Handling schema changes to BigQuery export data tables

This page addresses how to handle the schema changes made on **October 28, 2020** to the Cloud Billing data that is exported to tables in BigQuery.

Understanding the changes

The table schema for the Cloud Billing standard usage cost data exported to BigQuery has been updated to provide more clarity with additional data fields. This table is named `gcp_billing_export_v1_<BILLING_ACCOUNT_ID>` in the BigQuery dataset.

The following data fields have been added to the Cloud Billing BigQuery usage export schema:

- `project.number`
- `adjustment_info`
- `adjustment_info.id`
- `adjustment_info.mode`
- `adjustment_info.description`
- `adjustment_info.type`

This data is new as of October 29, 2020, and is not available for data usage recorded before then. Please update your integrations or automations based on the new schema by performing migrations, if and when necessary. For information on the data these new fields provide, refer to Understanding Cloud Billing data tables in BigQuery. (/billing/docs/how-to/export-data-bigquery-tables#standard-usage-cost-data-schema).

Impact to existing tables and queries

Because the table structure for the standard usage cost data export changed, any queries that *directly reference* the exported tables no longer provide you with all of the available data. To resolve this, we recommend creating BigQuery views (/bigquery/docs/views) that query the exported tables and present the information in your preferred structure. You can then adjust the queries that feed your reports and dashboards to pull from the views, instead of the exported tables.

By using views, you can standardize the structure of the data used in your queries and dashboards.

The views you create should normalize the data so that all of the relevant tables present the same schema to your queries. This protects you from future schema changes, allowing you to modify the view's underlying query in those instances when the data schema changes.

Creating views to deal with schema changes

If you need to retain tables that use the previous schema, we recommend creating BigQuery views for those tables to normalize the data schema. When creating a view to migrate from the previous schema to the new one, you can use a `UNION` statement (/bigquery/docs/reference/standard-sql/query-syntax#union) to combine tables with mismatching schemas. The view you create will depend on the data fields you use in your queries and dashboards.

One or more of the following examples might apply to your situation, where your queries might or might not use the new fields `project.number` and `adjustment_info`.

1. You use tables that include pre-existing schema properties and new ones, such as `credits.type`, `credits.id`, `credits.full`, `project.number`, and `adjustment_info`. For an example of how to create this view, see Creating a view for tables with all fields in the updated schema (#create_view_all_fields).
2. You use tables that do *not* include the pre-existing schema properties `credits.type`, `credits.id`, and `credits.full`. For an example of how to create this view, see Creating a view for tables without `credits.type`, `credits.id`, and `credits.full` (#create_view_without_credits).
3. You use tables that include pre-existing schema properties `credits.type`, `credits.id`, and `credits.full` but do *not* include the new schema properties `project.number` and `adjustment_info`. For an example of how to create this view, see Creating a view for tables without `project.number` and `adjustment_info` (#create_view_without_project_and_adjustment).

You can create a view by composing a SQL query that is used to define the data accessible to the view. For further details, see Creating a view (/bigquery/docs/views#creating_a_view).

The following are a summary of the steps to create a BigQuery view.

1. Select the query to create the view
2. Run the query and observe the results
3. Save the view
4. Enter the name for the new view
5. Observe the schema of the new view

1. Creating a view for tables with all fields in the updated schema

The following is a query that will create a new view using both the pre-existing and updated schemas. This type of view limits your exposure to future schema changes.

By using this view for your queries, they will all have the same schema and allow `UNION` statements to work successfully. This query preserves the `credits.type`, `credits.id`, `credits.full`, `project.number`, and `adjustment_info` fields and values from the underlying tables.

