Proprietary + Confidential



Derived Tables

Google Cloud

Hello again! In this module, we'll focus on derived tables, a great feature of Looker that you can use to create new custom tables that do not yet exist in your underlying database.

Proprietary + Confidential

Agenda

01	Introduction to Derived Tables
02	Types of Derived Tables
03	Example 5: Creating SQL Derived Tables with LookML
04	Example 6: Creating Native Derived Tables with LookML
05	Useful Parameters for Native Derived Tables
06	Persistent Derived Tables
07	Lab 2: Creating Derived Tables with LookML



Google Cloud

First, we'll introduce how derived tables work in Looker and review the types of derived tables that you can create using LookML.

Next, we'll walk through two examples of creating derived tables in Looker: one for SQL derived tables and another for native derived tables.

Then, we'll review some additional parameters for native derived tables that you may find useful to further customize these tables.

We'll also examined how to create persistent derived tables, which are written back to and stored in the underlying database.

Last, you'll get some hands-on practice with modeling dimensions and measures in Looker using Qwiklabs.

Hope you're excited to keep going!

Proprietary + Confidential

Derived Tables

01 Introduction to Derived Tables

02 Types of Derived Tables

03 Example 5: Creating SQL Derived Tables with LookML

04 Example 6: Creating Native Derived Tables with LookML

05 Useful Parameters for Native Derived Tables

06 Persistent Derived Tables

07 Lab 2: Creating Derived Tables with LookML



Google Cloud

Derived tables are a great feature of Looker that allows you create new tables that do not yet exist in a database.

In this section, we'll explore how to use derived tables to define new custom tables to address complex business questions.

Proprietary + Confidential

Going beyond the tables in your database

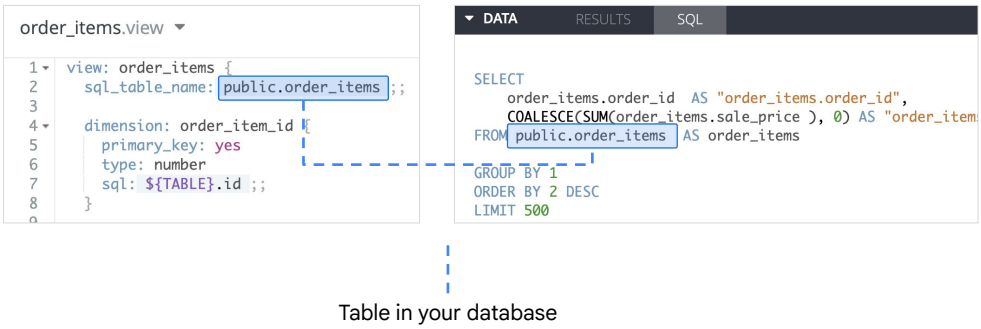


Table in your database

Examples are based on fictional data.

Google Cloud

Recall that when you work with existing tables in your connected database, you reference the necessary table names in the **`sql_table_name`** parameter of the view files.

You can also see the table names used by a view by reviewing the generated SQL for Explores queries.

Proprietary + Confidential

Going beyond the tables in your database

In SQL, when you create new tables or sub-selects, these can be known as:

Temporary tables or materialized views

```
--Temp Table:
CREATE TABLE dbo.#Cars
   (
   Car_id int NOT NULL,
   ColorCode varchar(10),
   ModelName varchar(20),
   Code int,
   DateEntered datetime
   )

INSERT INTO dbo.#Cars (Car_id, ColorCode, ModelName,
VALUES (1,'BlueGreen', 'Austen', 200801, GETDATE())
```

Common table expressions (CTEs)

```
WITH temporaryTable (averageValue) as
    (SELECT avg(Attr1)
    FROM Table),
    SELECT Attr1
    FROM Table
    WHERE Table.Attr1 > temporaryTable.averageValue;
```

Subqueries

```
SELECT column-names
  FROM table-name1
 WHERE value IN (SELECT column-name
                   FROM table-name2
                  WHERE condition)
```

Examples are based on fictional data.

Google Cloud

However, it is not unusual for existing database tables to be insufficient to assist us in answering more complex questions of our data. Sometimes we need to structure our data in other ways through SQL objects like temporary tables, materialized views, common table expressions (CTEs), and subqueries.

In LookML terms, if you need to produce a CTE or temp table, you need to define a *derived table*.

Proprietary + Confidential

Multi-step aggregations

This will fail:

```
SELECT MAX(SUM(sales))
FROM orders
GROUP BY department
```

Example use case:

Which department has the highest total sales?

- Find the total sales by department.
- Find the maximum of those totals.

Examples are based on fictional data.

Google Cloud

The most common use case for derived tables is to overcome structural limitations for patterns that would require a subquery in raw SQL. A nested or multi-step aggregation is a common pattern where we need to do an aggregate of an aggregate.

