**Defining External Tables**

External tables enable querying of data stored outside BigQuery without direct ingestion.

config {

type: "operation",

schema: "external\_data",

name: "cloud\_logs",

hasOutput: true

}

CREATE OR REPLACE EXTERNAL TABLE ${self()}

OPTIONS (

format = 'PARQUET',

uris = ['gs://your-bucket/logs/\*.parquet']

);

**Key Config Block Options**

* **type:** Specifies whether the entity is a table, view, or operation.
* **schema:** Defines the schema where the entity is created.
* **name:** Sets the name of the table, view, or operation.
* **tags:** Optional labels to categorize datasets.
* **disabled:** When set to true, prevents execution of the transformation.
* **hasOutput:** Required for operations that generate output tables.

### **Referencing External Data Sources**

To incorporate external datasets, use Dataform's declaration mechanism.

declare {

schema: "external\_sources",

name: "customer\_purchases"

}

Now, the dataset can be referenced like any managed table.

SELECT

customer\_id,

purchase\_amount

FROM

${ref("customer\_purchases")};

#### **Example: Querying External Financial Data**

**Example: Querying External Financial Data**

declare {

schema: "external\_sources",

name: "market\_data"

}

SELECT

stock\_symbol,

closing\_price,

trade\_volume

FROM

${ref("market\_data")}

WHERE

trade\_date = CURRENT\_DATE();

Using external tables maintains efficiency while integrating seamlessly into

**Optimizing Dataform Workflows: Advanced Configurations for Efficiency and Scalability**

Dataform provides powerful features to enhance SQL workflows, ensuring scalability, maintainability, and flexibility. This guide explores how to:

* Execute custom SQL logic before and after transformations
* Dynamically configure queries using environment variables
* Promote code reusability with shared components
* Maintain data integrity through assertions and schema validation

**1. Pre- and Post-Operations: Custom SQL Execution in Dataform**

Dataform allows running SQL commands before and after transformations to handle tasks such as staging data, cleaning up temporary tables, and logging updates.

**Pre-Operations: Preparing Data Before Execution**

Pre-operations help preprocess data before the main transformation runs. This is useful for creating temporary tables, applying filters, or setting session parameters.

**Post-Operations: Managing Data After Execution**

Post-operations execute commands after the transformation completes, often used for logging, auditing, or triggering downstream actions.

**Example: Using Pre/Post Operations for Data Preparation and Logging**

sql

CopyEdit

config {

type: "incremental",

schema: "sales\_reporting",

name: "monthly\_sales\_summary",

uniqueKey: "order\_id"

}

pre\_operations {

CREATE TEMP TABLE temp\_sales AS

SELECT

order\_id,

SUM(amount) AS total\_revenue,

COUNT(\*) AS order\_count

FROM ${ref("raw\_transactions")}

WHERE order\_date >= DATE\_SUB(CURRENT\_DATE(), INTERVAL 30 DAY)

GROUP BY order\_id;

}

SELECT

t.order\_id,

t.total\_revenue,

t.order\_count,

p.category

FROM temp\_sales t

JOIN ${ref("product\_details")} p ON t.order\_id = p.order\_id;

post\_operations {

INSERT INTO admin.audit\_logs (event, table\_name, processed\_at)

VALUES ('Data Refresh', 'monthly\_sales\_summary', CURRENT\_TIMESTAMP());

}

In this example:

* A temporary table is created before the main transformation.
* Sales data is aggregated and enriched with product details.
* A post-operation logs the transformation completion in an audit table.

**2. Dynamic Querying with Environment-Specific Variables**

Managing configurations across multiple environments (e.g., development, testing, production) can be complex. Dataform allows defining environment-specific variables in workflow\_settings.yaml, ensuring seamless transitions between environments.

**Defining Variables in Configuration Files**

Variables help manage database locations, dataset names, and other environment-specific parameters.

**Example: workflow\_settings.yaml Configuration**

yaml

CopyEdit

defaultProject: retail-insights

defaultLocation: US

defaultDataset: reporting

dataformCoreVersion: 3.2.0

vars:

environment: "staging"

storage\_bucket: "data-backup-${dataform.projectConfig.vars.environment}"

reporting\_table: "sales\_data\_${dataform.projectConfig.vars.environment}"

Here, environment-based variables are dynamically set, making the SQL transformations flexible.

