Scripting in standard SQL

BigQuery scripting

BigQuery scripting enables you to send multiple statements to BigQuery in one request, to use variables, and to use control flow statements such as `IF` and `WHILE`. For example, you can declare a variable, assign a value to it, and then reference it in a third statement.

In BigQuery, a script is a SQL statement list to be executed in sequence. A SQL statement list is a list of any valid BigQuery statements that are separated by semicolons.

For example:

```
-- Declare a variable to hold names as an array.
DECLARE top_names ARRAY<STRING>;
-- Build an array of the top 100 names from the year 2017.
SET top_names = (
  SELECT ARRAY_AGG(name ORDER BY number DESC LIMIT 100)
  FROM `bigquery-public-data`.usa_names.usa_1910_current
  WHERE year = 2017
);
-- Which names appear as words in Shakespeare's plays?
SELECT
  name AS shakespeare_name
FROM UNNEST(top_names) AS name
WHERE name IN (
  SELECT word
  FROM `bigquery-public-data`.samples.shakespeare
);
```

Scripts are executed in BigQuery using `jobs.insert`, similar to any other query, with the multi-statement script specified as the query text. When a script executes, 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`, passing in the script's job ID as the `parentJobId` parameter.

When `jobs.getQueryResults` is invoked on a script, it will return 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 obtain 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 of `CREATE TEMP FUNCTION` statement(s), with a single final query statement. For example, the following would not be considered a script:

```
CREATE TEMP FUNCTION Add(x INT64, y INT64) AS (x + y);

SELECT Add(3, 4);
```

DECLARE

```
DECLARE variable_name[, ...] [variable_type] [DEFAULT expression];
```

`variable_name` must be a valid identifier, and `variable_type` is any BigQuery type.

Description

Declares a variable of the specified type. If the `DEFAULT` clause is specified, the variable is initialized with the value of the expression; if no `DEFAULT` clause is present, the variable is initialized with the value `NULL`.

If `[variable_type]` is omitted then a `DEFAULT` clause must be specified. The variable's type will be inferred by the type of the expression in the `DEFAULT` clause.

Variable declarations must appear at the start of a script, prior to any other statements, or at the start of a block declared with `BEGIN`. Variable names are case-insensitive.

Multiple variable names can appear in a single `DECLARE` statement, but only one `variable_type` and `expression`.

It is an error to declare a variable with the same name as a variable declared earlier in the current block or in a containing block.

If the `DEFAULT` clause is present, the value of the expression must be coercible to the specified type. The expression may reference other variables declared previously within the same block or a containing block.

The maximum size of a variable is 1 MB, and the maximum size of all variables used in a script is 10 MB.

Examples

The following example initializes the variable `x` as an INT64 with the value `NULL`.

```
DECLARE x INT64;
```

The following example initializes the variable `d` as a DATE with the value of the current date.

```
DECLARE d DATE DEFAULT CURRENT_DATE();
```

The following example initializes the variables `x`, `y`, and `z` as `INT64` with the value 0.

```
DECLARE x, y, z INT64 DEFAULT 0;
```

The following example declares a variable named `item` corresponding to an arbitrary item in the `dataset1.products` table. The type of `item` is inferred from the table schema.

```
DECLARE item DEFAULT (SELECT item FROM dataset1.products LIMIT 1);
```

SET

Syntax

```
SET name = expression;
```

```
SET (variable_name_1, variable_name_2, …, variable_name_n) =
  (expression_1, expression_2, …, expression_n);
```

Description

Sets a variable to have the value of the provided expression, or sets multiple variables at the same time based on the result of multiple expressions.

The `SET` statement may appear anywhere within the body of a script.