Consider the example of trying to find which department has the highest total sales. You may think to try something like:

```
SELECT MAX(SUM(sales))
FROM orders
GROUP BY department
```

However, this query won't work for two reasons:

1. We cannot take the max of a sum—any database dialect would complain about these nested aggregates.
2. The **GROUP BY** logic needs to be applied separately.

Proprietary + Confidential

Multi-step aggregations

This resolves the failure:

```
SELECT MAX(subquery.total_sales)
FROM
(SELECT SUM(sales) as total_sales
 FROM orders
 GROUP BY department) AS subquery
```

Example use case:

Which department has the highest total sales?

- Find the total sales by department.
- Find the maximum of those totals.

Examples are based on fictional data.

Google Cloud

To resolve the issue in SQL, you can isolate the first aggregation and grouping in a subquery, then find the max from those results using:

```
SELECT MAX(subquery.total_sales)
FROM
     (SELECT SUM(sales) as total_sales
      FROM orders
      GROUP BY department) AS subquery
```

Proprietary + Confidential

Multi-step aggregations

Example use case:

The **`order_items`** table has many **items** inside a single order, like a shopping cart. You may want to perform some shopping cart analysis, for example:

Examples are based on fictional data.

Google Cloud

Here's another scenario. Consider an **order_items** table that can have many items per order, like a shopping cart.

Proprietary + Confidential

Multi-step aggregations

Example use case:

The **`order_items`** table has many **items** inside a single order, like a shopping cart. You may want to perform some shopping cart analysis, for example:

Number of items per order

- Filter out single-item orders to answer questions on multi-item orders.
- Find the average number of items per order.

Examples are based on fictional data.

Google Cloud

Now, maybe you want to do some shopping cart analysis, like: are there any patterns to, or drivers behind, orders that contain multiple items as opposed to just one?

Or, what is the average number of items people tend to combine in a single order?

For the question, you would need to separate orders with one item each versus orders with multiple items, while the second question would be like an average of a count.

Proprietary + Confidential

Multi-step aggregations

Example use case:

The **`order_items`** table has many **items** inside a single order, like a shopping cart. You may want to perform some shopping cart analysis, for example:

Number of items per order

- Filter out single-item orders to answer questions on multi-item orders.
- Find the average number of items per order.

Total order cost

- Classify high-value orders from small-value orders.
- Find the average total order cost.

Examples are based on fictional data.

Google Cloud

Or maybe you want classify “high-” vs. “low-value” orders, or determine the average total order cost using a table that is granular to the item level.

For the first question, this would be like doing a **`CASE WHEN`** with each **order_id**’s sum of **sale_price**, while the second question is another inquiry about the average of a sum, specifically average customer lifetime spending.

Proprietary + Confidential

Multi-step aggregations: Example data

This is the starting structure of the data:

	Order ID	Item ID	Cost
	1	111	$10
	1	112	$20
	2	113	$60
	3	114	$15
	3	115	$25
	3	116	$40

Examples are based on fictional data.

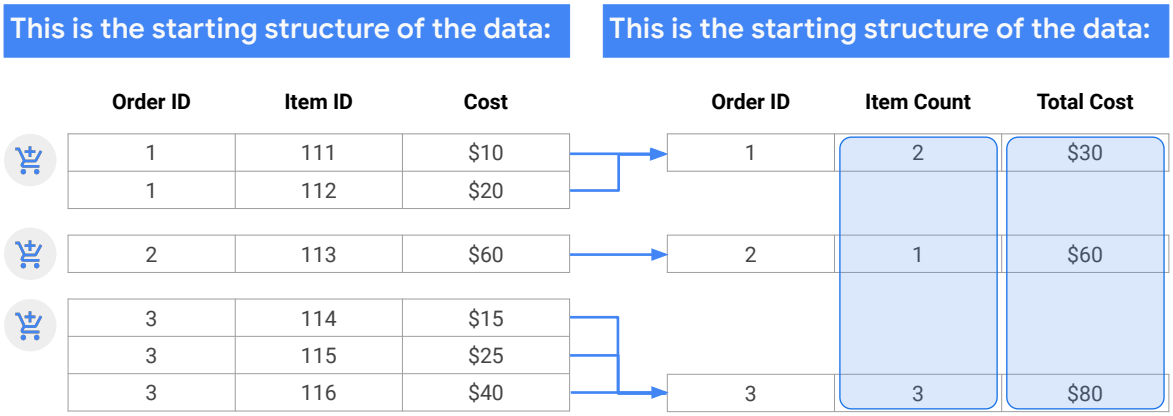
Google Cloud

To understand how derived tables can help answer these questions, let's take a look at an example dataset.

Imagine that to the left is the actual structure of the data in the table.

Proprietary + Confidential

Multi-step aggregations: Example data



Examples are based on fictional data.