Standard SQL

```
SELECT
    billing_account_id,
    STRUCT(service.id as id,
        service.description as description) as service,
    STRUCT(sku.id as id,
        sku.description as description) as sku,
    usage_start_time,
    usage_end_time,
    STRUCT(
        project.id as id,
        project.name as name,
        project.number as number,
        ARRAY(SELECT AS STRUCT
            label.key as key,
            label.value as value,
            FROM UNNEST(project.labels) as label) as labels,
        project.ancestry_numbers as ancestry_numbers) as project,
    ARRAY(SELECT AS STRUCT
        label.key as key,
        label.value as value,
        FROM UNNEST(labels) as label) as labels,
    ARRAY(SELECT AS STRUCT
        system_label.key as key,
        system_label.value as value,
        FROM UNNEST(system_labels) as system_label) as system_labels,
    STRUCT(
        location.location as location,
        location.country as country,
        location.region as region,
        location.zone as zone) as location,
    export_time,
    cost,
    currency,
    currency_conversion_rate,
    STRUCT(
        usage.amount as amount,
        usage.unit as unit,
        usage.amount_in_pricing_units as amount_in_pricing_units,
        usage.pricing_unit as pricing_unit) as usage,
    ARRAY(SELECT AS STRUCT
        credit.name as name,
        credit.amount as amount,
        credit.type as type,
        credit.id as id,
        credit.full_name as full_name,
        FROM UNNEST(credits) as credit) as credits,
    STRUCT(
        invoice.month as month) as invoice,
    cost_type,
    STRUCT(
        adjustment_info.id as id,
        adjustment_info.description as description,
```

```
        adjustment_info.mode as mode,
        adjustment_info.type as type) as adjustment_info,
    FROM TABLE_WITH_CREDITINFO_PROJECT_NUMBER_AND_ADJUSTMENT_INFO
```

2. Creating a view for tables without `credits.type`, `credits.id`, and `credits.full`

The following is a query that will create a new view using tables that do *not* include the pre-existing schema properties `credits.type`, `credits.id`, and `credits.full`.

Standard SQL

```
SELECT
    billing_account_id,
    STRUCT(service.id as id,
        service.description as description) as service,
    STRUCT(sku.id as id,
        sku.description as description) as sku,
    usage_start_time,
    usage_end_time,
    STRUCT(
        project.id as id,
        project.name as name,
        CAST(NULL as string) as number,
        ARRAY(SELECT AS STRUCT
            label.key as key,
            label.value as value,
            FROM UNNEST(project.labels) as label) as labels,
        project.ancestry_numbers as ancestry_numbers) as project,
    ARRAY(SELECT AS STRUCT
        label.key as key,
        label.value as value,
        FROM UNNEST(labels) as label) as labels,
    ARRAY(SELECT AS STRUCT
        system_label.key as key,
        system_label.value as value,
        FROM UNNEST(system_labels) as system_label) as system_labels,
    STRUCT(
        location.location as location,
        location.country as country,
        location.region as region,
        location.zone as zone) as location,
    export_time,
    cost,
    currency,
    currency_conversion_rate,
    STRUCT(
        usage.amount as amount,
        usage.unit as unit,
        usage.amount_in_pricing_units as amount_in_pricing_units,
        usage.pricing_unit as pricing_unit) as usage,
```

```
    ARRAY(SELECT AS STRUCT
        credit.name as name,
        credit.amount as amount,
        CAST(NULL as STRING) as type,
        CAST(NULL as STRING) as id,
        CAST(NULL as STRING) as full_name,
        FROM UNNEST(credits) as credit) as credits,
    STRUCT(
        invoice.month as month) as invoice,
    cost_type,
    STRUCT(
        CAST(NULL as STRING) as id,
        CAST(NULL as STRING) as description,
        CAST(NULL as STRING) as mode,
        CAST(NULL as STRING) as type) as adjustment_info,
FROM TABLE_WITHOUT_CREDIT_ID_TYPE_FULL_NAME
```

3. Creating a view for tables without `project.number` and `adjustment_info`

The following is a query that will create a new view using tables that include the pre-existing schema properties **`credits.type`**, **`credits.id`**, and **`credits.full`** but do *not* include the new schema properties **`project.number`** and **`adjustment_info`**.