Examples

The following example sets the variable `x` to have the value 5.

```
SET x = 5;
```

The following example sets the variable `a` to have the value 4, `b` to have the value 'foo', and the variable `c` to have the value `false`.

```
SET (a, b, c) = (1 + 3, 'foo', false);
```

The following example assigns the result of a query to multiple variables. First, it declares two variables, `target_word` and `corpus_count`; next, it assigns the results of a `SELECT AS STRUCT` query to the two variables. The result of the query is a single row containing a `STRUCT` with two fields; the first element is assigned to the first variable, and the second element is assigned to the second variable.

```
DECLARE target_word STRING DEFAULT 'methinks';
DECLARE corpus_count, word_count INT64;

SET (corpus_count, word_count) = (
  SELECT AS STRUCT COUNT(DISTINCT corpus), SUM(word_count)
  FROM `bigquery-public-data`.samples.shakespeare
  WHERE LOWER(word) = target_word
);

SELECT
```

```
  FORMAT('Found %d occurrences of "%s" across %d Shakespeare works',
         word_count, target_word, corpus_count) AS result;
```

This statement list outputs the following string:

```
Found 151 occurrences of "methinks" across 38 Shakespeare works
```

EXECUTE IMMEDIATE

Syntax

```
EXECUTE IMMEDIATE sql_expression [ INTO variable[, ...] ] [ USING identifier[, ...] ];

sql_expression:
  { "query_statement" | expression("query_statement") }

identifier:
  { variable | value } [ AS alias ]
```

Description

Executes a dynamic SQL statement on the fly.

- `sql_expression`: Represents a query statement, an expression that you can use on a query statement, a single DDL statement, or a single DML statement.
- `expression`: Can be a function, conditional expression, or expression subquery.
- `query_statement`: Represents a valid standalone SQL statement to execute. If this returns a value, the `INTO` clause must contain values of the same type. You may access both system variables and values present in the `USING` clause; all other local variables and query parameters are not exposed to the query statement.
- `INTO` clause: After the SQL expression is executed, you can store the results in one or more variables, using the `INTO` clause.
- `USING` clause: Before you execute your SQL expression, you can pass in one or more identifiers from the `USING` clause into the SQL expression. These identifiers function similarly to query parameters, exposing values to the query statement. An identifier can be a variable or a value.

You can include these placeholders in the `query_statement` for identifiers referenced in the `USING`clause:

- `?`: The value for this placeholder is bound to an identifier in the `USING` clause by index.

  ```
  -- y = 1 * (3 + 2) = 5
  EXECUTE IMMEDIATE "SELECT ? * (? + 2)" INTO y USING 1, 3;
  ```

- `@identifier`: The value for this placeholder is bound to an identifier in the `USING` clause by name. This syntax is identical to the query parameter syntax.

```
-- y = 1 * (3 + 2) = 5
EXECUTE IMMEDIATE "SELECT @a * (@b + 2)" INTO y USING 1 as a, 3 as b;
```

Here are some additional notes about the behavior of the `EXECUTE IMMEDIATE` statement:

- `EXECUTE IMMEDIATE` is restricted from being executed dynamically as a nested element.
- If an `EXECUTE IMMEDIATE` statement returns results, then those results become the result of the entire statement and any appropriate system variables are updated.
- The same variable can appear in both the `INTO` and `USING` clauses.
- `query_statement` can contain a single parsed statement that contains other statements (for example, BEGIN...END)
- If zero rows are returned from `query_statement`, including from zero-row value tables, all variables in the `INTO` clause are set to NULL.
- If one row is returned from `query_statement`, including from zero-row value tables, values are assigned by position, not variable name.
- If an `INTO` clause is present, an error is thrown if you attempt to return more than one row from `query_statement`.

Examples

In this example, we create a table of books and populate it with data. Note the different ways that you can reference variables, save values to variables, and use expressions.

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

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

-- Add a row for Hamlet (less secure)
EXECUTE IMMEDIATE
  "INSERT INTO Books (title, publish_date) VALUES('Hamlet', 1599)";

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

-- add a row for Emma, using the identifier placeholder
EXECUTE IMMEDIATE
  "INSERT INTO Books (title, publish_date) VALUES(@name, @year)"
  USING 1815 as year, "Emma" as name;

-- add a row for Middlemarch, using an expression
EXECUTE IMMEDIATE
  CONCAT("INSERT INTO Books (title, publish_date)", "VALUES('Middlemarch', 1871)");

-- save the publish date of the first book, Hamlet, to a variable called first_date
EXECUTE IMMEDIATE "SELECT publish_date FROM Books LIMIT 1" INTO first_date;