**Using Variables in SQLX Files**

sql

CopyEdit

config {

type: "table",

schema: "reporting",

name: dataform.projectConfig.vars.reporting\_table

}

SELECT

order\_id,

amount,

created\_at

FROM `${dataform.projectConfig.vars.defaultProject}.transactions\_${dataform.projectConfig.vars.environment}`;

* The dataset and table names adjust automatically based on the environment.
* Switching from "staging" to "production" requires no manual changes, reducing deployment risks.

**3. Reusability with Shared SQL Logic and Metadata**

For scalable and maintainable workflows, Dataform allows sharing common logic across multiple transformations.

**Using the Includes Folder for Reusable Logic**

Common SQL snippets or column definitions can be stored in the includes directory and referenced across multiple transformations.

**Example: Defining Column Metadata in column\_metadata.js**

javascript

CopyEdit

module.exports = {

revenue: "Total revenue generated from the order.",

order\_date: "Timestamp when the order was placed.",

customer\_id: "Unique identifier for the customer."

};

**Referencing Metadata in SQLX Files**

sql

CopyEdit

config {

type: "table",

schema: "analytics",

name: "customer\_orders",

columns: {

revenue: ${column\_metadata.revenue},

order\_date: ${column\_metadata.order\_date},

customer\_id: ${column\_metadata.customer\_id}

}

}

SELECT

customer\_id,

order\_id,

revenue,

order\_date

FROM ${ref("processed\_orders")};

* The column descriptions remain consistent across all dependent tables.
* Any update in column\_metadata.js is automatically applied across all datasets.

**4. Data Quality Control: Assertions and Schema Validation**

Ensuring data accuracy is critical in any ETL pipeline. Dataform provides assertions and schema validation to detect and prevent data inconsistencies.

**Assertions: Enforcing Business Rules**

Assertions validate that data meets predefined business rules before it is written to the final dataset.

**Example: Validating Positive Transaction Amounts**

sql

CopyEdit

config {

type: "assertion",

schema: "analytics",

name: "validate\_transaction\_amounts"

}

SELECT \*

FROM ${ref("transactions")}

WHERE amount < 0;

If any negative values exist in transactions, Dataform will trigger an error, preventing the transformation from running.

**Schema Validation: Ensuring Consistency**

For external integrations and reporting, schema validation ensures all tables conform to expected column types and structures.

**Optimizing Table Configurations in Dataform: Strategies for Performance and Efficiency**

In large-scale data pipelines, efficient table design is critical for improving query speed, reducing storage costs, and maintaining data accuracy. This guide explores key techniques in Dataform to enhance table management, covering:

* **Partitioning** for efficient data retrieval
* **Clustering** to optimize query execution
* **Incremental processing** for managing continuously growing datasets
* **Advanced table options** to enforce best practices in BigQuery

**1. Structuring Tables for Efficient Querying**

**Partitioning: Breaking Down Large Tables**

Partitioning divides a table into smaller segments based on a chosen column, typically a **date or timestamp**. This ensures queries scan only relevant partitions, significantly improving performance.

💡 **Use Case:** A table storing millions of user events over time can be partitioned by event\_date, allowing queries to focus only on relevant timeframes.

**How to Define Partitioned Tables**

sql

CopyEdit

config {

type: "table",

schema: "user\_engagement",

name: "session\_events",

partitionBy: "event\_date"

}

SELECT

session\_id,

user\_id,

event\_type,

event\_date,

device\_type

FROM ${ref("raw\_event\_logs")};

🔹 **Key Benefits:**  
✔ Limits the data scanned in queries, reducing costs  
✔ Improves query execution time  
✔ Simplifies historical data management

**Clustering: Enhancing Data Organization**

While partitioning groups data by a single column, **clustering** organizes records within partitions based on multiple columns. This further improves performance, especially for queries that filter or group by those columns.

💡 **Use Case:** If queries frequently filter by device\_type and user\_id, clustering on these columns optimizes execution speed.

**Partitioning + Clustering for Faster Queries**

sql

CopyEdit

config {

type: "table",

schema: "user\_engagement",

name: "session\_events",

partitionBy: "event\_date",

clusterBy: ["device\_type", "user\_id"]

}

SELECT

session\_id,

user\_id,

event\_type,

event\_date,

device\_type

FROM ${ref("raw\_event\_logs")};

🔹 **Why Use Clustering?**  
✔ Reduces the need for expensive full-table scans  
✔ Accelerates filtering, aggregations, and joins  
✔ Works seamlessly alongside partitioning

**2. Managing Data Growth with Incremental Processing**

**Incremental Tables: Processing Only New Data**

Instead of reloading entire datasets, **incremental tables** process only new or modified records. This approach is ideal for continuously growing datasets, such as logs or transactions.

