The Extract, Load, and Transform Data Pipeline Pattern

In this module, you learn to ...

01 Explain the baseline extract, load, and transform (ELT) architecture diagram.

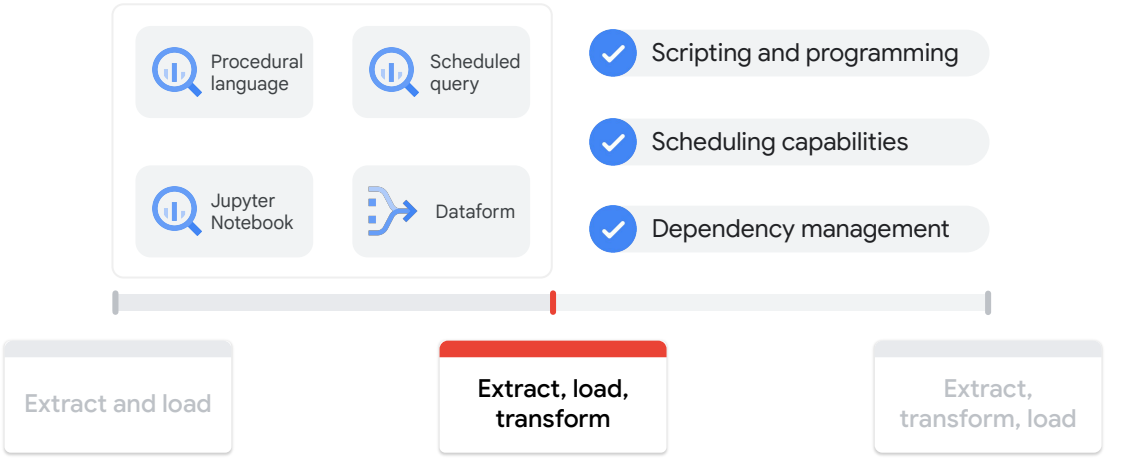
02 Understand a common ELT pipeline on Google Cloud.

03 Learn about BigQuery's SQL scripting and scheduling capabilities.

04 Explain the functionality and use cases for Dataform.

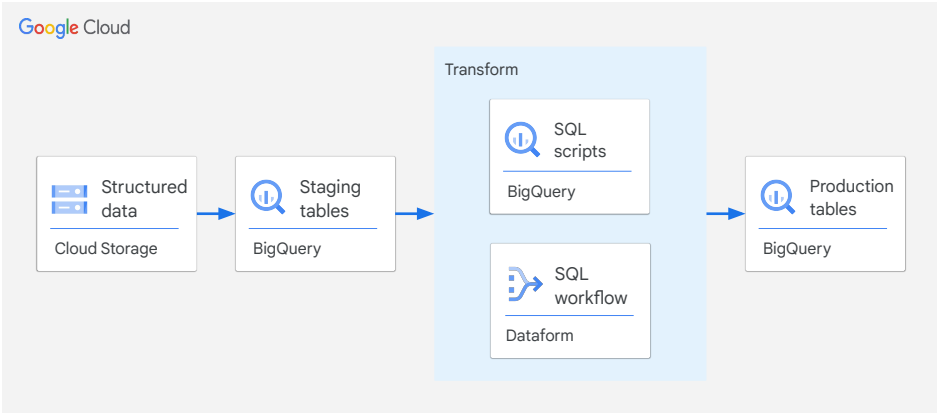
In this module, first, you review the baseline extract, load, and transform architecture diagram. Second, you look at a common ELT pipeline on Google Cloud. Then, you review BigQuery's SQL scripting and scheduling capabilities. Finally, you look at the functionality and use cases for Dataform.

Once data is loaded into BigQuery, there are multiple ways to transform it



Extract, load, and transform centers around data being loaded into BigQuery first. Once data is loaded, there are multiple ways to transform it. Procedural languages like SQL can be used to transform data. Scheduled queries can be used to transform data on a regular basis. Scripting and programming languages like Python can be used to transform data. And a tool like Dataform simplifies transformation beyond basic programming options.

Use an ELT pipeline to apply transformations after data is staged in BigQuery



In an extract, load, and transform pattern pipeline, structured data is first loaded into BigQuery staging tables. Transformations are then applied within BigQuery itself using SQL scripts or tools like Dataform with SQL workflows. The transformed data is finally moved to production tables in BigQuery, ready for use. This approach leverages BigQuery's processing power for efficient data transformation.