Google Cloud

To the right is the table structure that you need to answer those various questions related to number of items per order and total order cost.

Specifically, you need to calculate each order ID's count of **item IDs** and sum of **cost**.

Using derived tables, you can create a new table of these two aggregations, and then use this new derived table to answer questions about orders based on the total number of items or total revenue.

Proprietary + Confidential

Derived tables can help!

You may want flexibility to **restructure data** and **define complex query logic** to do useful tasks like:

Use window functions to sessionalize event data.

Examples are based on fictional data.

Google Cloud

So far, we have explored a few use cases where we would need to pre-aggregate, or "dimensionalize," metrics using derived tables. Of course, there are many more situations in which derived tables would be useful, such as:

… analyzing behavioral patterns using raw event data that needs to be partitioned and organized into user sessions based on specific activity timeframes.

With a derived table, you can ask questions such as:

- How often does someone who starts on page A end up converting (doing a specific, desired action)?
- Or, how much more likely is a consumer to purchase Product B given that they purchased Product A?

Proprietary + Confidential

Derived tables can help!

You may want flexibility to **restructure data** and **define complex query logic** to do useful tasks like:

Use window functions to sessionalize event data.

Roll up user data to the month level to track periods of (in)activity and analyze retention.

Examples are based on fictional data.

Google Cloud

Another example might be order information that needs to be rolled up by customer and month, so you can determine things like the average amount of time that passes between one order and the next.

Proprietary + Confidential

Derived tables can help!

You may want flexibility to **restructure data** and **define complex query logic** to do useful tasks like:

Use window functions to sessionalize event data.

Roll up user data to the month level to track periods of (in)activity and analyze retention.

UNION different marketing channels' tables so that key performance indicators (KPIs) are standardized and aggregated across the channels.

Examples are based on fictional data.

Google Cloud

Or, maybe you are working with multiple tables for various marketing channels. You may want to combine these tables into one large table using a union, so that you can calculate and compare results across all the channels.

Proprietary + Confidential

Defining derived tables



Google Cloud

So, now that we know why derived tables are useful, how do we actually define derived tables to create new aggregations or tables that we need?

Proprietary + Confidential

Defining derived tables

Manually written or produced query with results that can be queried like a regular database table.

Integrated into Looker as views.

Can be joined into Explores just like standard views.

Can be ephemeral or written into the database (as a persistent derived table).

```
view: repeat_purchase_facts {
  derived_table: {
    sql: SELECT
        order_items.order_id
        , COUNT(DISTINCT repeat_order_items.id) AS number_subsequent_orders
        , MIN(repeat_order_items.created_at) AS next_order_date
        , MIN(repeat_order_items.order_id) AS next_order_id
      FROM ecomm.order_items
      LEFT JOIN ecomm.order_items repeat_order_items
        ON order_items.user_id = repeat_order_items.user_id
        AND order_items.created_at < repeat_order_items.created_at
      GROUP BY 1
      ;;
  }
```

Examples are based on fictional data.

Google Cloud

Derived tables are either manually written **SELECT** statements or Looker-generated **SELECT** statements that produce results that can be queried just like a “real” database table.

Proprietary + Confidential

Defining derived tables

Manually written or produced query with results that can be queried like a regular database table.

Integrated into Looker as views.

Can be joined into Explores just like standard views.

Can be ephemeral or written into the database (as a persistent derived table).

```
view: repeat_purchase_facts {
  derived_table: {
    sql: SELECT
        order_items.order_id
        , COUNT(DISTINCT repeat_order_items.id) AS number_subsequent_orders
        , MIN(repeat_order_items.created_at) AS next_order_date
        , MIN(repeat_order_items.order_id) AS next_order_id
      FROM ecomm.order_items
      LEFT JOIN ecomm.order_items repeat_order_items
        ON order_items.user_id = repeat_order_items.user_id
        AND order_items.created_at < repeat_order_items.created_at
      GROUP BY 1
      ;;
  }
```

Examples are based on fictional data.

Google Cloud

Derived tables are integrated in LookML projects as views. You can explore or join those views in the same way as the other views that point to actual database tables.

Proprietary + Confidential

Defining derived tables

Manually written or produced query with results that can be queried like a regular database table.

Integrated into Looker as views.

Can be joined into Explores just like standard views.

Can be ephemeral or written into the database (as a persistent derived table).

```
view: repeat_purchase_facts {
  derived_table: {
    sql: SELECT
        order_items.order_id
        , COUNT(DISTINCT repeat_order_items.id) AS number_subsequent_orders
        , MIN(repeat_order_items.created_at) AS next_order_date
        , MIN(repeat_order_items.order_id) AS next_order_id
      FROM ecomm.order_items
      LEFT JOIN ecomm.order_items repeat_order_items
        ON order_items.user_id = repeat_order_items.user_id
        AND order_items.created_at < repeat_order_items.created_at
      GROUP BY 1
      ;;
  }
```

Examples are based on fictional data.

