Incrementally building persistent tables

Beta Guide

Disclaimer:

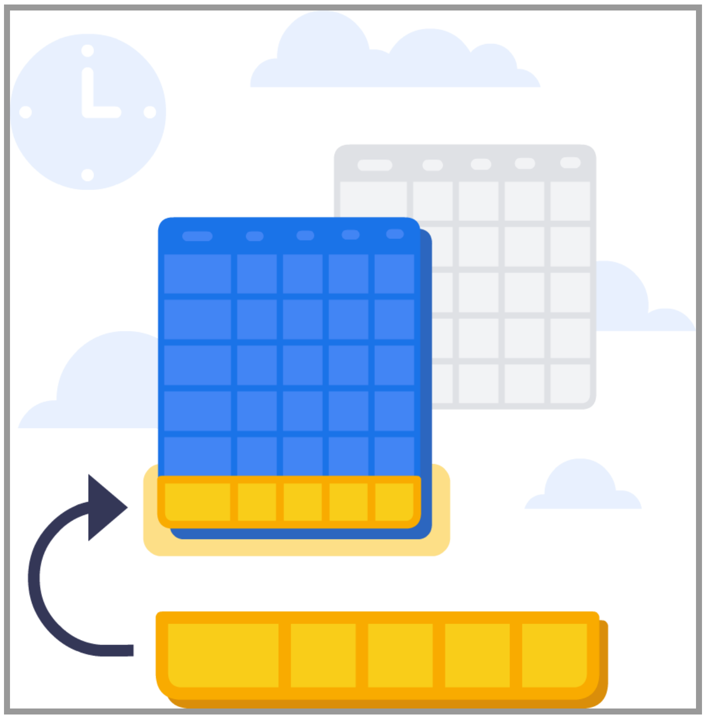
- This feature is in closed beta. Our support personnel are not trained on this feature and will not be able to help you if there are problems
- As a beta participant you are encouraged to give feedback. Please send all feedback to aaleksic@google.com

What are incrementally building persistent tables?

Today Looker supports various ways for users to create new tables in their database. These tables, known as derived tables, can be ephemeral or persistent. Ephemeral derived tables have to be re-created every time they are queried, whereas persistent tables are stored and can be referenced for a specified period of time until they are dropped and rebuilt. Looker has two types of persistent tables:

- Persistent derived tables (PDTs), which can be SQL-based derived tables or native derived tables (LookML based). PDTs can improve query performance in many cases and reduce database strain, since the table does not have to be recreated every time a user queries it. To learn more about derived tables, check out our documentation here.
- Aggregate tables, which Looker developers create to act as roll-ups or summary tables that Looker can query instead of raw, atomic data tables. This feature is called aggregate awareness, which you can learn about here.

The PDTs and aggregate tables that Looker creates today are rebuilt in their entirety every time they need to be updated according to a trigger or persist for value. There is no way to append data to an existing table without rebuilding the whole thing. This can result in long build times when you are working with large amounts of data.



In this closed beta users will be able to define incrementally building PDTs. What this means is that instead of the table being rebuilt in its entirety, fresh data will be appended to the table. The benefit here is that it will reduce build times for large persistent tables as now only a small subset of the entire persistent table is being built rather than the persistent table in its entirety.

What does this look like in Looker?

Defining incremental builds

Two new LookML parameters have been added:

- **increment_key** (required), which defines the time period for which new records should be queried. Looker recommends using a timeframe belonging to a dimension group of **type: time**, but it is also possible to use a column name from the **explore_source** in the case of an NDT. The **increment_key** must specify a dimension or column that is included in the table, and can only specify truncated absolute times, such as day, month, year, fiscal quarter, and so on. Time frames such as day of the week are not supported. The **increment_key** parameter is required to define a table as an incrementally building persistent table.

- **increment_offset** (optional), which is an integer that defines the number of previous time periods (at the increment key's granularity) that will be rebuilt. This is helpful in the case of late-arriving data, where data from previous time periods that may not have been included when the table was last built. For example, if your **increment_key** is day and your **increment_offset** is **3**, then when the incremental table builds, it will append data going back three days in time. An **increment_offset** is not required to define an incremental build. If no **increment_offset** is specified, it defaults to **0**.

The filters that are written for increments will work in conjunction with any existing filters in the PDT. For example, let's say you have a table that goes back to 2015 and want to build an incremental table starting from 2019, you should define this as a normal filter for the PDT. During the initial build for an incremental PDT it will first build in its entirety as a regular PDT would and then build incrementally going forward based on the **increment_key** and optional **increment_offset** parameters that you defined.

Example: Defining an incremental build on a LookML based PDT

```
derived_table: {
   datagroup_trigger: usagetable_etl
   increment_key: "event_month"
   increment_offset: 1
   explore_source: events {
     column: name { field: account.name }
     column: account_id { field: account.id }
     column: database_dialect {}
     column: count_events {}
     column: event_month {}
     filters: {
       field: events.yes
       value: "No"
     }
     filters: {
       field: events.database_dialect
       value: "-EMPTY"
     }
   }
 }
```

Defining incremental builds for Aggregate Tables

When defining incremental builds for aggregate tables, the **increment_key** and **increment_offset** need to be defined within the materialization block.

The **increment_key** must point to a dimension that’s specified in the `aggregate_table` > `query` > `dimensions` list.