BigQuery supports procedural language queries to perform multiple statements in a sequence

```
-- create variables
DECLARE book_name STRING DEFAULT 'Ulysses';
DECLARE book_year INT64 DEFAULT 1922;

-- Create a temporary table called Books.
EXECUTE IMMEDIATE
  "CREATE TEMP TABLE Books
     (title STRING, publish_date INT64)";

-- add a row for Ulysses, using the variables
-- declared and the ? placeholder
EXECUTE IMMEDIATE
  "INSERT INTO Books (title, publish_date) VALUES(?, ?)"
  USING book_name, book_year;
```

Procedural language in BigQuery:

- Runs multiple statements in a sequence with shared state.
- Automates management tasks, such as creating or dropping tables.
- Implements complex logic using programming constructs like `IF` and `WHILE`.
- Can declare user-created variables or reference BigQuery's system variables.
- Can use transactions (`COMMIT`, `ROLLBACK`).

Google Cloud

With support for procedural language, BigQuery allows the execution of multiple SQL statements in sequence with shared state. This enables the automation of tasks like table creation, implementation of complex logic using constructs like `IF` and `WHILE`, and the use of transactions for data integrity. You can also declare variables and reference system variables within your procedural code.

References:

https://cloud.google.com/bigquery/docs/reference/standard-sql/procedural-language
https://cloud.google.com/bigquery/docs/multi-statement-queries

User-defined functions (UDFs) transform data by using SQL or JavaScript

```
# create a permanent user-defined function
CREATE FUNCTION dataset_name.AddFourAndDivide
  (x INT64, y INT64) RETURNS FLOAT64 AS (
    (x + 4) / y
  );
```

```
# use the function in your SQL script
SELECT
  val, dataset_name.AddFourAndDivide(val, 2)
FROM ...;
```

Usage notes:

- Can be persistent (`CREATE FUNCTION`) or temporary (`CREATE TEMPORARY FUNCTION`).
- Use SQL where possible. Use JavaScript if the same cannot be accomplished with SQL.
- JavaScript UDFs can use additional input libraries.
- The `bigquery-utils` project and the `bigquery-public-data.persistent_udfs` public dataset contain community-contributed UDFs.

BigQuery supports user-defined functions or UDFs for custom data transformations using SQL or JavaScript. These UDFs can be persistent or temporary, and it is recommended to use SQL UDFs when possible. JavaScript UDFs offer the flexibility to use external libraries, and community-contributed UDFs are available for reuse.

References:

https://cloud.google.com/bigquery/docs/user-defined-functions

Stored procedures execute a collection of SQL statements

```
# create a stored procedure
CREATE OR REPLACE PROCEDURE
  dataset_name.create_customer()
BEGIN
 DECLARE id STRING;
 SET id = GENERATE_UUID();
 INSERT INTO dataset_name.customers (customer_id)
   VALUES(id);
 SELECT FORMAT("Created customer %s", id);
END
```

```
# call the procedure in your SQL script
CALL dataset_name.create_customer();
```

Usage notes:

- Can be called from queries or other stored procedures.
- Accept input values and return values as output.
- Can access or modify data across multiple datasets.
- Can contain multi-statement queries and support procedural language statements.

Stored procedures are pre-compiled SQL statement collections, streamlining database operations by encapsulating complex logic for enhanced performance and maintainability. Benefits include reusability, parameterization for flexible input, and transaction handling. Stored procedures are called from applications or within SQL scripts, promoting modular design.

References:

https://cloud.google.com/bigquery/docs/procedures

Instructor notes:

- procedural language statements = can define variables and implement control flow (IF statements, WHILE loops)

Run stored procedures for Apache Spark on BigQuery

```
CREATE OR REPLACE PROCEDURE dataset_name.word_count()
WITH CONNECTION `<location>.<connection>`
OPTIONS(engine="SPARK", runtime_version="1.1")
LANGUAGE PYTHON AS R"""

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("spark-bq-wordcount").getOrCreate()

# Load data from BigQuery.
words = spark.read.format("bigquery") \
 .option("table", "bigquery-public-data:samples.shakespeare").load()
words.createOrReplaceTempView("words")

# Perform word count.
word_count = words.select('word','word_count') \
 .groupBy('word').sum('word_count') \
 .withColumnRenamed("sum(word_count)", "sum_word_count")

# [...]
"""
```

Spark procedures on BigQuery:

- Can be defined using the BigQuery PySpark Editor in the UI.
- Can be defined using the `CREATE PROCEDURE` statement:
  - Supports Python, Java, or Scala.
  - Can point to code in a file on Cloud Storage.
  - Code can be defined inline in the BigQuery SQL Editor.

BigQuery supports running stored procedures for Apache Spark. Apache Spark stored procedures on BigQuery can be defined in the BigQuery PySpark editor or using the CREATE PROCEDURE statement with Python, Java, or Scala code. The code can be stored in a Cloud Storage file or defined inline within the BigQuery SQL editor.