Google Cloud

Derived tables have two types of existence:

- *Ephemeral* tables, which compile at runtime and are generated as CTEs or temp tables.
- *Persistent* tables, or PDTs, which are stored in the underlying database.

The benefit in persisting derived tables is that they are ready to go when business users need them, and therefore reduce query runtimes.

The downsides are that they take up storage space in your database (which may correlate to cost), and they are more rigid (they can't accept dynamic logic). So you may choose to persistent only some of your derived tables, such as the most frequently used ones.

Proprietary + Confidential

Lesson summary

Manually written or produced query with results that can be queried like a regular database table.	Integrated into Looker as views.	Can be joined into Explores just like standard views.	Can be ephemeral or written into the database (as a persistent derived table).

Google Cloud

In summary, derived tables are a useful feature for defining new custom tables that do not yet exist in your database.

Derived tables are used just like other views in your LookML project and produce results that can be queried just like a real database table. Depending on your use case, derived tables can be ephemeral or written back to the database as a persistent derived table.

With derived tables, you can create custom tables to aggregate your data in a variety of ways to help you answer business questions. Hope you have fun experimenting with derived tables in your organization's Looker instance!

Proprietary + Confidential

Derived Tables

01 Introduction to Derived Tables

02 Types of Derived Tables

03 Example 5: Creating SQL Derived Tables with LookML

04 Example 6: Creating Native Derived Tables with LookML

05 Useful Parameters for Native Derived Tables

06 Persistent Derived Tables

07 Lab 2: Creating Derived Tables with LookML



Google Cloud

In Looker, you can use derived tables to define new custom tables that do not exist in your underlying database.

Derived tables are created in LookML projects as views that you can join in an Explore, just like other views that point to actual database tables.

Derived tables also produce results that can be queried just like real database table, and they can be either ephemeral or written back to the database as a persistent derived table.

Proprietary + Confidential

Two ways to write derived tables

01

SQL derived table

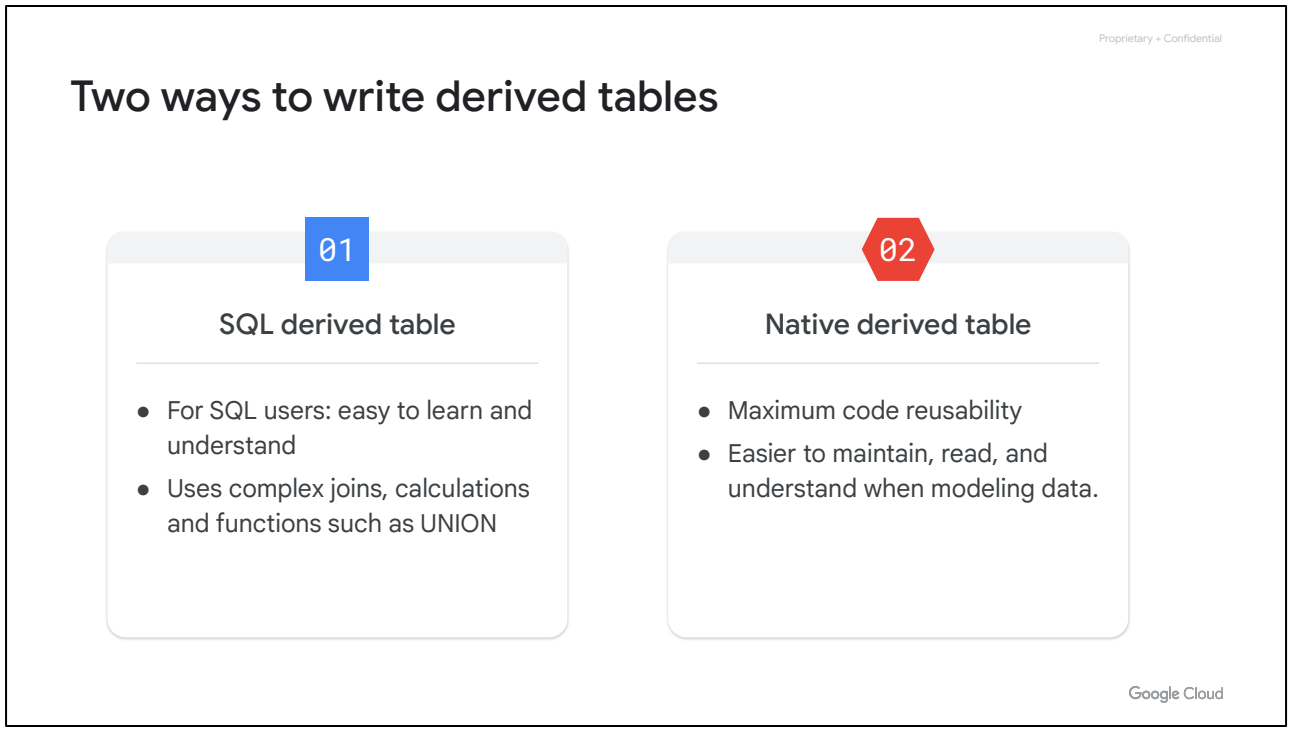
- For SQL users: easy to learn and understand
- Uses complex joins, calculations and functions such as UNION

Google Cloud

Independent from the two degrees of permanence (ephemeral or persistent), we can use two methods of defining derived tables.

