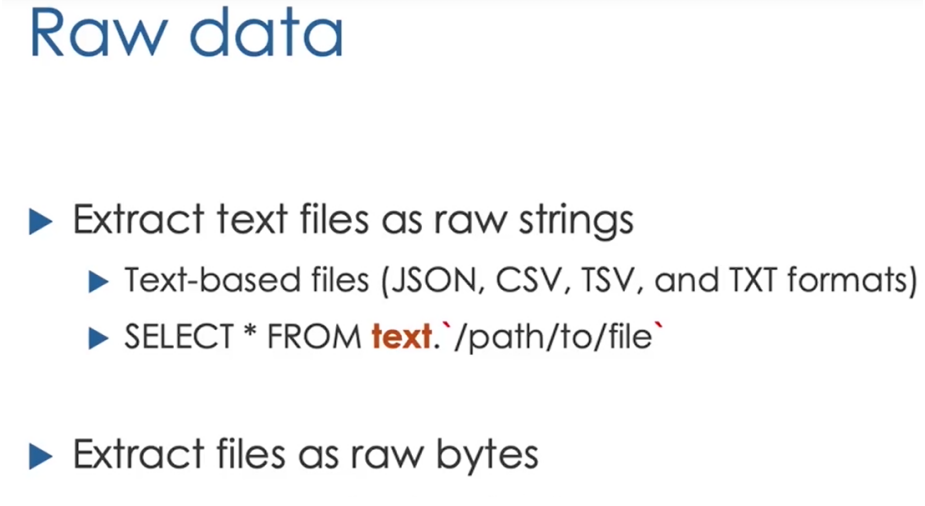
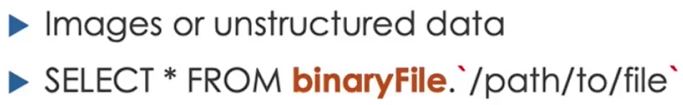
- You can extract data files directly using SQL.
- Use below SQL syntax to extract data directly from different files. If using directory in path, files must follow same format and schema. Use backticks when specifying the location/path of file, no single quotations.



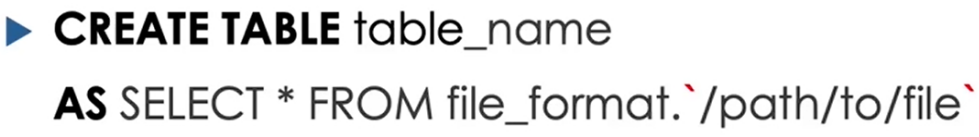
- Example for JSON format querying.

- Raw data extraction.

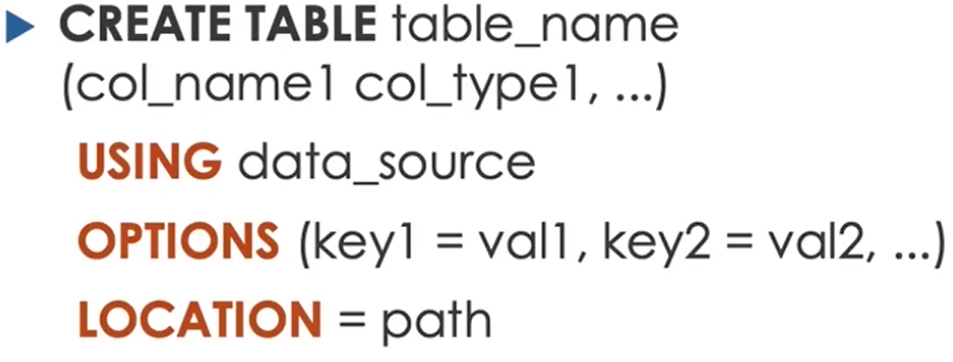




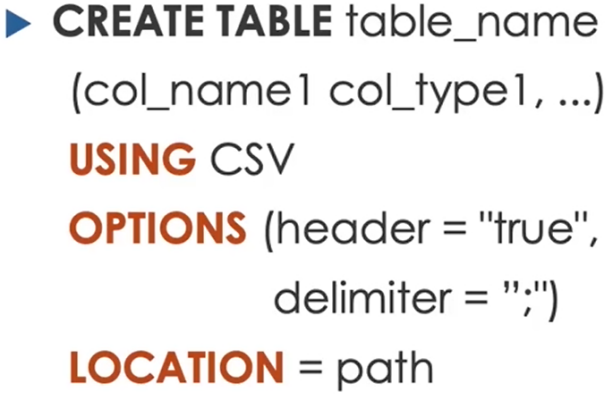
- CTAS: Registering tables from files. Automatically infer schema information from query results, no support for manual schema declaration - Useful for external data digestion with well-defined schema.
- CTAS do not support file options.



- Registering tables on external sources. No data moving during data creation unlike with CTAS statements, we are pointing to files located in an external location.
- Files are kept in original format, which means we are creating a non-Delta and external table here.



- CSV example.



- External Database.

```
(col_name1 col_type1, ...)
USING JDBC
OPTIONS (url = "jdbc:sqlite://hostname:port",
         dbtable = "database.table",
         user = "username",
         password = "pwd" )
```

- Limitations using external location - Not a delta table and we can't expect performance guarantees associated with Delta Lake & Lakehouse.
- To overcome above limitations, create temp view instead of table and create delta table against temp view.

```
CREATE TEMP VIEW temp_view_name (col_name1 col_type1, ...)
USING data_source
OPTIONS (key1 = val1, key2 = val2, ...)
LOCATION = path
```

```
CREATE TABLE table_name
AS SELECT * FROM temp_view_name
```

- **input_file_name()** is built-in Spark SQL command that records source data file for each record. Helpful if troubleshooting problems in the source data become necessary.
- We can refresh table with latest version using below SQL syntax.