References:

https://cloud.google.com/bigquery/docs/connect-to-spark
https://cloud.google.com/bigquery/docs/spark-procedures

Remote functions provide transformations with more complex programming logic

Cloud Run functions in Python:

```python
import functions_framework
import json
import urllib.request

# return the length of an object on Cloud Storage
@functions_framework.http
def object_length(request):
 calls = request.get_json()['calls']
 replies = []
 for call in calls:
   object_content = urllib.request.urlopen(call[0]).read()
   replies.append(len(object_content))
 return json.dumps({'replies': replies}
```

Usage in BigQuery:

```
# create the remote function in BigQuery
CREATE FUNCTION dataset_name.object_length(
  signed_url STRING) RETURNS INT64
REMOTE WITH CONNECTION `<location>.<connection>`
OPTIONS(
  endpoint =
  "https://[...].cloudfunctions.net/object_length",
  max_batching_rows = 1
);
```

- Provides a direct integration to code hosted on Cloud Run functions.
- Call the function the same way as a UDF.

Google Cloud

Remote functions extend BigQuery's capabilities by integrating with Cloud Run functions. This enables complex data transformations using Python code.

You define the remote function in BigQuery, specifying the connection and endpoint to your Cloud Run function. This function can then be called directly within your SQL queries, similar to a UDF, allowing for seamless integration of custom logic.

The example here shows a Python function that gets a list of signed URLs pointing to Cloud Storage objects and returns the length for these objects. In BigQuery, we just need to register the function. We call it `object_length()`. Then, we can use it as is in SQL.

References:

https://cloud.google.com/bigquery/docs/remote-functions
https://cloud.google.com/bigquery/docs/object-table-remote-function

Use Jupyter notebooks for data exploration and transformation

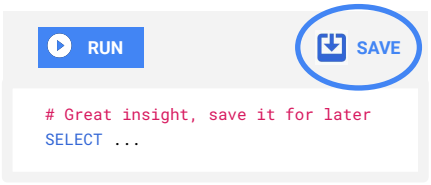
```
# Use BigQuery DataFrames for results not fitting into runtime memory.

import bigframes.pandas as bf
bf.options.bigquery.location = "your-location"
bf.options.bigquery.project = "your-project-id"
df = bf.read_gbq("bigquery-public-data.ml_datasets.penguins")

# Find the heaviest penguin species using the groupby operation to
# calculate the mean body_mass_g:
(
   df["body_mass_g"]
   .groupby(by=df["species"])
   .mean()
   .sort_values(ascending=False)
   .head(10)
)
```

Python notebooks features:

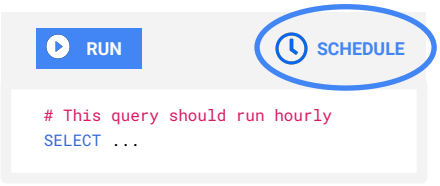
- Join datasets or aggregate data.
- Parse data from complex structures using SQL or Python functions.
- Schedule notebooks to execute at a specified frequency.
- BigQuery DataFrames is integrated into notebooks, no setup required.
- Use `matplotlib`, `seaborn`, and other popular libraries to visualize data.

Jupyter Notebooks coupled with BigQuery DataFrames facilitate efficient data exploration and transformation. This integration emphasizes the ability to handle large datasets that exceed runtime memory, perform complex data manipulations using SQL or Python, and schedule notebook executions. The seamless integration of BigQuery DataFrames and popular visualization libraries further streamlines the entire process.

Save your queries and schedule them for repeating use cases



- Is visible in Explorer > Queries.
- Supports version control.
- Share with users or groups.
- Download as `.sql` file.
- Upload other queries from `.sql` files.



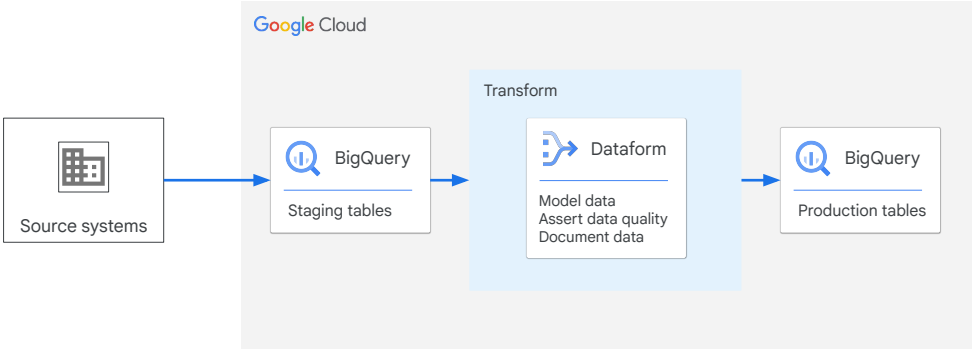
- Set repeat frequency.
- Set start and end times.
- Specify destination for query results.
- Can use runtime parameters.
- **Tip:** use Dataform for SQL workflows.

