On this page

BigQuery configurations

Use `project` and `dataset` in configurations

- `schema` is interchangeable with the BigQuery concept `dataset`
- `database` is interchangeable with the BigQuery concept of `project`

For our reference documentation, you can declare `project` in place of `database.` This will allow you to read and write from multiple BigQuery projects. Same for `dataset`.

Using table partitioning and clustering

Partition clause

▶ *Changelog*

BigQuery supports the use of a partition by clause to easily partition a table by a column or expression. This option can help decrease latency and cost when querying large tables. Note that partition pruning only works when partitions are filtered using literal values (so selecting partitions using a subquery won't improve performance).

The `partition_by` config can be supplied as a dictionary with the following format:

```
{
  "field": "<field name>",
  "data_type": "<timestamp | date | datetime | int64>",
  "granularity": "<hour | day | month | year>"

  # Only required if data_type is "int64"
  "range": {
    "start": <int>,
```

```
    "end": <int>,
    "interval": <int>
  }
}
```

Partitioning by a date or timestamp

▶ *Changelog*

When using a `datetime` or `timestamp` column to partition data, you can create partitions with a granularity of hour, day, month, or year. A `date` column supports granularity of day, month and year. Daily partitioning is the default for all column types.

If the `data_type` is specified as a `date` and the granularity is day, dbt will supply the field as-is when configuring table partitioning.

Source code | Compiled code

bigquery_table.sql

```
{{ config(
    materialized='table',
    partition_by={
      "field": "created_at",
      "data_type": "timestamp",
      "granularity": "day"
    }
)}}

select
  user_id,
  event_name,
  created_at

from {{ ref('events') }}
```

Partitioning with integer buckets

If the `data_type` is specified as `int64`, then a `range` key must also be provied in the `partition_by` dict. dbt will use the values provided in the `range` dict to generate the

partitioning clause for the table.

Source code | Compiled code

bigquery_table.sql

```
{{ config(
    materialized='table',
    partition_by={
      "field": "user_id",
      "data_type": "int64",
      "range": {
        "start": 0,
        "end": 100,
        "interval": 10
      }
    }
)}}

select
  user_id,
  event_name,
  created_at

from {{ ref('events') }}
```

Additional partition configs

▶ *Changelog*

If your model has `partition_by` configured, you may optionally specify two additional configurations:

- `require_partition_filter` (boolean): If set to `true`, anyone querying this model *must* specify a partition filter, otherwise their query will fail. This is recommended for very large tables with obvious partitioning schemes, such as event streams grouped by day. Note that this will affect other dbt models or tests that try to select from this model, too.

- `partition_expiration_days` (integer): If set for date- or timestamp-type partitions, the partition will expire that many days after the date it represents. E.g. A partition representing `2021-01-01`, set to expire after 7 days, will no longer be queryable as of `2021-01-08`, its storage costs zeroed out,

and its contents will eventually be deleted. Note that table expiration will take precedence if specified.

bigquery_table.sql

```
{{ config(
    materialized = 'table',
    partition_by = {
      "field": "created_at",
      "data_type": "timestamp",
      "granularity": "day"
    },
    require_partition_filter = true,
    partition_expiration_days = 7
)}}
```

Clustering Clause

BigQuery tables can be clustered to colocate related data.

Clustering on a single column:

bigquery_table.sql

```
{{
  config(
    materialized = "table",
    cluster_by = "order_id",
  )
}}

select * from ...
```

Clustering on a multiple columns:

bigquery_table.sql

```
{{
  config(
    materialized = "table",
```

```
    cluster_by = ["customer_id", "order_id"],
  )
}}

select * from ...
```

Managing KMS Encryption

Customer managed encryption keys can be configured for BigQuery tables using the `kms_key_name` model configuration.

Using KMS Encryption

To specify the KMS key name for a model (or a group of models), use the `kms_key_name` model configuration. The following example sets the `kms_key_name` for all of the models in the `encrypted/` directory of your dbt project.

dbt_project.yml

```
name: my_project
version: 1.0.0

...

models:
  my_project:
    encrypted:
      +kms_key_name:
'projects/PROJECT_ID/locations/global/keyRings/test/cryptoKeys/quickstart'
```

Labels and Tags

Specifying labels

dbt supports the specification of BigQuery labels for the tables and views that it creates. These labels can be specified using the `labels` model config.

The `labels` config can be provided in a model config, or in the `dbt_project.yml` file, as shown below.