  REFRESH TABLE <table name>

- When writing to tables, it's better to overwrite the data in table instead of deleting and recreating tables. We can use time travel to check old versions of the table.
- Overwriting is an atomic operation, concurrent queries can still read the data while you are overwriting the data in table.

  CREATE OR REPLACE TABLE AS SELECT …(CRAS statement)

INSERT OVERWRITE TABLE SELECT (Data in target table will be replaced by data from query, INSERT OVERWRITE will write the data which is matching the schema of target table)

- Appending records into table, **INSERT INTO TABLE SELECT ...** This statement does not have any built-in guarantee to avoid duplicate data in table.
- To resolve duplicate data issue, use **MERGE INTO** statement. MERGE INTO is upsert statement which can update, append or delete the records in single atomic transaction.

```
CREATE OR REPLACE TEMP VIEW customers_updates AS
SELECT * FROM json.`${dataset.bookstore}/customers-json-new`

MERGE INTO customers c
USING customers_updates u
ON c.customer_id = u.customer_id
WHEN MATCHED AND c.email IS NULL AND u.email IS NOT NULL THEN
  UPDATE SET email = u.email, updated = u.updated
WHEN NOT MATCHED THEN INSERT *
```

- Spark SQL has built-in functionality to directly interact with JSON data stored as strings. Use **colon syntax** to traverse nested data structures.

```
SELECT customer_id, profile:first_name, profile:address:country
FROM customers
```

- Spark has ability to parse JSON object into struct types using **from_json()** function - Struct is native spark data type with nested attributes.

```
SQL
CREATE OR REPLACE TEMP VIEW parsed_customers AS
  SELECT customer_id, from_json(profile, schema_of_json('{"first_name":"Thomas","last_name":"Lane","gender":"Male","address":{"street":"06
Boulevard Victor Hugo","city":"Paris","country":"France"}}')) AS profile_struct
  FROM customers;

SELECT * FROM parsed_customers
```

- You can use **dot syntax** to traverse struct data type.

```
SELECT customer_id, profile_struct.first_name, profile_struct.address.country
FROM parsed_customers
```

- Use **star operation** to flatten fields from struct data type into columns.

```
CREATE OR REPLACE TEMP VIEW customers_final AS
  SELECT customer_id, profile_struct.*
  FROM parsed_customers;

SELECT * FROM customers_final
```

- Spark SQL has number of functions to deal with arrays. **Explode function** allow us to put each element of an array on its own row.

```
SELECT order_id, customer_id, explode(books) AS book
FROM orders
```

- **collect_set()** aggregation function allows us to collect unique values for a field including fields within arrays.

```
SELECT customer_id,
  collect_set(order_id) AS orders_set,
  collect_set(books.book_id) AS books_set
FROM orders
GROUP BY customer_id
```

- Use **array_distinct()** and **flatten()** functions on collect_set() to keep only distinct values among all values in array.
- Spark SQL supports all standard join operations.
- Spark SQL supports set operations like **UNION, INTERSECT, MINUS**.
- Spark SQL supports pivot clause - Aggregated values based on specific column values. Useful for dashboarding and apply machine learning algorithms for prediction & inference.

```
CREATE OR REPLACE TABLE transactions AS

SELECT * FROM (
  SELECT
    customer_id,
    book.book_id AS book_id,
    book.quantity AS quantity
  FROM orders_enriched
) PIVOT (
  sum(quantity) FOR book_id in (
    'B01', 'B02', 'B03', 'B04', 'B05', 'B06',
    'B07', 'B08', 'B09', 'B10', 'B11', 'B12'
  )
);

SELECT * FROM transactions
```

- To deal with complex data types (array of struct type), we can use **higher order functions**. These functions allow us to work directly with hierarchical data like arrays and map type objects.
- **FILTER()** is one of the common higher order functions used.

```
SELECT
  order_id,
  books,
  FILTER (books, i -> i.quantity >= 2) AS multiple_copies
FROM orders
```

- If any value don't match filter criteria, it will create an empty array in new return column. Use **WHERE** clause to show only non-empty array values in new return column using sub-query.

```
SELECT order_id, multiple_copies
FROM (
  SELECT
    order_id,
    FILTER (books, i -> i.quantity >= 2) AS multiple_copies
  FROM orders)
WHERE size(multiple_copies) > 0;
```

- **TRANSFORM()** is another higher order function used to transform all items in array and extract transformed value.

```
SELECT
  order_id,
  books,
  TRANSFORM (
    books,
    b -> CAST(b.subtotal * 0.8 AS INT)
  ) AS subtotal_after_discount
FROM orders;
```

- User defined functions (UDF) is custom logic that can be reused in sql queries. It requires function name, optional parameters, return value data type and custom logic. UDFs are permanent objects that are persisted to database. You can use them between different spark sessions and notebooks.

```
CREATE OR REPLACE FUNCTION get_url(email STRING)
```

```
RETURNS STRING

RETURN concat("https://www.", split(email, "@")[1])
```

```
CREATE FUNCTION site_type(email STRING)
RETURNS STRING
RETURN CASE
          WHEN email like "%.com" THEN "Commercial business"
          WHEN email like "%.org" THEN "Non-profits organization"
          WHEN email like "%.edu" THEN "Educational institution"
          ELSE concat("Unknow extenstion for domain: ", split(email, "@")[1])
       END;
```

- **DESCRIBE FUNCTION** provides information about the function input parameters, data types.

```
DESCRIBE FUNCTION get_url
```

- **DESCRIBE FUNCTION EXTENDED** provides all details about function like createdBy, Custom logic used, Owner of function.
- **DROP FUNCTION** removes the function from the database.