BigQuery offers the option to save and schedule queries for repeated use. You can save queries, manage versions, and share them with others. Scheduling allows you to automate query execution by setting frequency, start and end times, and result destinations. Dataform is recommended for more complex SQL workflows.

What if you wanted to perform additional operations after query completion?

Often, there are needs to perform additional tasks after a scheduled query is executed in BigQuery. These include tasks such as triggering subsequent SQL scripts, running data quality tests on the output, or configuring security measures. In an ideal situation, these actions can be automated, ensuring data pipelines remain efficient and reliable.

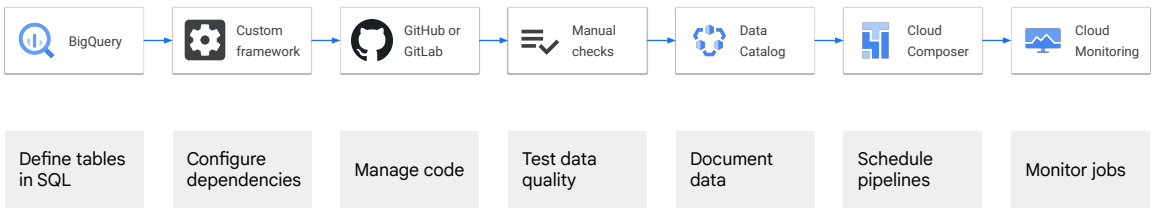
Dataform is a serverless framework to develop and operationalize ELT pipelines in SQL



Dataform is a serverless framework that simplifies the development and management of ELT pipelines using SQL. Dataform enables data transformation within BigQuery, ensuring data quality and providing documentation. This approach streamlines the process of moving data from source systems to production tables in BigQuery, making operations more efficient and manageable.

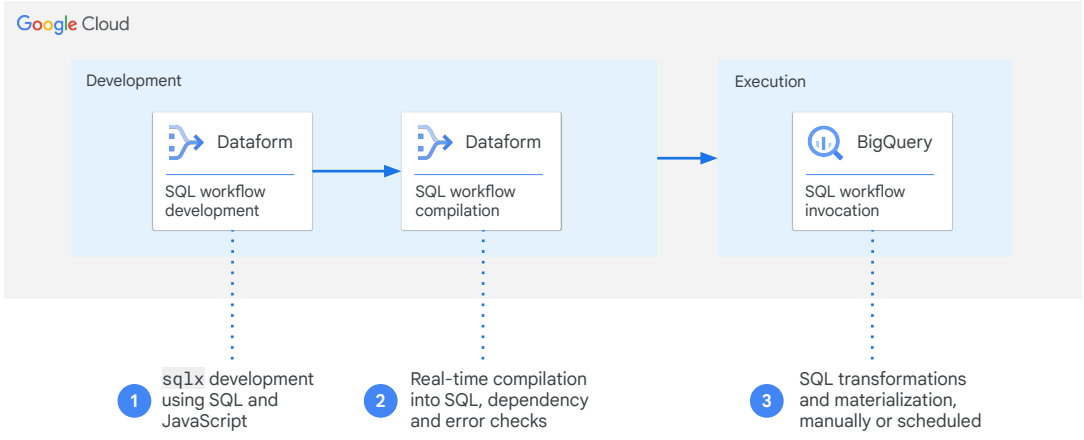
Dataform unifies data transformation, assertion, and automation in BigQuery

Without Dataform, this would be a time-consuming and error-prone process



Dataform streamlines data operations in BigQuery by unifying transformation, assertion, and automation. Without Dataform, tasks like defining tables, managing code, testing data quality, and scheduling pipelines would be time-consuming and prone to errors. They could also involve multiple tools and manual processes. Dataform simplifies these tasks within BigQuery, improving efficiency and data reliability.

SQL workflow development and compilation run in Dataform, workflow execution runs in BigQuery

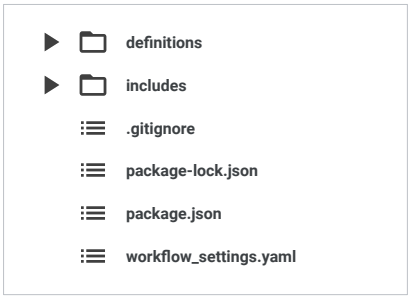


Dataform and BigQuery work together to manage SQL workflows. With Dataform, developers create and compile SQL workflows using SQL and JavaScript. Dataform then performs real-time compilation, including dependency checks and error handling. Finally, the compiled SQL workflows are executed within BigQuery, enabling SQL transformations and materialization either on-demand or through scheduled runs.