> **Note**
>
> BigQuery requires that both key-value pair entries for labels have a maximum length of 63 characters.

Configuring labels in a model file

model.sql

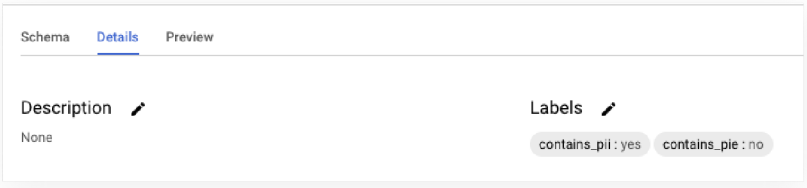
```
{{
  config(
    materialized = "table",
    labels = {'contains_pii': 'yes', 'contains_pie': 'no'}
  )
}}

select * from {{ ref('another_model') }}
```

Configuring labels in dbt_project.yml

dbt_project.yml

```
models:
  my_project:
    snowplow:
      +labels:
        domain: clickstream
    finance:
      +labels:
        domain: finance
```



Viewing labels in the BigQuery console

Specifying tags

BigQuery table and view *tags* can be created by supplying an empty string for the label value.

model.sql

```
{{
  config(
    materialized = "table",
    labels = {'contains_pii': ''}
  )
}}

select * from {{ ref('another_model') }}
```

Policy tags

BigQuery enables column-level security by setting policy tags on specific columns.

dbt enables this feature as a column resource property, `policy_tags` (*not* a node config).

models/<filename>.yml

```
version: 2

models:
- name: policy_tag_table
  columns:
    - name: field
      policy_tags:
        - 'need_to_know'
```

Please note that in order for policy tags to take effect, column-level `persist_docs` must be enabled for the model, seed, or snapshot.

Merge behavior (incremental models)

The `incremental_strategy` config controls how dbt builds incremental models. dbt uses a merge statement on BigQuery to refresh incremental tables.

The `incremental_strategy` config can be set to one of two values:

- `merge` (default)
- `insert_overwrite`

Performance and cost

The operations performed by dbt while building a BigQuery incremental model can be made cheaper and faster by using clustering keys in your model configuration. See this guide for more information on performance tuning for BigQuery incremental models.

Note: These performance and cost benefits are applicable to incremental models built with either the `merge` or the `insert_overwrite` incremental strategy.

The `merge` strategy

The `merge` incremental strategy will generate a `merge` statement that looks something like:

```
merge into {{ destination_table }} DEST
using ({{ model_sql }}) SRC
on SRC.{{ unique_key }} = DEST.{{ unique_key }}

when matched then update ...
when not matched then insert ...
```

The `merge` approach has the benefit of automatically updating any late-arriving facts in the destination incremental table. The drawback of this approach is that BigQuery must scan all source tables referenced in the model SQL, as well as the entirety of the destination table. This can be slow and costly if the incremental model is transforming very large amounts of data.

Note: The `unique_key` configuration is required when the `merge` incremental strategy is selected.

The `insert_overwrite` strategy

▶ *Changelog*

The `insert_overwrite` strategy generates a merge statement that replaces entire partitions in the destination table. **Note:** this configuration requires that the model is configured with a Partition clause. The `merge` statement that dbt generates when the `insert_overwrite` strategy is selected looks something like:

```
/*
  Create a temporary table from the model SQL
*/
create temporary table {{ model_name }}__dbt_tmp as (
  {{ model_sql }}
);

/*
  If applicable, determine the partitions to overwrite by
  querying the temp table.
*/

declare dbt_partitions_for_replacement array<date>;
set (dbt_partitions_for_replacement) = (
    select as struct
        array_agg(distinct date(max_tstamp))
    from `my_project`.`my_dataset`.{{ model_name }}__dbt_tmp
);

/*
  Overwrite partitions in the destination table which match
  the partitions in the temporary table
*/
merge into {{ destination_table }} DEST
using {{ model_name }}__dbt_tmp SRC
on FALSE

when not matched by source and {{ partition_column }} in
unnest(dbt_partitions_for_replacement)
then delete

when not matched then insert ...
```

For a complete writeup on the mechanics of this approach, see this explainer post.

Determining partitions to overwrite

dbt is able to determine the partitions to overwrite dynamically from the values present in the temporary table, or statically using a user-supplied configuration.

The "dynamic" approach is simplest (and the default), but the "static" approach will reduce costs by eliminating multiple queries in the model build script.

