**Scripts and stored procedures**

A *script* is a collection of SQL statements that you can execute in one request. Scripts can use variables and control-flow statements, and can have side effects. You can use scripts to:

* Run multiple queries in a sequence, with shared state.
* Automate management tasks such as creating or dropping tables.
* Implement more complex logic using programming constructs such as IF and WHILE.

A *procedure* is a persistent script that you can invoke from inside a SQL statement. A procedure can take input arguments and return values as output.

The following example shows a script that sets a variable, runs an INSERT statement, and displays the result as a formatted text string. You can run this script in the same way as a query, for example in the Google Cloud Console or using the bq command-line tool. You can also [save the script](https://cloud.google.com/bigquery/docs/saving-sharing-queries) as a query.

DECLARE id STRING;  
SET id = GENERATE\_UUID();  
  
INSERT INTO mydataset.customers (customer\_id)  
   VALUES(id);  
  
SELECT FORMAT("Created customer ID %s", id);

Here is the same script converted into a procedure:

CREATE OR REPLACE PROCEDURE mydataset.create\_customer()  
BEGIN  
  DECLARE id STRING;  
  SET id = GENERATE\_UUID();  
  INSERT INTO mydataset.customers (customer\_id)  
    VALUES(id);  
  SELECT FORMAT("Created customer %s", id);  
END

In the preceding example, the name of the procedure is mydataset.create\_customer, and the body of procedure appears between [BEGIN](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#begin) and END statements.

To call the procedure, use the [CALL](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#call) statement:

CALL mydataset.create\_customer();

**Writing a script**

A script consists of one or more SQL statements separated by semicolons. Any valid SQL statement can be used in a script. Scripts can also include [scripting statements](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting), which let you use variables or implement control flow with your SQL statements. The following example declares a variable and uses the variable inside an [IF](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#if) statement:

DECLARE day INT64;  
SET day = (SELECT EXTRACT(DAYOFWEEK from CURRENT\_DATE));  
if day = 1 or day = 7 THEN  
  SELECT 'Weekend';  
ELSE  
  SELECT 'Weekday';  
END IF

Scripts are executed in BigQuery using [jobs.insert](https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs/insert), similar to any other query, with the multi-statement script specified as the query text. When a script runs, additional jobs, known as child jobs, are created for each statement in the script. You can enumerate the child jobs of a script by calling [jobs.list](https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs/list), passing in the script’s job ID as the parentJobId parameter.

The [jobs.getQueryResults](https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs/getQueryResults) method returns the query results for the last SELECT, DML, or DDL statement to execute in the script, with no query results if none of the above statements have executed. To get the results of all statements in the script, enumerate the child jobs and call jobs.getQueryResults on each of them.

BigQuery interprets any request with multiple statements as a script, unless the statements consist entirely of CREATE TEMP FUNCTION statements followed by a single query statement. For example, the following is not considered a script:

CREATE TEMP FUNCTION Add(x INT64, y INT64) AS (x + y);  
SELECT Add(3, 4);

**Variables**

Variables must be declared either at the start of the script or at the start of a [BEGIN](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#begin) block. Variables declared at the start of the script are in scope for the entire script. Variables declared inside a BEGIN block have scope for the block. They go out of scope after the corresponding END statement. The maximum size of a variable is 1 MB, and the maximum size of all variables used in a script is 10 MB.

DECLARE x INT64;  
  
BEGIN  
DECLARE y INT64;  
-- Here you can reference x and y  
END;  
  
-- Here you can reference x, but not y

If a variable and column share the same name, the column takes precedence.

This returns column x + column x:

DECLARE x INT64 DEFAULT 0;  
SET x = 10;  
  
WITH Numbers AS (SELECT 50 AS x)  
SELECT (x+x) AS result FROM Numbers;

+--------+  
| result |  
+--------+  
| 100    |  
+--------+

This returns column y + variable x:

DECLARE x INT64 DEFAULT 0;  
SET x = 10;  
  
WITH Numbers AS (SELECT 50 AS y)  
SELECT (y+x) AS result FROM Numbers;

+--------+  
| result |  
+--------+  
| 60     |  
+--------+

**Temporary tables**

Temporary tables let you save intermediate results to a table. These temporary tables exist at the session level, so you don't need to save or maintain them in a dataset. They are automatically deleted some time after the script completes.

The following example creates a temporary table to store the results of a query and uses the temporary table in a subquery:

-- Find the top 100 names from the year 2017.