First, derived tables can be written in SQL, which is an easy option if you're already comfortable with **SELECT** statements in SQL.

In addition to being easier to learn and understand for SQL users, SQL derived tables have the advantage of being able to leverage complex and custom joins and calculations, as well as the **UNION** function



Second, derived tables can also be written entirely in LookML, which we call *native* derived tables, or NDTs.

NDTs embody the essential LookML principle of reusability by allowing you to leverage existing objects in your model, such as dimensions and measures, to define new tables.

Because NDTs minimize the number of “hard-coded” database references in your code, they are easier to maintain and allow your model to scale efficiently as your organization’s usage of Looker increases.

Proprietary + Confidential

Example: Use case

Example: For each order, you want the:

- number of ordered items
- total price

Google Cloud

Let's explore a simple example to further highlight the key differences between SQL derived and native derived tables.

Imagine that you want to create a derived table that contains the numbered of ordered items and the total price for each order in your database.

Proprietary + Confidential

Example: SQL query

The query for a SQL derived table could include the following:

- `COUNT(*)` as `order_item_count`
- `SUM(order_items.sale_price)` as `order_revenue`

Google Cloud

You could create a SQL derived table by writing a **SELECT** statement that includes a **COUNT** of order items and a **SUM** of **sale_price**.

Proprietary + Confidential

Example: Native derived table

Example: For each order, you want the:

- number of ordered items
- total price

The query for a SQL derived table could include the following:

- **`COUNT(*)`** as **`order_item_count`**.
- **`SUM(order_items.sale_price)`** as **`order_revenue`**.

```
measure: order_item_count {
  type: count
  drill_fields: [detail*]
}
```

```
measure: total_revenue {
  type: sum
  value_format_name: usd
  sql: ${sale_price} ;;
  drill_fields: [detail*]
}
```

If these measures are already defined within the LookML in the order_items view, **why not use an NDT?**

Google Cloud

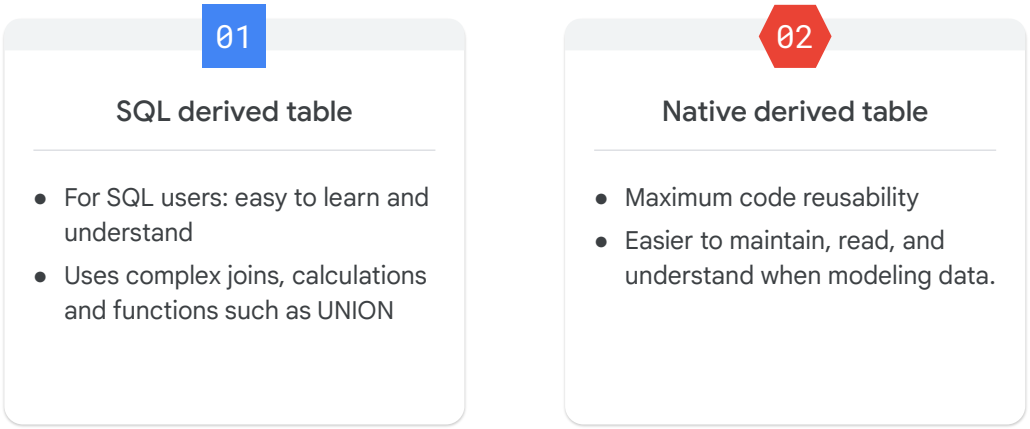
However, what if your LookML model already contains measures for **order_item_count** and **total_revenue**?

Rather than creating a SQL derived table, you can reuse these existing measures to easily define a new NDT.

This would allow you to avoid hard-coding another reference to the underlying database table, thereby saving you work in the future if and when you need to update those measures.

Proprietary + Confidential

Lesson summary



Google Cloud

In summary, there are two ways to define derived tables in Looker.

First, SQL derived tables are easy for SQL users to learn and understand, and can include complex joins and calculations, as well as the **`UNION`** function.

In contrast to SQL derived tables, native derived tables are expressed entirely in LookML and reuse existing objects in your model such as dimensions and measures to create new tables. They are often easier to maintain and scale because they minimize the number of “hard-coded” database references in your code.