💡 **Use Case:** A system that tracks user interactions can incrementally update only the latest events rather than rebuilding the entire table.

**Configuring an Incremental Table**

sql

CopyEdit

config {

type: "incremental",

schema: "user\_engagement",

name: "session\_event\_log",

uniqueKey: "event\_id"

}

incremental {

where: "event\_timestamp > COALESCE((SELECT MAX(event\_timestamp) FROM user\_engagement.session\_event\_log), '1970-01-01')"

}

SELECT

event\_id,

user\_id,

session\_id,

event\_type,

event\_timestamp

FROM ${ref("raw\_event\_logs")};

🔹 **Why Use Incremental Processing?**  
✔ Saves processing time by avoiding unnecessary updates  
✔ Reduces resource consumption and query costs  
✔ Ensures real-time data updates with minimal overhead

**3. Customizing Table Properties with Advanced Options**

**BigQuery Table Settings for Optimization**

Beyond partitioning and clustering, BigQuery allows defining additional table settings to **control data retention, enforce best practices, and improve query performance**.

💡 **Use Case:** A session data table should retain only the last **60 days** of records while ensuring that queries always include a partition filter.

**Applying Advanced Table Settings**

sql

CopyEdit

config {

type: "table",

schema: "user\_engagement",

name: "session\_events",

partitionBy: "event\_date",

additionalOptions: {

"partitionExpirationDays": "60",

"requirePartitionFilter": "true"

}

}

SELECT

session\_id,

user\_id,

event\_type,

event\_date

FROM ${ref("raw\_event\_logs")};

**Creating a Reusable JavaScript Package in Dataform**

Using reusable JavaScript packages in Dataform helps maintain consistency across multiple projects. Instead of copying and pasting the same functions in different repositories, you can create a dedicated package and import it wherever needed.

This guide explains:

* How to create a reusable JavaScript package in Dataform
* How to define and export functions
* How to use the package in SQLX files
* How to reuse JavaScript within a single repository without creating a package

**Why Use a JavaScript Package in Dataform?**

A JavaScript package allows you to centralize logic that multiple teams or projects may need. It helps in:

* Standardizing data transformations
* Reducing duplication across repositories
* Making updates easier by managing functions in a single location

For example, if different teams need to clean customer names or mask email addresses in their workflows, they can use a shared package instead of maintaining separate versions of the same function.

**Steps to Create a JavaScript Package in Dataform**

**1. Set Up a New Repository for the Package**

To create a reusable package, you first need to create a new Dataform repository dedicated to storing JavaScript functions.

* Create a new repository in Dataform.
* Connect it to a Git provider like GitHub or GitLab.
* Set up a basic folder structure.

Example structure:

bash

CopyEdit

dataform-utils/

│── index.js # Main file containing functions and constants

│── helpers/ # Optional folder for additional logic

│── package.json # Optional file for metadata

**2. Define Constants and Functions in index.js**

Inside the repository, create an index.js file. This file will contain all reusable functions and constants.

**Defining Constants**

js

CopyEdit

const DEFAULT\_CATEGORIES = ["electronics", "furniture", "clothing"];

module.exports = { DEFAULT\_CATEGORIES };

This allows all repositories using the package to access predefined category values.

**Defining Functions**

js

CopyEdit

function formatCustomerName(name) {

return `trim(upper(${name}))`;

}

function maskSensitiveEmail(email) {

return `regexp\_replace(${email}, r'([^@]+)', '\*\*\*\*\*')`;

}

module.exports = { formatCustomerName, maskSensitiveEmail };

* formatCustomerName ensures names are always in uppercase without extra spaces.
* maskSensitiveEmail replaces the local part of an email with \*\*\*\*\*, keeping the domain unchanged.

**3. Use the Package in a Dataform Repository**

Once the package is created and stored in a Git repository, it can be used in other Dataform repositories.

**Step 1: Import the Package in Dataform**

* Add the package as a dependency in your Dataform repository.
* Reference it in SQLX files where needed.

**Step 2: Use the Functions in SQLX Files**

sql

CopyEdit

config {

type: "table",

schema: "customer\_data",

name: "processed\_customers"

}

SELECT

customer\_id,

${package.formatCustomerName("customer\_name")} AS formatted\_name,

${package.maskSensitiveEmail("email")} AS masked\_email

FROM ${ref("raw\_customers")};

By using the functions from the package, any updates made to them in the central repository will automatically apply across all projects that use the package.