References:

- SQL workflow compilation result: https://cloud.google.com/dataform/reference/rest/v1beta1/projects.locations.repositories.compilationResults
- SQL workflow invocation: https://cloud.google.com/dataform/reference/rest/v1beta1/projects.locations.repositories.workflowInvocations

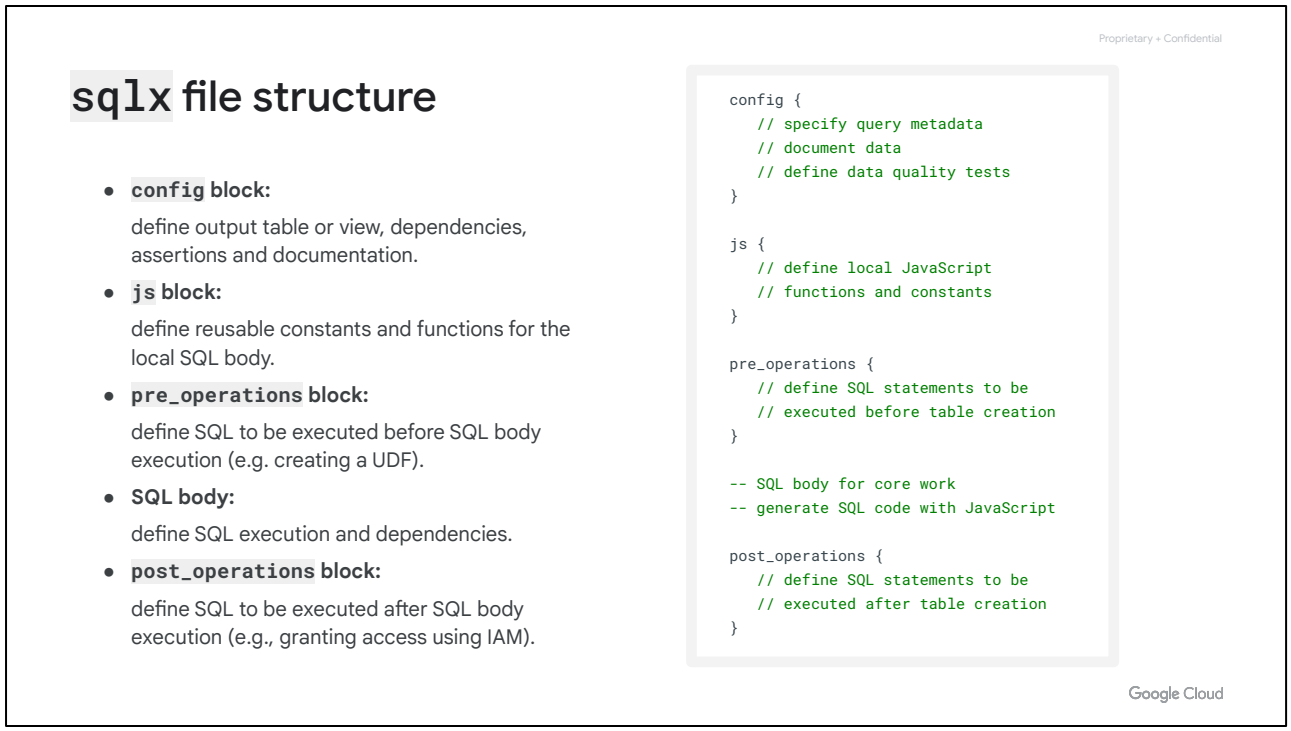
Development happens in workspaces with default files and folders



You can add custom files like `README.md` as well.

- **`definitions` folder:**
  contains `sqlx` files.
- **`includes` folder:**
  contains JavaScript files.
- **`.gitignore` file:**
  lists files and directories to ignore during Git commits.
- **`package.json` file (optional):**
  defines JavaScript packages and required versions.
- **`package-lock.json` file (optional):**
  contains installed packages with exact versions.
- **`workflow_settings.yaml` file:**
  contains basic settings required to compile the project.

Development using Dataform utilizes workspaces containing default files and folders. Key folders include "definitions" for SQLX files and "includes" for JavaScript files. The ".gitignore" file is used for managing Git commits. Developers may also use "package.json" and "package-lock.json" for handling JavaScript dependencies. The "workflow_settings.yaml" file stores project compilation settings, and custom files like "README.md" can also be added.