Though which option you use will depend on your particular details of your LookML project, both SQL derived and native derived tables are great features in Looker that you can use to create new, custom tables for your business needs.

Proprietary + Confidential

Derived Tables

01 Introduction to Derived Tables

02 Types of Derived Tables

03 Example 5: Creating SQL Derived Tables with LookML

04 Example 6: Creating Native Derived Tables with LookML

05 Useful Parameters for Native Derived Tables

06 Persistent Derived Tables

07 Lab 2: Creating Derived Tables with LookML



Google Cloud

In Looker, you can use SQL derived tables to create new custom tables using manually written SQL queries.

Proprietary + Confidential

Example: Creating SQL derived tables with LookML

Create a SQL derived table that contains the following details for each order: order ID, user ID, number of items, and total cost of the items ordered.

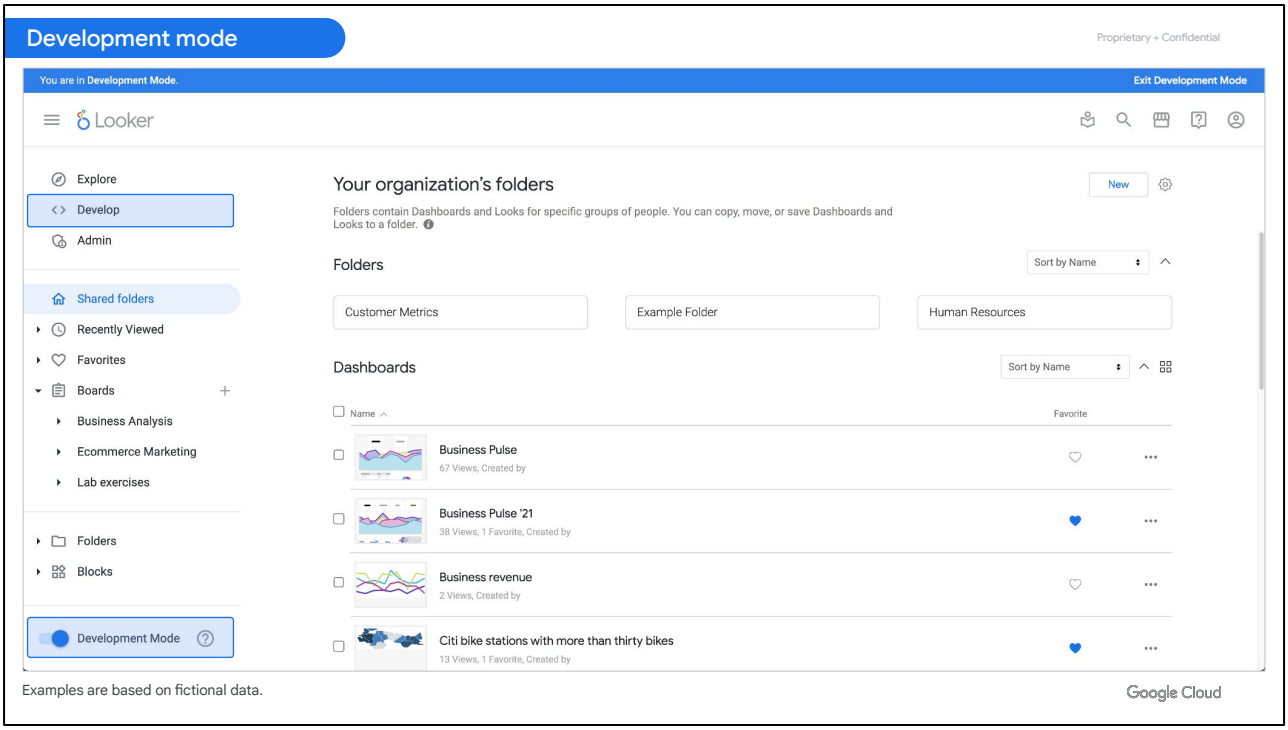


Google Cloud

For example, imagine that you want to create a table that contains the following details for each order in your database: order ID, user ID, the number of items ordered, and total price of the order.