CREATE TEMP TABLE top\_names(name STRING)

AS

 SELECT name

 FROM `bigquery-public-data`.usa\_names.usa\_1910\_current

 WHERE year = 1910

 ORDER BY number DESC LIMIT 100

;

-- Which names appear as words in Shakespeare's plays?

SELECT

 name AS shakespeare\_name

FROM top\_names

WHERE name IN (

 SELECT name

 FROM `bigquery-public-data`.usa\_names.usa\_1910\_2013

);

Other than the use of TEMP or TEMPORARY, the syntax is identical to the CREATE TABLE syntax.

When you create a temporary table, don't use a project or dataset qualifier in the table name. The table is automatically created in a special dataset.

You can refer to a temporary table by name for the duration of the current script. This includes temporary tables created by a procedure within the script. You cannot share temporary tables, and they are not visible using any of the standard list or other table manipulation methods.

After a script finishes, the temporary table exists for up to 24 hours. It is not saved using the name you gave it, however, but is assigned a random name instead. To view the table structure and data, go to the [BigQuery console](https://console.cloud.google.com/bigquery" \t "console), click **Query history**, and choose the query that created the temporary table. Then, in the **Destination table** row, click **Temporary table**. You are not charged for storing temporary tables.

You can delete a temporary table explicitly before the script completes by using theDROP TABLE statement:

CREATE TEMP TABLE table1(x INT64);  
SELECT \* FROM table1;  -- Succeeds  
DROP TABLE table1;  
SELECT \* FROM table1;  -- Results in an error

When temporary tables are used together with a default dataset, unqualified table names refer to a temporary table if one exists, or a table in the default dataset. The exception is for CREATE TABLE statements, where the target table is considered a temporary table if and only if the TEMP or TEMPORARY keyword is present.

For example, consider the following script:

-- Create table t1 in the default dataset  
CREATE TABLE t1 (x INT64);  
  
-- Create temporary table t1.  
CREATE TEMP TABLE t1 (x INT64);  
  
-- This statement selects from the temporary table.  
SELECT \* FROM t1;  
  
-- Drop the temporary table  
DROP TABLE t1;  
  
-- Now that the temporary table is dropped, this statement selects from the  
-- table in the default dataset.  
SELECT \* FROM t1;

You can explicitly indicate that you are referring to a temporary table by qualifying the table name with \_SESSION:

-- Create a temp table  
CREATE TEMP TABLE t1 (x INT64);  
  
-- Create a temp table using the `\_SESSION` qualifier  
CREATE TEMP TABLE \_SESSION.t2 (x INT64);  
  
-- Select from a temporary table using the `\_SESSION` qualifier  
SELECT \* FROM \_SESSION.t1;

If you use the \_SESSION qualifier for a query of a temporary table that does not exist, the script throws an error indicating the table does not exist. For example, if there is no temporary table named t3, the script throws an error even if a table named t3 exists in the default dataset.

You cannot use \_SESSION to create a non-temporary table:

CREATE TABLE \_SESSION.t4 (x INT64);  -- Fails

**Debugging a script**

Here are some tips for debugging scripts and stored procedures:

* Use the [ASSERT](https://cloud.google.com/bigquery/docs/reference/standard-sql/debugging-statements#assert) statement to assert that a Boolean condition is true.
* Use [BEGIN...EXCEPTION](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#beginexception) to catch errors and display the error message and stack trace.
* Use SELECT FORMAT("....") to show intermediate results.
* When you run a script in the Google Cloud Console, you can view the output of each statement in the script. The bq command-line tool's 'bq query` command also shows the results of each step when you run a script.
* In the Google Cloud Console, you can select an individual statement inside the query editor and run it.

**Writing a procedure**

To create a procedure, use the [CREATE PROCEDURE](https://cloud.google.com/bigquery/docs/reference/standard-sql/data-definition-language#create_procedure) statement. The body of the procedure appears between [BEGIN](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#begin) and END statements:

CREATE PROCEDURE dataset\_name.procedure\_name  
BEGIN  
-- statements here  
END

**Parameters**

A procedure can take a list of named parameters. Each parameter has a data type.

CREATE OR REPLACE PROCEDURE mydataset.create\_customer(name STRING)  
BEGIN  
DECLARE id STRING;  
SET id = GENERATE\_UUID();  
INSERT INTO mydataset.customers (customer\_id, name)  
  VALUES(id, name);  
SELECT FORMAT("Created customer %s (%s)", id, name);  
END

A procedure can have output parameters. An output parameter returns a value from the procedure, but does not allow input for the procedure. To create an output parameter, use the OUT keyword before the name of the parameter.