+------------------+------------------+
| title            | publish_date     |
+------------------+------------------+
| Hamlet           | 1599             |
| Ulysses          | 1922             |
| Emma             | 1815             |
| Middlemarch      | 1871             |
+------------------+------------------+
```

BEGIN

Syntax

```
BEGIN
  sql_statement_list
END;
```

Description

`BEGIN` initiates a block of statements where declared variables exist only until the corresponding `END`. `sql_statement_list` is a list of zero or more SQL statements ending with semicolons.

Variable declarations must appear at the start of the block, prior to other types of statements. Variables declared inside a block may only be referenced within that block and in any nested blocks. It is an error to declare a variable with the same name as a variable declared in the same block or an outer block.

There is a maximum nesting level of 50 for blocks and conditional statements such as `BEGIN`/`END`, `IF`/`ELSE`/`END IF`, and `WHILE`/`END WHILE`.

`BEGIN`/`END` is restricted from being executed dynamically as a nested element.

Examples

The following example declares a variable `x` with the default value 10; then, it initiates a block, in which a variable `y` is assigned the value of `x`, which is 10, and returns this value; next, the `END`statement terminates the block, ending the scope of variable `y`; finally, it returns the value of `x`.

```
DECLARE x INT64 DEFAULT 10;
BEGIN
  DECLARE y INT64;
  SET y = x;
  SELECT y;
END;
SELECT x;
```

BEGIN...EXCEPTION

Syntax

```
BEGIN
  sql_statement_list
EXCEPTION WHEN ERROR THEN
```

```
  sql_statement_list
END;
```

Description

`BEGIN...EXCEPTION` executes a block of statements. If any of the statements encounter an error, the script skips the remainder of the block and executes the statements in the `EXCEPTION` clause.

Within the `EXCEPTION` clause, you can access details about the error using the following `EXCEPTION`system variables:

Name	Type	Description
`@@error.formatted_stack_trace`	`STRING`	The content of `@@error.stack_trace` expressed as a human readable string. This value is intended for display purposes, and is subject to change without notice. Programmatic access to an error's stack trace should use `@@error.stack_trace`instead.
`@@error.message`	`STRING`	Specifies a human-readable error message.
`@@error.stack_trace`	See 1.	Each element of the array corresponds to a statement or procedure call executing at the time of the error, with the currently executing stack frame appearing first. The meaning of each field is defined as follows: • line/column: Specifies the line and column number of the stack frame, starting with 1. If the frame occurs within a procedure body, then `line 1 column 1` corresponds to the `BEGIN` keyword at the start of the procedure body. • location: If the frame occurs within a procedure body, specifies the full name of the procedure, in the form `[project_name].[dataset_name].[procedure_name]`. If the frame refers to a location in a top-level script, this field is `NULL`. • filename: Reserved for future use. Always `NULL`.
`@@error.statement_text`	`STRING`	Specifies the text of the statement which caused the error.

[1] The type for `@@error.stack_trace` is `ARRAY<STRUCT<line INT64, column INT64, filename STRING, location STRING>>`.

As BigQuery reserves the right to revise error messages at any time, consumers of `@@error.message`should not rely on error messages remaining the same or following any particular pattern. Do not obtain error location information by extracting text out of the error message — use`@@error.stack_trace` and `@@error.statement_text` instead.

To handle exceptions that are thrown (and not handled) by an exception handler itself, you must wrap the block in an outer block with a separate exception handler.

The following shows how to use an outer block with a separate exception handler:

```
BEGIN
  BEGIN
    ...
  EXCEPTION WHEN ERROR THEN
    SELECT 1/0;
  END;
EXCEPTION WHEN ERROR THEN
```

```
  -- The exception thrown from the inner exception handler lands here.
END;
```

`BEGIN...EXCEPTION` blocks also support `DECLARE` statements, just like any other `BEGIN` block. Variables declared in a `BEGIN` block are valid only in the `BEGIN` section, and may not be used in the block's exception handler.

Examples

```
CREATE OR REPLACE PROCEDURE dataset1.proc1() BEGIN
  SELECT 1/0;
END;