The sqlx file structure provides a clear framework for organizing SQL code and associated tasks. It begins with a `config` block for metadata and data quality tests, followed by a `js` block to define reusable JavaScript functions. The `pre_operations` block handles SQL statements executed before the main SQL body, which defines the core SQL logic. Finally, the `post_operations` block contains SQL statements to be run after the main execution, ensuring a structured and efficient workflow.

References:

- config block and sql body: https://cloud.google.com/dataform/docs/dataform-core
- pre_operations / post_operations block: https://cloud.google.com/dataform/docs/table-settings
- js block: https://cloud.google.com/dataform/docs/javascript-in-dataform

Instructor notes:

- Each SQLX file contains a query that defines a database relation that Dataform creates and updates inside your data warehouse.
- notice that comments inside a block start with //, whereas comments inside the

- SQL body start with --
    - comments using # don’t work with Dataform, since these are supported in BigQuery only
- SQL body is required, all other blocks are optional
- SQLX file names can only include numbers, letters, hyphens, and underscores: https://cloud.google.com/dataform/docs/declare-source#create_a_sqlx_file_for_data_source_declaration

`sqlx` development takes care of boilerplate code by replacing redundant SQL with definitions

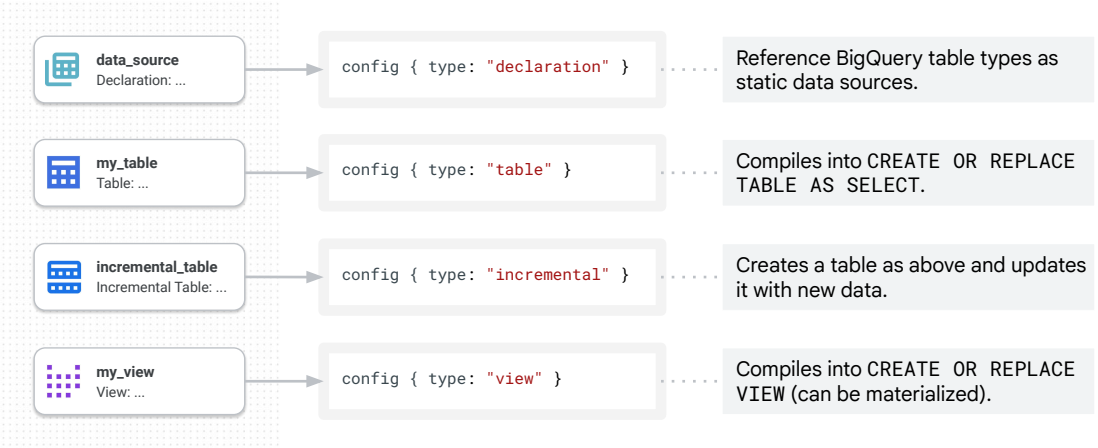
```
CREATE OR REPLACE TABLE
  `projectid.dataset.new_table` AS
SELECT
  country AS country,
  CASE
    WHEN country IN ('US', 'CA') THEN 'NA'
    WHEN country IN ('GB','FR','DE') THEN 'EU'
    WHEN country IN ('AU') THEN country
    ELSE 'Other countries'
  END
  AS country_group,
  device_type AS device_type,
  SUM(revenue) AS revenue,
  SUM(pageviews) AS pageviews
FROM
  `projectid.dataset.table_1`
GROUP BY
  1,2,3
```

```
config { type: "table" }

SELECT
  country AS country,
  ${mapping.region("country")} AS country_group,
  device_type AS device_type,
  SUM(revenue) AS revenue,
  SUM(pageviews) AS pageviews
FROM
  ${ref("table_1")}
GROUP BY
  1,2,3
```


SQLX development streamlines SQL code by replacing repetitive patterns with concise definitions.

The code example demonstrates how a complex `CASE` statement for categorizing countries can be replaced with a simple function call `${mapping.region("country")}`. This approach improves code readability and maintainability by reducing boilerplate code and promoting reusability.