For example, this version of the procedure returns the new customer ID through the id parameter:

CREATE OR REPLACE PROCEDURE mydataset.create\_customer(name STRING, OUT id STRING)  
BEGIN  
SET id = GENERATE\_UUID();  
INSERT INTO mydataset.customers (customer\_id, name)  
  VALUES(id, name);  
SELECT FORMAT("Created customer %s (%s)", id, name);  
END

To call this procedure, you must use a variable to receive the output value:

--- Create a new customer record.  
DECLARE id STRING;  
CALL mydataset.create\_customer("alice",id);  
  
--- Display the record.  
SELECT \* FROM temp.customers  
WHERE customer\_id = id;

A procedure can also have input/output parameters. An input/output parameter returns a value from the procedure and also accepts input for the procedure. To create an input/output parameter, use the INOUT keyword before the name of the parameter.

# BigQuery Create Procedure - Examples

## Simple Stored Procedure

CREATE OR REPLACE PROCEDURE Dataset.Procedurename()  
BEGIN  
insert into dataset.tablename  
select \* from dataset.table2;  
END

## Stored Procedure with Input Parameter

CREATE OR REPLACE PROCEDURE Dataset.Procedurename(DLY\_DT DATE)  
BEGIN  
insert into dataset.tablename  
select \* from dataset.table2 where File\_date = DLY\_DT;  
END

## Stored Procedure that uses Temp table

CREATE OR REPLACE PROCEDURE Dataset.Procedurename()  
BEGIN  
CREATE TEMP TABLE IF NOT EXISTS TEMP1  
(  
ID INT64  
);

INSERT INTO TEMP1  
SELECT ID FROM EMPLOYEE;

insert into dataset.tablename  
select \* from TEMP1;  
END

## Stored Procedure with variables

CREATE OR REPLACE PROCEDURE Dataset.Procedurename()  
BEGIN  
DECLARE V\_DT DATE;  
SET V\_DT = (SELECT MAX(DT) FROM DATASET.TABLENAME);

insert into dataset.tablename  
select \* from dataset.table2 where File\_date = V\_DT;  
END

# Create View Statement in Google BigQuery

## Create BigQuery View

CREATE OR REPLACE VIEW DATASET.VIEWNAME AS   
SELECT \* FROM DATASET.TABLENAME;

## Create BigQuery View with Joins

CREATE OR REPLACE VIEW DATASET.VIEWNAME AS   
SELECT A.ID,B.NAME FROM DATASET.TABLENAME A  
INNER JOIN DATASET.TABLENAME B ON A.ID = B.ID;

## Create BigQuery Materialized View

Materialized views are precomputed views that periodically cache results of a query for increased performance and efficiency. The Materialized views help improve performance and save cost. Take a look at its [limitations](https://cloud.google.com/bigquery/docs/materialized-views-intro#limitations)

CREATE MATERIALIZED VIEW dataset.viewname  
AS SELECT product\_id, SUM(clicks) AS sum\_clicks  
FROM my\_dataset.my\_base\_table  
GROUP BY 1

**Multi-statement transactions**

BigQuery supports multi-statement transactions inside scripts. A multi-statement transaction lets you perform mutating operations, such as inserting or deleting rows, on one or more tables, and either commit or roll back the changes atomically.

Uses for multi-statement transactions include:

* Performing DML mutations on multiple tables as a single transaction. The tables can span multiple datasets or projects.
* Performing mutations on a single table in several stages, based on intermediate computations.

Transactions guarantee [ACID](https://en.wikipedia.org/wiki/ACID) properties and support snapshot isolation. During a transaction, all reads return a consistent snapshot of the tables referenced in the transaction. If a statement in a transaction modifies a table, the changes are visible to subsequent statements within the same transaction.

**Note:** Reads from external data sources are not guaranteed to be consistent within a transaction if the underlying data source changes during the transaction.

**Transaction scope**

A transaction must be contained in a single SQL script. A script can contain multiple transactions, but they cannot be nested.

To start a transaction, use the [BEGIN TRANSACTION](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#begin_transaction) statement. The transaction ends when any of the following occur:

* The script executes a [COMMIT TRANSACTION](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#commit_transaction) statement. This statement atomically commits all changes made inside the transaction.
* The script executes a [ROLLBACK TRANSACTION](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#rollback_transaction) statement. This statement abandons all changes made inside the transaction.
* The script ends before reaching either of these two statements. In that case, BigQuery automatically rolls back the transaction.