CREATE OR REPLACE PROCEDURE dataset1.proc2() BEGIN
  CALL dataset1.proc1();
END;

BEGIN
  CALL dataset1.proc2();
EXCEPTION WHEN ERROR THEN
  SELECT
    @@error.message,
    @@error.stack_trace,
    @@error.statement_text,
    @@error.formatted_stack_trace;
END;
```

In this example, when the division by zero error occurs, instead of terminating the entire script, BigQuery will terminate `dataset1.proc1()` and `dataset1.proc2()` and execute the SELECT statement in the exception handler. When the exception handler runs, the variables will have the following values:

Variable	Value
`@@error.message`	`"Query error: division by zero: 1 / 0 at <project>.dataset1.proc1:2:3]"`
`@@error.stack_trace`	`[ STRUCT(2 AS line, 3 AS column, NULL AS filename, "<project>.dataset1.proc1:2:3" AS location), STRUCT(2 AS line, 3 AS column, NULL AS filename, "<project>.dataset1.proc2:2:3" AS location), STRUCT(10 AS line, 3 AS column, NULL AS filename, NULL AS location), ]`
`@@error.statement_text`	`"SELECT 1/0"`
`@@error.formatted_stack_trace`	`"At <project>.dataset1.proc1[2:3]\nAt <project>.dataset1.proc2[2:3]\nAt [10:3]"`

END

Terminates a block initiated by `BEGIN`. `BEGIN/END` is restricted from being executed dynamically as a nested element.

IF

Syntax

```
IF condition THEN [sql_statement_list]
[ELSEIF condition THEN sql_statement_list]
[ELSEIF condition THEN sql_statement_list]...
[ELSE  sql_statement_list]
END IF;
```

Description

Executes the first `sql_statement_list` where the condition is true, or the optional `ELSEsql_statement_list` if no conditions match.

There is a maximum nesting level of 50 for blocks and conditional statements such as `BEGIN/END`, `IF/ELSE/END IF`, and `WHILE/END WHILE`.

`IF` is restricted from being executed dynamically as a nested element.

Examples

The following example declares a INT64 variable `target_product_id` with a default value of 103; then, it checks whether the table `dataset.products` contains a row with the `product_id` column matches the value of `target_product_id`; if so, it outputs a string stating that the product has been found, along with the value of `default_product_id`; if not, it outputs a string stating that the product has not been found, also with the value of `default_product_id`.

```
DECLARE target_product_id INT64 DEFAULT 103;
IF EXISTS (SELECT 1 FROM dataset.products
           WHERE product_id = target_product_id) THEN
  SELECT CONCAT('found product ', CAST(target_product_id AS STRING));
  ELSEIF EXISTS (SELECT 1 FROM dataset.more_products
           WHERE product_id = target_product_id) THEN
  SELECT CONCAT('found product from more_products table',
  CAST(target_product_id AS STRING));
ELSE
  SELECT CONCAT('did not find product ', CAST(target_product_id AS STRING));
END IF;
```

Loops

LOOP

Syntax

```
LOOP
  sql_statement_list
END LOOP;
```

Description

Executes `sql_statement_list` until a `BREAK` or `LEAVE` statement exits the loop. `sql_statement_list` is a list of zero or more SQL statements ending with semicolons. `LOOP` is restricted from being executed dynamically as a nested element.

Examples

The following example declares a variable `x` with the default value 0; then, it uses the `LOOP` statement to create a loop that executes until the variable `x` is greater than or equal to 10; after the loop terminates, the example outputs the value of `x`.

```
DECLARE x INT64 DEFAULT 0;
LOOP
  SET x = x + 1;
  IF x >= 10 THEN
    LEAVE;
  END IF;
END LOOP;
SELECT x;
```

This example outputs the following:

```
+----+
| x  |
+----+
| 10 |
+----+
```

WHILE

Syntax

```
WHILE boolean_expression DO
  sql_statement_list
END WHILE;
```

There is a maximum nesting level of 50 for blocks and conditional statements such as `BEGIN`/`END`, `IF`/`ELSE`/`END IF`, and `WHILE`/`END WHILE`.

Description

While `boolean_expression` is true, executes `sql_statement_list`. `boolean_expression` is evaluated for each iteration of the loop. `WHILE` is restricted from being executed dynamically as a nested element.

BREAK

Description

Exit the current loop.

It is an error to use `BREAK` outside of a loop.

Examples

The following example declares two variables, `heads` and `heads_count`; next, it initiates a loop, which assigns a random boolean value to `heads`, then checks to see whether `heads` is true; if so, it outputs "Heads!" and increments `heads_count`; if not, it outputs "Tails!" and exits the loop; finally, it outputs a string stating how many times the "coin flip" resulted in "heads."