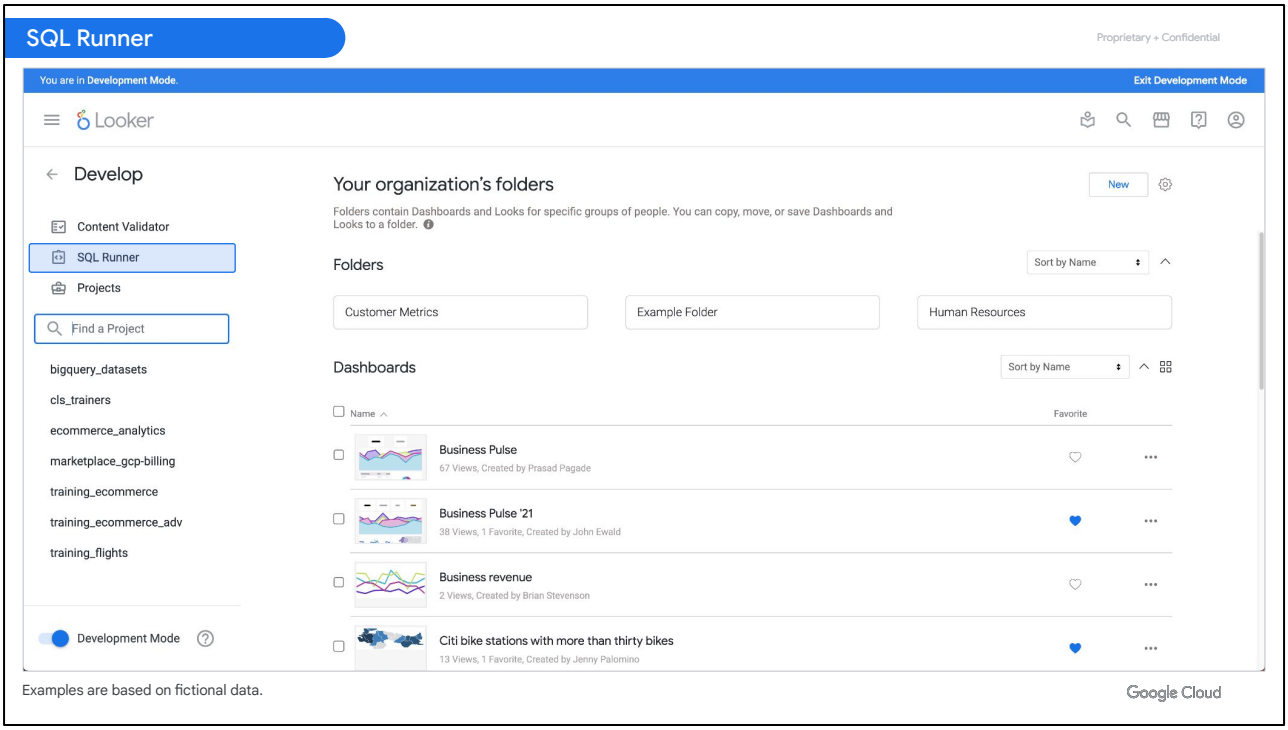
To create a SQL derived table with this information, you need to manually write a SQL query that selects the data attributes and aggregations that you want to include in the table.

Let's walk through an example of creating a SQL derived table that summarizes the desired information for each order.



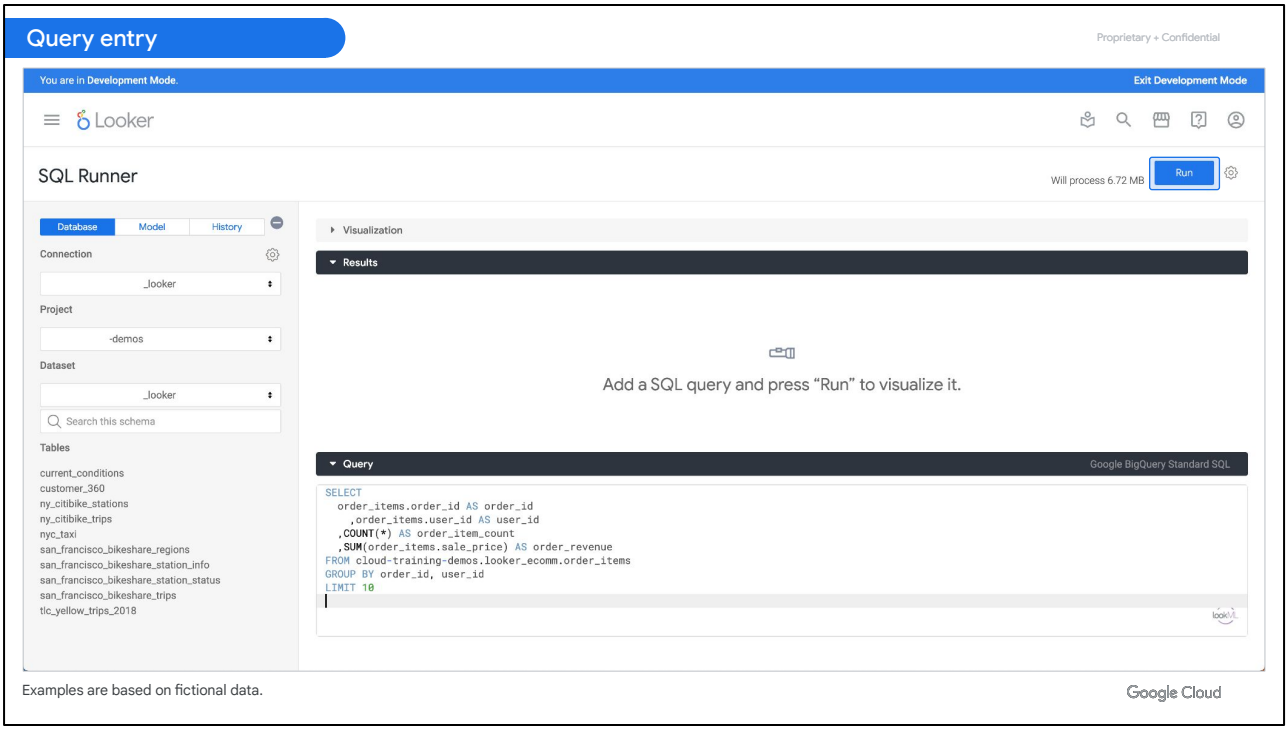
Begin by ensuring that you are in Development Mode. You can enable Development Mode by toggling the button for Development Mode on the bottom left corner of the Looker home page.