If an error occurs during a transaction and the script has an [exception handler](https://cloud.google.com/bigquery/docs/reference/standard-sql/scripting#beginexceptionend), then BigQuery transfers control to the exception handler. Inside the exception block, can choose whether to commit or roll back the transaction.

If an error occurs during a transaction and there is no exception handler, then the script fails and BigQuery automatically rolls back the transaction.

The following example shows an exception handler that rolls back a transaction:

BEGIN  
  
  BEGIN TRANSACTION;  
  INSERT INTO mydataset.NewArrivals  
    VALUES ('top load washer', 100, 'warehouse #1');  
  -- Trigger an error.  
  SELECT 1/0;  
  COMMIT TRANSACTION;  
  
EXCEPTION WHEN ERROR THEN  
  -- Roll back the transaction inside the exception handler.  
  SELECT @@error.message;  
  ROLLBACK TRANSACTION;  
END;

**Statements supported in transactions**

The following statement types are supported in transactions:

* Query statements: SELECT
* DML statements: INSERT, UPDATE, DELETE, and MERGE
* DDL statements on temporary entities:
  + CREATE TEMP TABLE
  + CREATE TEMP FUNCTION
  + DROP TABLE on a temporary table
  + DROP FUNCTION on a temporary function

DDL statements that create or drop permanent entities, such as datasets, tables, and functions, are not supported inside transactions.

**Date/time functions in transactions**

Within a transaction, the following date/time functions have special behaviors:

* The [CURRENT\_TIMESTAMP](https://cloud.google.com/bigquery/docs/reference/standard-sql/timestamp_functions#current_timestamp), [CURRENT\_DATE](https://cloud.google.com/bigquery/docs/reference/standard-sql/date_functions#current_date), and [CURRENT\_TIME](https://cloud.google.com/bigquery/docs/reference/standard-sql/time_functions#current_time) functions return the timestamp when the transaction started.
* You cannot use the [FOR SYSTEM\_TIME AS OF](https://cloud.google.com/bigquery/docs/reference/standard-sql/query-syntax#for_system_time_as_of) clause to read a table beyond the timestamp when the transaction started. Doing so returns an error.

**Example of a transaction**

This example assumes there are two tables named Inventory and NewArrivals, created as follows:

CREATE OR REPLACE TABLE mydataset.Inventory  
(  
 product string,  
 quantity int64,  
 supply\_constrained bool  
);  
  
CREATE OR REPLACE TABLE mydataset.NewArrivals  
(  
 product string,  
 quantity int64,  
 warehouse string  
);  
  
INSERT mydataset.Inventory (product, quantity)  
VALUES('top load washer', 10),  
     ('front load washer', 20),  
     ('dryer', 30),  
     ('refrigerator', 10),  
     ('microwave', 20),  
     ('dishwasher', 30);  
  
INSERT mydataset.NewArrivals (product, quantity, warehouse)  
VALUES('top load washer', 100, 'warehouse #1'),  
     ('dryer', 200, 'warehouse #2'),  
     ('oven', 300, 'warehouse #1');

The Inventory table contains information about current inventory, and NewArrivals contains information about newly arrived items.

The following transaction updates Inventory with new arrivals and deletes the corresponding records from NewArrivals. Assuming that all statements complete successfully, the changes in both tables are committed atomically as a single transaction.

BEGIN TRANSACTION;  
  
-- Create a temporary table that holds new arrivals from 'warehouse #1'.  
CREATE TEMP TABLE tmp  
  AS SELECT \* FROM mydataset.NewArrivals WHERE warehouse = 'warehouse #1';  
  
-- Delete the matching records from the NewArravals table.  
DELETE mydataset.NewArrivals WHERE warehouse = 'warehouse #1';  
  
-- Merge the records from the temporary table into the Inventory table.  
MERGE mydataset.Inventory AS I  
USING tmp AS T  
ON I.product = T.product  
WHEN NOT MATCHED THEN  
 INSERT(product, quantity, supply\_constrained)  
 VALUES(product, quantity, false)  
WHEN MATCHED THEN  
 UPDATE SET quantity = I.quantity + T.quantity;  
  
-- Drop the temporary table and commit the transaction.  
DROP TABLE tmp;  
  
COMMIT TRANSACTION;

**Transaction concurrency**

If a transaction mutates (update or deletes) rows in a table, then other transactions or DML statements that mutate rows in the same table cannot run concurrently. Conflicting transactions are cancelled. Conflicting DML statements that run outside of a transaction are queued to run later, subject to [queuing limits](https://cloud.google.com/bigquery/docs/reference/standard-sql/data-manipulation-language#update_delete_merge_dml_concurrency).