Example: Defining an incremental build on an aggregate table

```
aggregate_table: product_usage {
  materialization: {
    datagroup_trigger: usagetable_etl
    increment_key: “user_events.event_month”
    increment_offset: 1
  }
  query: {
    dimensions: [
      account.id
      , account.name
      , account.is_current_customer
      , user_events.event_month
    ]
    measures: [
      user_events.number_of_active_users
    ]
    filters: [
      user_events.is_production_instance: "Yes"
      , user_events.event_month: "12 months"
    ]
  }
}
```

Defining incremental builds for SQL-defined PDTs

To define an incremental build for a SQL-defined PDT you use the same syntax as above (***increment_key*** and ***increment_offset***) but must also add a Liquid filter to your SQL query that connects the increment key to a database time column. For example:

WHERE {% incrementcondition %} created_at {% endincrementcondition %}

Example 1: Hourly increment on a SQL query with a timestamp

The increment key corresponds to a dimension defined in your view. The timeframe of the dimension controls the increment for this table. In this example, our increment is hourly, so we use `event_hour`. This dimension should have SQL that refers to a column in your SQL-derived table. That column name should appear in the WHERE clause. In this case, the column name is `event_timestamp_column`. This column does not have to be truncated to the same timeframe as the dimension (in this example, the column is a timestamp).

```
view: my_sql_derived_table {
  derived_table: {
    datagroup_trigger: trigger_condition
    increment_key: "event_hour"
    increment_offset: 1
    sql: SELECT event_timestamp_column
        , other_column1
        , other_column2
    FROM my_database_table
    WHERE {% incrementcondition %} event_timestamp_column {% endincrementcondition
%} ;;

  }

  #### DIMENSIONS ####
  dimension_group: event {
    type: time
    timeframes: [raw, hour, date, week, month, year]
    datatype: timestamp
    sql:  ${TABLE}.event_timestamp_column ;;
  }
}
```

Example 2: Daily increment on a SQL query with a limit on the minimum date

In this example, the table we are selecting from (`my_database_table`) has information going back very far in time. We don't want our incrementally built table to have that much data, so we add a second WHERE clause that restricts the minimum data for the first table build (`event_date_column > '2020-01-01'`).

```
view: my_sql_derived_table {
  derived_table: {
    datagroup_trigger: trigger_condition
    increment_key: "event_date"
    increment_offset: 1
    sql: SELECT event_date_column
        , other_column1
        , other_column2
    FROM my_database_table
    WHERE {% incrementcondition %} event_date_column {% endincrementcondition %}
        AND event_date_column > '2020-01-01';;

  }

  #### DIMENSIONS ####
  dimension_group: event {
    type: time
    timeframes: [raw, date, week, month, year]
    datatype: timestamp
    sql:  ${TABLE}.event_date_column ;;
  }
}
```

Example 3: 15 minute increment on a SQL query with an alias for the timestamp column

In this example, the our timestamp column is a string in the database and must be parsed in SQL (`PARSE_TIMESTAMP("%c", timestamp_text_column)`) and assigned an alias (`parsed_timestamp_column`). We can use the alias for the SQL in the dimension definition, but we use the SQL expression in the WHERE clause. Finally, to get a 15 minute increment, we use the dimension `event_minute15` as the increment_key.

```
view: my_sql_derived_table {
  derived_table: {
    datagroup_trigger: trigger_condition
    increment_key: "event_minute15"
    increment_offset: 1
    sql: SELECT PARSE_TIMESTAMP("%c", timestamp_text_column) as
parsed_timestamp_column
        , other_column1
        , other_column2
    FROM my_database_table
    WHERE {% incrementcondition %} PARSE_TIMESTAMP("%c", timestamp_text_column)
          {% endincrementcondition %} ;;

  }

  #### DIMENSIONS ####
  dimension_group: event {
    type: time
    timeframes: [raw, minute15, hour, date, week, month, year]
    datatype: timestamp
    sql:  ${TABLE}.parsed_timestamp_column ;;
  }
}
```

Monitoring incremental builds

You can use your Looker instance's System Activity PDT Event Log to monitor whether an incremental PDT builds successfully. As a starting point, you can:

- Filter on:
    - PDT Builds Dev Build (Yes / No): Select is No to find Production PDTs
    - PDT Builds End Date: Helps to narrow down your query to recent events
    - PDT Builds Model Name: Model name of your incremental PDT
    - PDT Builds View Name: View name of the incremental PDT
- Select the following result fields:
    - PDT Builds End Date
    - PDT Builds Count

    - PDT Builds Average Build Time Seconds

Which tables are good candidates for incremental builds?

In order for a table to be a good candidate for an incremental rebuild there are a few basic requirements that must be met:

- Only tables with immutable rows are good candidates (e.g. tables resembling event streams or sales data)
- The table must be persisted
- The table must have a timestamp column

Additionally, changing the definition of an incrementally built table will cause it to be rebuilt from scratch. Proceed with caution when changing the definition of a large incrementally built table.

Which database dialects are supported?

Currently we support the following dialects:

- Bigquery standard sql
- Mysql
- Postgres
- Redshift
- Snowflake
- Vertica
- Azure Synapse

If there is a dialect that you would like to see supported, but is not on this list, please reach out to aaleksic@google.com