Before we jump into all things about PDT, what actually is derived table?

Derived Table

A derived table is a technique for creating a temporary set of records which can be used within another query in SQL. You can use derived tables to shorten long queries, or even just to break a complex process into logical steps. In Looker, a derived table is a query whose results are used as if it were an actual table in the database.

Persistent Derived Table

A persistent derived table (PDT) is a derived table that is written into a scratch schema on your database and regenerated on the schedule that you specify with a persistence strategy in Looker. A PDT can be either a native derived table or a SQL-based derived table.

Requirements for PDTs

To use PDTs in your Looker project, you need the following:

A database dialect that supports PDTs. See the Supported database dialects for PDTs section later on this page for the lists of dialects that support persistent SQL-based derived tables and persistent native derived tables.

A scratch schema on your database. This can be any schema on your database, but we recommend creating a new schema that will be used only for this purpose. Your database administrator must configure the schema with write permission for the Looker database user.

A Looker connection that is configured with the Persistent Derived Tables option enabled. This is usually set up when you initially configure your Looker connection (see the Looker dialects documentation page for instructions for your database dialect), but you can also enable PDTs for your connection after the initial setup.

Why PDT

Persistent derived tables are the materialized results of a query, written to the Looker scratch schema in the connected database and rebuilt on a defined schedule. Commonly used to reduce database load and increase query performance, PDTs mean fewer rows of data are queried for subsequent responses. PDTs are also the underlying mechanism for Looker’s aggregate awareness, allowing Looker to intelligently query the smallest possible dataset for any query using computational algebra.



PDT Creation

To make a derived table into a persistent derived table (PDT), you define a persistence strategy for the table. To optimize performance, you should also add an optimization strategy.

Persistence Strategies

The persistence of a derived table can be managed by Looker or, for dialects that support materialized views, by your database using materialized views.

To make a derived table persistent, add one of the following parameters to the derived\_table definition:

Looker-managed persistence parameters:

datagroup\_trigger  
Looker maintains the PDT in the database until its datagroup is triggered. When the datagroup is triggered, Looker rebuilds the PDT to replace the previous version. This means that, in most cases, your users will not have to wait for the PDT to be built. If a user requests data from the PDT while it is being built and the query results aren't in the cache, Looker will return data from the existing PDT until the new PDT is built. See the Caching queries and rebuilding PDTs with datagroups documentation page for an overview of datagroups.

sql\_trigger\_value  
The sql\_trigger\_value parameter triggers the regeneration of a PDT based on a SQL statement that you provide. If the result of the SQL statement is different from the previous value, the PDT is regenerated. Otherwise, the existing PDT is maintained in the database. This means that, in most cases, your users will not have to wait for the PDT to be built. If a user requests data from the PDT while it is being built, and the query results aren't in the cache, Looker will return data from the existing PDT until the new PDT is built.

interval\_trigger  
The interval\_trigger parameter triggers the regeneration of a PDT based on a time interval that you provide, such as "24 hours" or "60 minutes".

persist\_for  
Yet another option is to use the persist\_for parameter to set the length of time the derived table should be stored before it is marked as expired, so that it is no longer used for queries and will be dropped from the database.

Database-managed persistence parameters:

materialized\_view: {yes}  
Materialized views allow you to leverage your database's functionality to persist derived tables in your Looker project. If your database dialect supports materialized views and your Looker connection is configured with the Persistent Derived Tables option enabled, you can create a materialized view by specifying materialized\_view: yes for a derived table. Materialised views are supported for both native derived tables and SQL-based derived tables.  
  
Materialised views allow you to leverage your database's functionality to persist derived tables in your Looker project.

With trigger-based persistence strategies (datagroup\_trigger, sql\_trigger, and interval\_trigger), Looker maintains the PDT in the database until the PDT is triggered for rebuild. When the PDT is triggered, Looker rebuilds the PDT to replace the previous version. This means that, with trigger-based PDTs, your users will not have to wait for the PDT to be built in order to get answers for Explore queries from the PDT.