Operations that read or append new rows can run concurrently with the transaction. For example, any of the following operations can be performed concurrently on a table while a transaction mutates data in the same table:

* SELECT statements
* BigQuery Storage Read API read operations
* Queries from BigQuery BI Engine
* INSERT statements
* Load jobs that use WRITE\_APPEND disposition to append rows
* Streaming writes

If a transaction only reads a table or appends new rows to it, any operation can be performed concurrently on that table.

**Viewing transaction information**

BigQuery assigns a transaction ID to each multi-statement transaction. The transaction ID is attached to each query that executes inside the transaction. To view the transaction IDs for your jobs, query the [INFORMATION\_SCHEMA.JOBS\_BY\_\*](https://cloud.google.com/bigquery/docs/information-schema-jobs) views for the transaction\_id column.

When a script runs, BigQuery creates a child job for each statement in the script. For a given transaction, every child job that is associated with that transaction has the same transaction\_id value.

The following examples show how to find information about your transactions.

**Find all committed or rolled back transactions**

The following query returns all transactions that were successfully committed.

SELECT transaction\_id, parent\_job\_id, query  
FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_PROJECT  
WHERE statement\_type = "COMMIT\_TRANSACTION" AND error\_result IS NULL

The following query returns all transactions that were successfully rolled back.

SELECT  
  transaction\_id, parent\_job\_id, query  
FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_PROJECT  
WHERE statement\_type = "ROLLBACK\_TRANSACTION" AND error\_result IS NULL

**Find the start and end time of a transaction**

The following query returns the starting and ending times for a specified transaction ID.

SELECT transaction\_id, start\_time, end\_time, statement\_type  
FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_USER  
WHERE transaction\_id = "***TRANSACTION\_ID***"  
AND statement\_type IN  
  ("BEGIN\_TRANSACTION", "COMMIT\_TRANSACTION", "ROLLBACK\_TRANSACTION")  
ORDER BY start\_time;

**Find the transaction in which a job is running**

The following query gets the transaction associated with a specified job ID. It returns NULL if the job is not running within a multi-statement transaction.

SELECT transaction\_id  
FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_PROJECT  
WHERE job\_id = '***JOB\_ID***';

**Find the current job running within a transaction**

The following query returns information about the job that is currently running within a specified transaction, if any.

SELECT job\_id, query, start\_time, total\_slot\_ms  
FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_PROJECT  
WHERE transaction\_id = '***TRANSACTION\_ID***' AND state = RUNNING;

**Find the active transactions that affect a table**

The following query returns the active transactions that affect a specified table. For each active transaction, it also returns the ID of the parent job — that is, the job for the script.

WITH running\_transactions AS (  
  SELECT DISTINCT transaction\_id  
    FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_PROJECT  
    EXCEPT DISTINCT  
    select transaction\_id FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_PROJECT  
    WHERE statement\_type = "COMMIT\_TRANSACTION" OR  
    statement\_type = "ROLLBACK\_TRANSACTION"  
)  
  
SELECT  
  transaction\_id, parent\_job\_id, query  
FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_PROJECT AS jobs, running\_transactions  
WHERE destination\_table = ("***PROJECT\_NAME***", "***DATASET\_NAME***", "***TABLE\_NAME***")  
  AND jobs.transaction\_id = running\_transactions.transaction\_id

**Find the active transactions running in a script**

The following query returns the active transactions for a particular job, specified by the ID of the job that is running the script.

SELECT DISTINCT transaction\_id  
  FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_PROJECT WHERE parent\_job\_id = "***JOB\_ID***"  
  EXCEPT DISTINCT  
  select transaction\_id FROM `region-us`.INFORMATION\_SCHEMA.JOBS\_BY\_PROJECT  
  WHERE parent\_job\_id = "***JOB\_ID***" AND  
    (statement\_type = "COMMIT\_TRANSACTION" OR statement\_type = "ROLLBACK\_TRANSACTION")

**Limitations**

* A transaction cannot span multiple scripts.
* Transactions cannot use DDL statements that affect permanent entities.
* Within a transaction, [materialized views](https://cloud.google.com/bigquery/docs/materialized-views-intro) are interpreted as logical views. You can still query a materialized view inside a transaction, but it doesn't result in any performance improvement or cost reduction compared with the equivalent logical view.