Static partitions

To supply a static list of partitions to overwrite, use the `partitions` configuration.

models/session.sql

```
{% set partitions_to_replace = [
  'timestamp(current_date)',
  'timestamp(date_sub(current_date, interval 1 day))'
] %}

{{
  config(
    materialized = 'incremental',
    incremental_strategy = 'insert_overwrite',
    partition_by = {'field': 'session_start', 'data_type': 'timestamp'},
    partitions = partitions_to_replace
  )
}}

with events as (

    select * from {{ref('events')}}

    {% if is_incremental() %}
        -- recalculate yesterday + today
        where date(event_timestamp) in ({{ partitions_to_replace | join(',') }})
    {% endif %}

),

... rest of model ...
```

This example model serves to replace the data in the destination table for both *today* and *yesterday* every day that it is run. It is the fastest and cheapest way to incrementally update a table using dbt. If we wanted this to run more dynamically— let's say, always for the past 3 days—we could leverage dbt's baked-in datetime macros and write a few of our own.

▶ *Changelog*

Think of this as "full control" mode. You must ensure that expressions or literal values in the the `partitions` config have proper quoting when templated, and that they match the `partition_by.data_type` (`timestamp`, `datetime`, `date`, or `int64`). Otherwise, the filter in the incremental `merge` statement will raise an error.

Dynamic partitions

If no `partitions` configuration is provided, dbt will instead:

1. Create a temporary table for your model SQL
2. Query the temporary table to find the distinct partitions to be overwritten
3. Query the destination table to find the *max* partition in the database

When building your model SQL, you can take advantage of the introspection performed by dbt to filter for only *new* data. The max partition in the destination table will be available using the `_dbt_max_partition` BigQuery scripting variable. **Note:** this is a BigQuery SQL variable, not a dbt Jinja variable, so no jinja brackets are required to access this variable.

Example model SQL:

```
{{
  config(
    materialized = 'incremental',
    partition_by = {'field': 'session_start', 'data_type': 'timestamp'},
    incremental_strategy = 'insert_overwrite'
  )
}}

with events as (

  select * from {{ref('events')}}

  {% if is_incremental() %}

    -- recalculate latest day's data + previous
    -- NOTE: The _dbt_max_partition variable is used to introspect the
destination table
    where date(event_timestamp) >= date_sub(date(_dbt_max_partition), interval 1
day)
```

```
{% endif %}

),

... rest of model ...
```

Controlling table expiration

▶ *Changelog*

By default, dbt-created tables never expire. You can configure certain model(s) to expire after a set number of hours by setting `hours_to_expiration`.

dbt_project.yml

```
models:
  <resource-path>:
    +hours_to_expiration: 6
```

models/<modelname>.sql

```
{{ config(
    hours_to_expiration = 6
) }}

select ...
```

Authorized Views

▶ *Changelog*

If the `grant_access_to` config is specified for a model materialized as a view, dbt will grant the view model access to select from the list of datasets provided. See BQ docs on authorized views for more details.

Note

The `grants` config and the `grant_access_to` config are distinct.

- **`grant_access_to`:** Enables you to set up authorized views. When configured, dbt provides an authorized view access to show partial information from other datasets, without providing end users with full access to those underlying datasets. For more information, see "BigQuery configurations: Authorized views"
- **`grants`:** Provides specific permissions to users, groups, or service accounts for managing access to datasets you're producing with dbt. For more information, see "Resource configs: grants"

You can use the two features together: "authorize" a view model with the `grants_access_to` configuration, and then add `grants` to that view model to share its query results (and *only* its query results) with other users, groups, or service accounts.

dbt_project.yml

```yaml
models:
  <resource-path>:
    +grant_access_to:
      - project: project_1
        dataset: dataset_1
      - project: project_2
        dataset: dataset_2
```

models/<modelname>.sql

```sql
{{ config(
    grant_access_to=[
      {'project': 'project_1', 'dataset': 'dataset_1'},
      {'project': 'project_2', 'dataset': 'dataset_2'}
    ]
) }}
```

Views with this configuration will be able to select from objects in `project_1.dataset_1` and `project_2.dataset_2`, even when they are located elsewhere and queried by users who do not

otherwise have access to `project_1.dataset_1` and `project_2.dataset_2`.

Limitations

The `grant_access_to` config is not thread-safe when multiple views need to be authorized for the same dataset. The initial `dbt run` operation after a new `grant_access_to` config is added should therefore be executed in a single thread. Subsequent runs using the same configuration will not attempt to re-apply existing access grants, and can make use of multiple threads.