PDT Creation Example

By using incremental strategy in PDT’s creation steps, it will let us to only append the most updated data into the existing PDT. Here’s the setup that is being used in most cases.



indexes : is used as as primary key for the view

increment\_key : is the parameter that makes a PDT into an incremental PDT by specifying the time increment for which fresh data should be queried and appended to the PDT.

increment\_offset (optional): is how long backfill data that will updated to PDT

datagroup\_trigger : is the main PDT that will be the base of the incremental PDT

distribution\_style : allows you to specify how the query for a persistent derived table (PDT) or an aggregate table is distributed across the nodes in a database.

Example 1  
This example uses a PDT with these properties:  
- Increment key: date  
- Increment offset: 3  
- Persistence strategy: triggered once a month on the first day of the month

Here is how this table will be updated:

A monthly persistence strategy means that the table is automatically built once a month. This means that on June 1st, for example, the last row in the table will have been added on May 1st.

Because this PDT has an increment key based on date, the PDT builder will truncate May 1st back to the beginning of the day and rebuild the data for May 1st and up to the current day, June 1st.

Additionally, this PDT has an increment offset of 3. So the PDT builder also rebuilds the data from the previous three time periods (days) before May 1st. The result is that data is rebuilt for April 28th, 29th, 30th, and up to the present day of June 1st.

Example 2  
This example uses a PDT with these properties:  
- Persistence strategy: triggered once a day  
- Increment key: month  
- Increment offset: 0

Here is how this table will be updated on June 1st:

The daily persistence strategy means that the table is automatically built once a day. On June 1st, the last row in the table will have been added on May 31st.

Because the increment key is based on the month, the PDT builder will truncate from May 31st back to the beginning of the month and rebuild the data for all of May and up to the current day, including June 1st.

Because this PDT has no increment offset, no previous time periods are rebuilt.

Here is how this table will be updated on June 2nd:

On June 2nd, the last row on the table will have been added on June 1st.

Because the PDT builder will truncate back to the beginning of the month of June, then rebuild the data starting with June 1st and up to the current day, the data is rebuilt for only June 1st and June 2nd.

Because this PDT has no increment offset, no previous time periods are rebuilt.

Example 3

This example uses a PDT with these properties:  
- Increment key: month  
- Increment offset: 3  
- Persistence strategy: triggered once a day

Here is how this table will be updated on June 1st:

The daily persistence strategy means that the table is automatically built once a day. On June 1st, for example, the last row in the table will have been added on May 31st.

Because the increment key is based on the month, the PDT builder will truncate from May 31st back to the beginning of the month and rebuild the data for all of May and up to the current day, including June 1st.

Additionally, this PDT has an increment offset of 3. This means that the PDT builder also rebuilds the data from the previous three time periods (months) before May. The result is that data is rebuilt from February, March, April, and up to the current day, June 1st.

Here is how this table will be updated on June 2nd:

On June 2nd, the last row in the table will have been added on June 1st.

The PDT builder will truncate the month back to June 1st and rebuild the data for the month of June, including June 2nd.

In addition, because of the increment offset, the PDT builder will rebuild the data from the previous three months before June. The result is that data is rebuilt from March, April, May, and up to the current day, June 2nd.

Pro and Cons

|  |  |  |  |
| --- | --- | --- | --- |
|  | PDT | PDT with Create Process | Aggregate Awareness |
| Pro | Simple to write  Easy to use as a rollup table in a narrow range of use cases  Simple to troubleshoot | Can be used to incrementally add data to the rollup table  Easy to use as a rollup table in a narrow range of use cases  Somewhat simple to troubleshoot | Most flexible in use  Looker intelligently chooses which agg table to use for a given query  Does not require writing SQL |
| Con | Expensive to recreate (the entire table rebuilds on the trigger interval)  May require complex logic to make it more flexible in use  Lack of monitoring option for PDT creation that resulting significant storage usage  May require additional procedure relates to data governance and query efficiency | More complex to write  May require complex logic to make it more flexible in use  Lack of monitoring option for PDT creation that resulting significant storage usage  May require additional procedure relates to data governance and query efficiency | Expensive to recreate (the entire table rebuilds on the trigger interval)  More complex to troubleshoot |

Conclusion

There are varies method that Halodoc take advantage of using PDT, still it is a great temporary solution for ad-hoc and one-timed projects that involve complex or big amount of data, it made the ability to revisit the table in the future for reference or documentation. In some other cases, creating physical table would seem more appropriate and efficient for long-term uses that would cover various cases.