**Alternative: Reusing JavaScript Functions Within a Single Repository**

If you do not need to share functions across multiple repositories, Dataform provides another way to reuse JavaScript within a single project. You can store functions in the includes folder.

**Steps to Use Includes for JavaScript Functions**

**1. Create a JavaScript File in the Includes Folder**

Inside your Dataform repository, create a file such as includes/utils.js.

js

CopyEdit

function cleanAddress(address) {

return `trim(lower(${address}))`;

}

module.exports = { cleanAddress };

**2. Use the Function in SQLX Files**

sql

CopyEdit

config {

type: "table",

schema: "location\_data",

name: "normalized\_addresses"

}

SELECT

address\_id,

${utils.cleanAddress("street\_address")} AS cleaned\_address

FROM ${ref("raw\_addresses")};

This approach works well if the functions are only needed within a single Dataform project.

**Choosing Between a Package and Includes**

* Use a **JavaScript package** if you need to share logic across multiple Dataform repositories.
* Use the **includes folder** if the functions are only needed within a single repository.

**Summary**

1. A JavaScript package helps maintain reusable logic across multiple repositories.
2. The package is stored in a dedicated Git repository and imported into Dataform projects.
3. Functions and constants are defined in an index.js file and referenced in SQLX files.
4. If functions are only needed in one repository, they can be stored in the includes folder instead of a separate package.

**Implementing Slowly Changing Dimensions (SCD) in Dataform**

**Slowly Changing Dimensions (SCD)** are techniques used in data warehousing to manage and track changes in dimension data over time. Dataform currently provides built-in support specifically for **SCD Type 2 (SCD-2)** implementations through an external JavaScript package.​[Iteration Insights](https://iterationinsights.com/article/how-to-implement-slowly-changing-dimensions-scd-type-2-using-delta-table/?utm_source=chatgpt.com)

**SCD-2 Support in Dataform**

SCD-2 allows for the preservation of historical data by creating a new record in the dimension table whenever a change occurs, rather than overwriting existing records. This approach maintains a complete history of data changes, enabling accurate historical analysis.​[Microsoft Learn+2Google Cloud Community+2Iteration Insights+2](https://www.googlecloudcommunity.com/gc/Databases/Ideas-on-how-to-backfill-a-SCD-Type-2-table-when-older-records/m-p/700821?utm_source=chatgpt.com)[Iteration Insights+2Microsoft Learn+2Google Cloud Community+2](https://learn.microsoft.com/en-us/fabric/data-factory/slowly-changing-dimension-type-two?utm_source=chatgpt.com)

Dataform facilitates SCD-2 implementation via the dataform-scd package, which offers common data models for creating SCD-2 tables from mutable data sources. This package supports multiple data warehouses, including BigQuery, Redshift/PostgreSQL, and Snowflake.​[Microsoft Learn+7GitHub+7GitHub+7](https://github.com/dataform-co/dataform-scd?utm_source=chatgpt.com)

**Utilizing the dataform-scd Package**

To implement SCD-2 in your Dataform project, follow these steps:

1. **Add the Package to Your Project:**

Include the dataform-scd package in your project's package.json dependencies. You can find the latest version on the [releases page](https://github.com/dataform-co/dataform-scd/releases).

Example package.json entry:

json

CopyEdit

{

"dependencies": {

"dataform-scd": "https://github.com/dataform-co/dataform-scd/archive/0.3.tar.gz"

}

}

1. **Configure the SCD Table:**

Create a new JavaScript file in your definitions/ directory and define the SCD table using the scd function from the dataform-scd package.

javascript

CopyEdit

const scd = require("dataform-scd");

scd("source\_data\_scd", {

// Unique identifier for rows in the table.

uniqueKey: "user\_id",

// Timestamp indicating when the row was last changed.

timestamp: "updated\_at",

// Optional: Field storing the hash value of the fields to track changes.

hash: "hash\_value"

});

In this configuration:

* + uniqueKey: Specifies the primary key for identifying unique records.​
  + timestamp: Indicates the column that records the last update time of the row.​[Stack Overflow+9Microsoft Learn+9Google Cloud Community+9](https://learn.microsoft.com/en-us/fabric/data-factory/slowly-changing-dimension-type-two?utm_source=chatgpt.com)
  + hash: (Optional) Provides a mechanism to detect changes by comparing hash values of specified fields.​[Stack Overflow](https://stackoverflow.com/questions/68558270/scd-2-in-data-modelling-how-do-i-detect-changes?utm_source=chatgpt.com)