```
DECLARE heads BOOL;
DECLARE heads_count INT64 DEFAULT 0;
LOOP
  SET heads = RAND() < 0.5;
  IF heads THEN
    SELECT 'Heads!';
    SET heads_count = heads_count + 1;
  ELSE
    SELECT 'Tails!';
    BREAK;
  END IF;
END LOOP;
SELECT CONCAT(CAST(heads_count AS STRING), ' heads in a row');
```

LEAVE

Synonym for `BREAK`.

CONTINUE

Description

Skip any following statements in the current loop and return to the start of the loop.

It is an error to use `CONTINUE` outside of a loop.

Examples

The following example declares two variables, `heads` and `heads_count`; next, it initiates a loop, which assigns a random boolean value to `heads`, then checks to see whether `heads` is true; if so, it outputs "Heads!", increments `heads_count`, and restarts the loop, skipping any remaining statements; if not, it outputs "Tails!" and exits the loop; finally, it outputs a string stating how many times the "coin flip" resulted in "heads."

```
DECLARE heads BOOL;
DECLARE heads_count INT64 DEFAULT 0;
LOOP
  SET heads = RAND() < 0.5;
```

```
  IF heads THEN
    SELECT 'Heads!';
    SET heads_count = heads_count + 1;
    CONTINUE;
  END IF;
  SELECT 'Tails!';
  BREAK;
END LOOP;
SELECT CONCAT(CAST(heads_count AS STRING), ' heads in a row');
```

ITERATE

Synonym for `CONTINUE`.

RAISE

Syntax

```
RAISE [USING MESSAGE = message];
```

Description

Raises an error, optionally using the specified error message when `USING MESSAGE = message` is supplied.

When `USING MESSAGE` is not supplied

The `RAISE` statement must only be used within an `EXCEPTION` clause. The `RAISE` statement will re-raise the exception that was caught, and preserve the original stack trace.

When `USING MESSAGE` is supplied

If the `RAISE` statement is contained within the `BEGIN` section of a `BEGIN...EXCEPTION` block:

- The handler will be invoked.
- The value of `@@error.message` will exactly match the `message` string supplied (which may be `NULL` if `message` is `NULL`).
- The stack trace will be set to the `RAISE` statement.

If the `RAISE` statement is not contained within the `BEGIN` section of a `BEGIN...EXCEPTION` block, the `RAISE` statement will terminate execution of the script with the error message supplied.

RETURN

In a BigQuery script, `RETURN` terminates execution of the current script.

CALL

Syntax

```
CALL procedure_name (procedure_argument[, …])
```

Description

Calls a procedure with an argument list. `procedure_argument` may be a variable or an expression. For `OUT` or `INOUT` arguments, a variable passed as an argument must have the proper BigQuery type.

The same variable may not appear multiple times as an `OUT` or `INOUT` argument in the procedure's argument list.

The maximum depth of procedure calls is 50 frames.

`CALL` is restricted from being executed dynamically as a nested element.

Examples

The following example declares a variable `retCode`. Then, it calls the procedure `updateSomeTables`in the dataset `myDataset`, passing the arguments `'someAccountId'` and `retCode`. Finally, it returns the value of `retCode`.