Standard SQL

```
SELECT
    billing_account_id,
    STRUCT(service.id as id,
        service.description as description) as service,
    STRUCT(sku.id as id,
        sku.description as description) as sku,
    usage_start_time,
    usage_end_time,
    STRUCT(
        project.id as id,
        project.name as name,
        CAST(NULL as string) as number,
        ARRAY(SELECT AS STRUCT
            label.key as key,
            label.value as value,
            FROM UNNEST(project.labels) as label) as labels,
        project.ancestry_numbers as ancestry_numbers) as project,
    ARRAY(SELECT AS STRUCT
        label.key as key,
        label.value as value,
        FROM UNNEST(labels) as label) as labels,
    ARRAY(SELECT AS STRUCT
        system_label.key as key,
        system_label.value as value,
```

```
        FROM UNNEST(system_labels) as system_label) as system_labels,
    STRUCT(
        location.location as location,
        location.country as country,
        location.region as region,
        location.zone as zone) as location,
    export_time,
    cost,
    currency,
    currency_conversion_rate,
    STRUCT(
        usage.amount as amount,
        usage.unit as unit,
        usage.amount_in_pricing_units as amount_in_pricing_units,
        usage.pricing_unit as pricing_unit) as usage,
    ARRAY(SELECT AS STRUCT
        credit.name as name,
        credit.amount as amount,
        credit.type as type,
        credit.id as id,
        credit.full_name as full_name,
        FROM UNNEST(credits) as credit) as credits,
    STRUCT(
        invoice.month as month) as invoice,
    cost_type,
    STRUCT(
        CAST(NULL as STRING) as id,
        CAST(NULL as STRING) as description,
        CAST(NULL as STRING) as mode,
        CAST(NULL as STRING) as type) as adjustment_info,
FROM TABLE_WITHOUT_PROJECTNUMBER_AND_ADJUSTMENT_INFO
```

4. Verifying that views are consistent with original tables

The following queries allow you to verify that the views you created provide data that is consistent with the original tables you were querying. The queries use the view created in the example without **`credits.type`**, **`credits.id`**, and **`credits.full`**. For details on how to create this view, see Creating a view for tables without **`credits.type`**, **`credits.id`**, and **`credits.full`** (#create_view_without_credits).

This query provides a row-by-row comparison of *cost* between the original table and the view created without **`credits.type`**, **`credits.id`**, and **`credits.full`**.

Standard SQL

```
-- ROW BY ROW COMPARISON OF COST BETWEEN ORIGINAL TABLE AND CONVERTED TABLE
SELECT cost FROM TABLE_WITHOUT_CREDIT_ID_TYPE_FULL_NAME
EXCEPT DISTINCT
SELECT cost FROM TABLE_WITHOUT_CREDIT_ID_TYPE_FULL_NAME_VIEW
```

This query provides a row by row comparison of *credits* between the original table and the view created without `credits.type`, `credits.id`, and `credits.full`.

Standard SQL

```
-- ROW BY ROW COMPARISON OF CREDITS BETWEEN ORIGINAL TABLE AND CONVERTED TABLE
WITH CONCAT_AMOUNTS AS (SELECT ARRAY_CONCAT_AGG(ARRAY(SELECT amount FROM UNNEST(credits) as cr
CONCAT_AMOUNTS_CONVERTED AS (SELECT ARRAY_CONCAT_AGG(ARRAY(SELECT amount FROM UNNEST(credits)

SELECT amounts FROM CONCAT_AMOUNTS, UNNEST(amounts) as amounts
EXCEPT DISTINCT
SELECT amounts FROM CONCAT_AMOUNTS_CONVERTED, UNNEST(amounts) as amounts
```

Related topics

- Understanding Cloud Billing data tables in BigQuery. (/billing/docs/how-to/export-data-bigquery-tables)
- Set up Cloud Billing data export to BigQuery (/billing/docs/how-to/export-data-bigquery-setup)
- Example queries for Cloud Billing data export to BigQuery (/billing/docs/how-to/bq-examples)
- Visualize spend over time with Data Studio (/billing/docs/how-to/visualize-data)

Last updated 2022-07-25 UTC.