Create table and view definitions that will compile into SQL statements

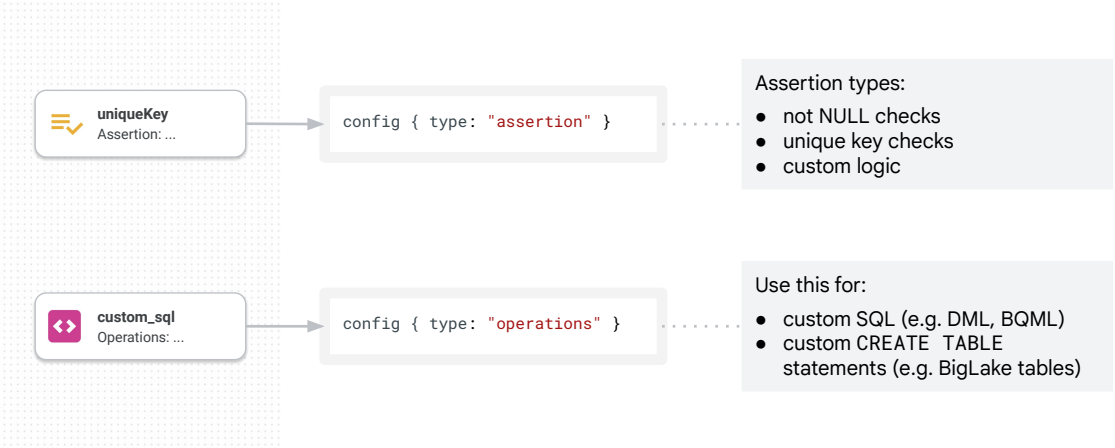


With Dataform, table and view definitions should be created in a specific manner so that they can be compiled into SQL statements.

Key configuration types are:

- `declaration` for referencing existing BigQuery tables,
- `table` for creating or replacing tables with a `SELECT` statement,
- `incremental` for creating tables and updating them with new data, and
- `view` for creating or replacing views, which can optionally be materialized.

Use assertions to define data quality tests and operations to run custom SQL statements



Dataform offers assertions to define data quality tests, ensuring data consistency and accuracy. Assertions can be written in SQL or JavaScript, providing flexibility for complex checks. Operations allow you to run custom SQL statements before, after, or during pipeline execution. These two options enable custom data transformations, data quality checks, and other tasks within your workflows. By combining assertions and operations, Dataform empowers you to create robust and reliable data pipelines in BigQuery.

Dependencies can be configured in two ways

```
config {
  ...
}

SELECT ...
FROM ${ref("customer_details")}
```

Implicit dependency declaration:

- Specify the table/view name in the JavaScript `ref()` function.
- If you want to reference a table without creating a dependency, use `resolve()` instead.

```
config {
  ...
  dependencies: ["customer_details"]
}

SELECT ...
```

Explicit dependency declaration:

- Specify the list of dependencies in the `dependencies` array.
- Dependencies can be multiple instances of data source declarations, tables/views, custom SQL operations, and assertions.

Dataform provides two methods to manage dependencies, implicit declaration and explicit declaration. Implicit declaration is when you reference tables or views directly within your SQL using the `ref()` function. Explicit declaration is when you list dependencies within a `config` block using the `dependencies` array. It is also possible to use the `resolve()` function to reference without creating a dependency.

References:

- reference: https://cloud.google.com/dataform/docs/dependencies
- ref attributes: https://cloud.google.com/dataform/docs/reference/dataform-core-reference#commoncontext
- resolve: https://cloud.google.com/dataform/docs/define-table#reference_other_tables_with_resolve

Instructor notes:

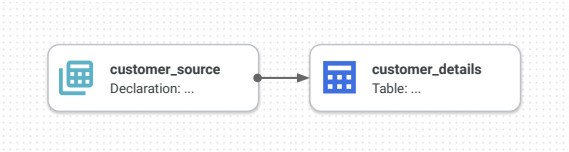
- custom SQL operation needs to have hasOuput : true
- not specifying an explicit dependency, or using ref() (e.g. using directly FROM

- `project.dataset.table`) won’t create a dependency
- You can also specify a different BigQuery project ID, dataset and table name: ${ref("database", "schema", "name")}

Dataform compiles your definitions into SQL scripts and chains them into a workflow

```
config {
  type: "table",
  name: "customer_details",
  description: "Customer details table",
}
SELECT ...
FROM ${ref("customer_source")}
```

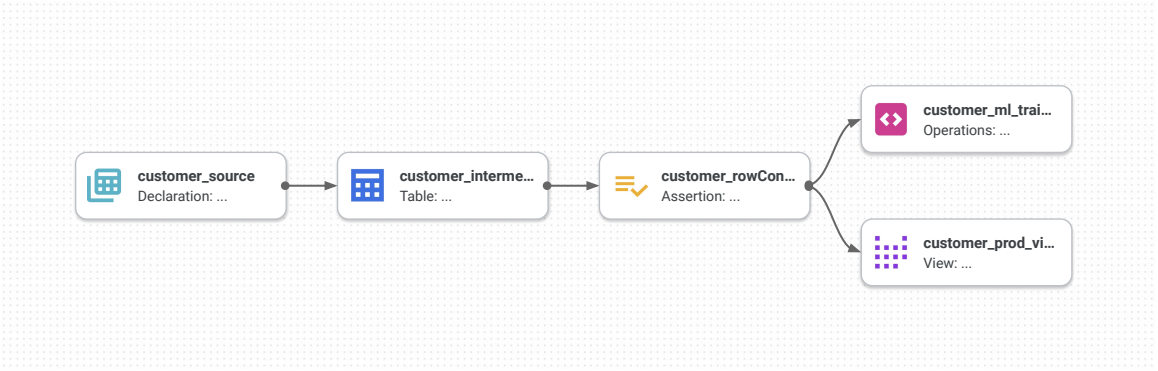
```
CREATE OR REPLACE TABLE
  `<project_id>.<dataset>.customer_details`
OPTIONS(description='''Customer details table''')
AS (
  SELECT ...
  FROM `<project_id>.<dataset>.customer_source`
);
```