```
DECLARE retCode INT64;
-- Procedure signature: (IN account_id STRING, OUT retCode INT64)
CALL myDataset.UpdateSomeTables('someAccountId', retCode);
SELECT retCode;
```

System Variables

You can use system variables to check information during execution of a script.

Name	Type	Description
`@@current_job_id`	`STRING`	Job ID of the currently executing job. In the context of a script, this returns the job responsible for the current statement, not the entire script.
`@@last_job_id`	`STRING`	Job ID of the most recent job to execute in the current script, not including the current one. If the script contains `CALL` statements, this job may have originated in a different procedure.
`@@project_id`	`STRING`	ID of the project used to execute the current query. In the context of a procedure, `@@project_id` refers to the project that is running the script, not the project which owns the procedure.
`@@row_count`	`INT64`	If used in a script and the previous script statement is DML, specifies the number of rows modified, inserted, or deleted, as a result of that DML statement. If the previous statement is a MERGE statement, `@@row_count` represents the combined total number of rows inserted, removed, and deleted. This value is `NULL` if not in a script.

Name	Type	Description
`@@script.bytes_billed`	`INT64`	Total bytes billed so far in the currently executing script job. This value is `NULL` if not in a script.
`@@script.bytes_processed`	`INT64`	Total bytes processed so far in the currently executing script job. This value is `NULL` if not in a script.
`@@script.creation_time`	`TIMESTAMP`	Creation time of the currently executing script job. This value is `NULL` if not in a script.
`@@script.job_id`	`STRING`	Job ID of the currently executing script job. This value is `NULL` if not in a script.
`@@script.num_child_jobs`	`INT64`	Number of currently completed child jobs. This value is `NULL` if not in a script.
`@@script.slot_ms`	`INT64`	Number of slot milliseconds used so far by the script. This value is `NULL` if not in a script.
`@@time_zone`	`STRING`	The default time zone to use in time zone-dependent SQL functions, when an explicit time zone is not specified as an argument. Unlike other system variables, `@@time_zone` can be modified by using a SET statement to any valid time zone name. At the start of each script, `@@time_zone` begins as “UTC”.

Note: For backward compatibility, expressions used in an **`OPTIONS`** or **`FOR SYSTEM TIME AS OF`** clause default to the **`America/Los_Angeles`** time zone, while all other date/time expressions default to the **`UTC`** time zone. If **`@@time_zone`** has been set earlier in the script, the chosen time zone will apply to all date/time expressions, including **`OPTIONS`** and **`FOR SYSTEM TIME AS OF`** clauses.

In addition to the system variables shown above, you can use `EXCEPTION` system variables during execution of a script. `EXCEPTION` system variables are `NULL` if not used in an exception handler. The following are `EXCEPTION` system variables.

- `@@error.formatted_stack_trace`
- `@@error.message`
- `@@error.stack_trace`
- `@@error.statement_text`

For more information about the `EXCEPTION` system variables, see BEGIN...EXCEPTION.

Permissions

Permission to access a table, model, or other resource is checked at the time of execution. If a statement is not executed, or an expression is not evaluated, BigQuery does not check whether the user executing the script has access to any resources referenced by it.

Within a script, the permissions for each expression or statement are validated separately. For example, consider the following script:

```
SELECT * FROM dataset_with_access.table1;
SELECT * FROM dataset_without_access.table2;
```

If the user executing the script has access to `table1` but does not have access to `table2`, the first query will succeed and the second query will fail. The script job itself will also fail.

Security constraints

Dynamic SQL is convenient but can offer new opportunities for misuse. For example, executing the following query poses a potential security threat since the table parameter could be improperly filtered, allow access to, and be executed on unintended tables.

```
EXECUTE IMMEDIATE CONCAT('SELECT * FROM ', @employee_table);
```

To avoid exposing or leaking sensitive data in a table or running commands like `DROP TABLE` to delete data in a table, BigQuery's dynamic SQL scripting has support for multiple security measures to reduce exposure to SQL injection attacks, including:

- You cannot execute multiple SQL statements embedded in the parameters passed into dynamic SQL.
- The following commands are restricted from being executed dynamically: `BEGIN`/`END`, `CALL`, `IF`, `LOOP`, `WHILE`, and `EXECUTE IMMEDIATE`.

Configuration field limitations

The following query configuration fields cannot be set for scripting:

- `clustering`
- `create_disposition`
- `destination_table`
- `destination_encryption_configuration`
- `range_partitioning`
- `schema_update_options`
- `time_partitioning`
- `user_defined_function_resources`
- `write_disposition`