Dataform allows you to compile user-defined table definitions into executable SQL scripts. The sample code shows a `customer_details` table being created or replaced based on a `customer_source` table using a `SELECT` statement.

Dataform manages the dependencies between these tables and orchestrates their execution within a workflow. This process streamlines data transformation and ensures efficient data pipeline management.

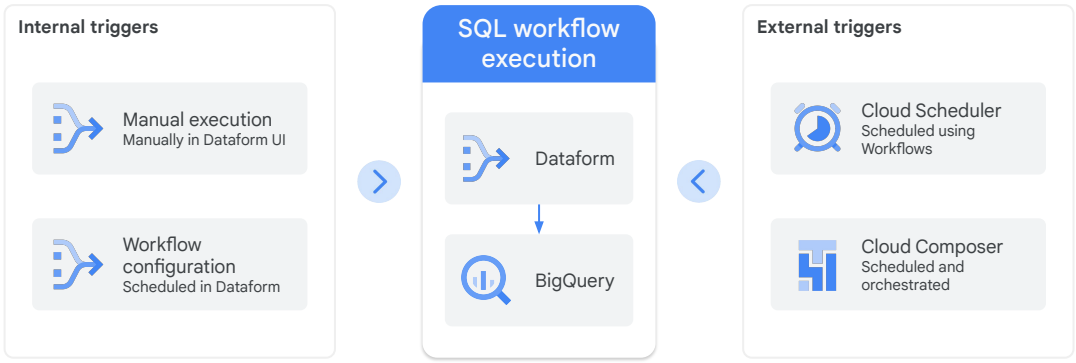
The compiled graph visualizes the SQL workflow with its definitions and dependencies



Dataform SQL workflows are best visualized in graph format. The sample workflow starts with a declaration of `customer_source` followed by a `customer_intermediate` table, likely derived from a source system as a pre-processed data source. Next, `customer_rowConsistency` applies assertions for data quality checks.

The graph then splits into two paths. In one path, an operation named `customer_ml_training` is invoked. It performs operations on the validated data. In the other path, a view named `customer_prod_view` is created.

SQL workflows can be scheduled and executed on a recurring basis



There are several scheduling and execution mechanisms for Dataform SQL workflows. One path is through internal triggers. These include manual execution in the Dataform UI or scheduled configurations within Dataform itself. The other path is through external triggers. These include tools like Cloud Scheduler and Cloud Composer. Ultimately, all workflows are executed within BigQuery, showcasing its central role in this process.

Instructor notes:

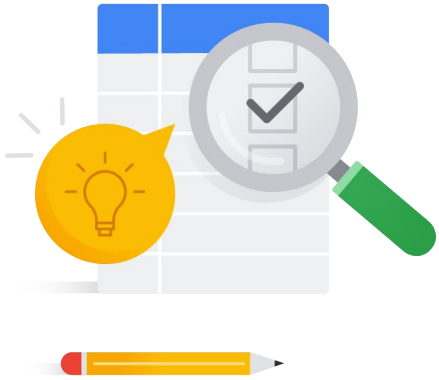
- Remember we can also invoke the Dataform API from any code outside, e.g. Cloud Run functions.
- We will talk in M6 about Cloud Functions, Cloud Scheduler and Cloud Composer.

Lab: Create and Execute a SQL Workflow in Dataform

30 min

Learning objectives

- Create a Dataform repository.
- Create and initialize a Dataform development workspace.
- Create and execute a SQL workflow.
- View execution logs in Dataform.



In this lab, you use Dataform to create and execute a SQL workflow. First, you create a Dataform repository. Second, you create and initialize a Dataform development workspace. Then, you create and execute a SQL workflow. Finally, you view execution logs in Dataform to confirm completion.

Lab URL: https://www.cloudskillsboost.google/catalog_lab/20934


Google Cloud