Then, click **Develop** on the left side navigation menu to see the develop options.



In Looker, the easiest way to create a SQL derived table is through SQL Runner, which provides an interface for you to write and test SQL queries to your available database connections.

To open this interface, click **SQL Runner** under the **Develop** options.



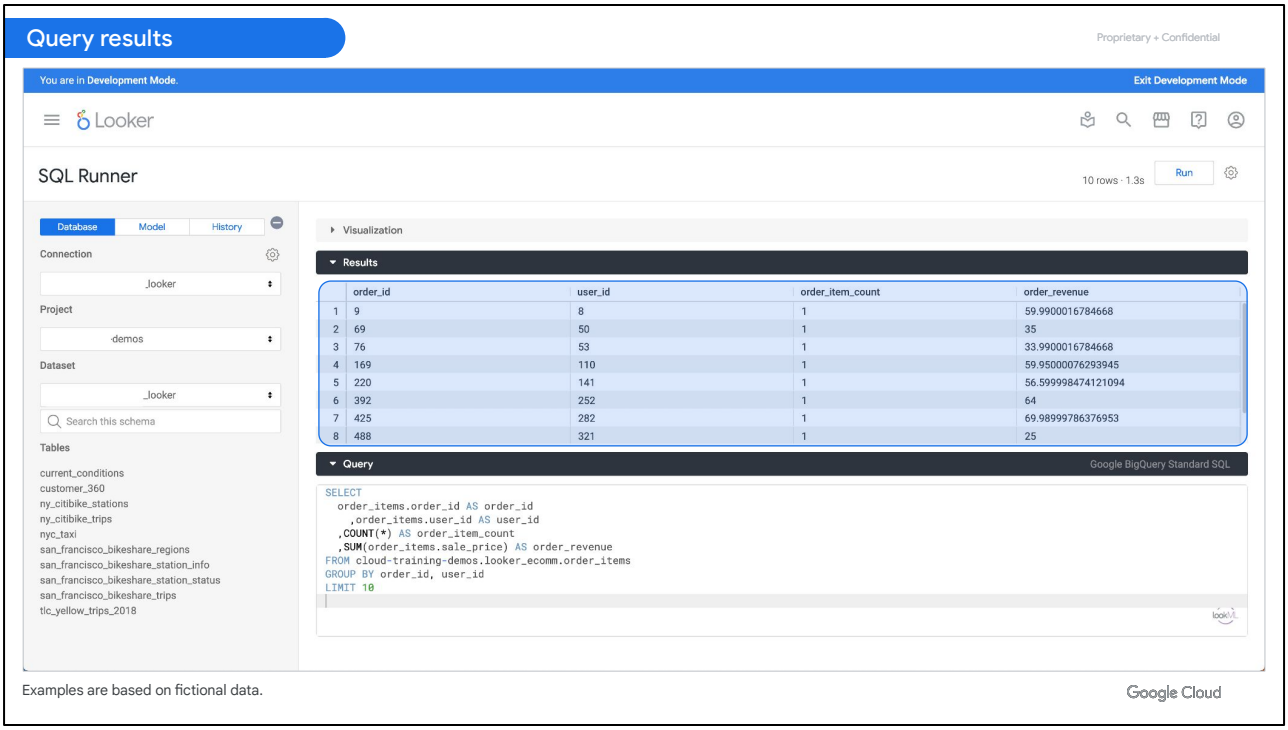
In **SQL Runner**, you can type out the desired query, and click **Run** to get the results**.**

For this example, the desired query selects the **order_id** and **user_id** from the **order_items** table, counts the number of items associated with each order, and then sums the price of those items.

Specifically, the **COUNT** clause is counting the number of individual order item IDs (the primary key of the **order_items** table), and the **SUM** clause is totaling the **sale_price** of the order item IDs.

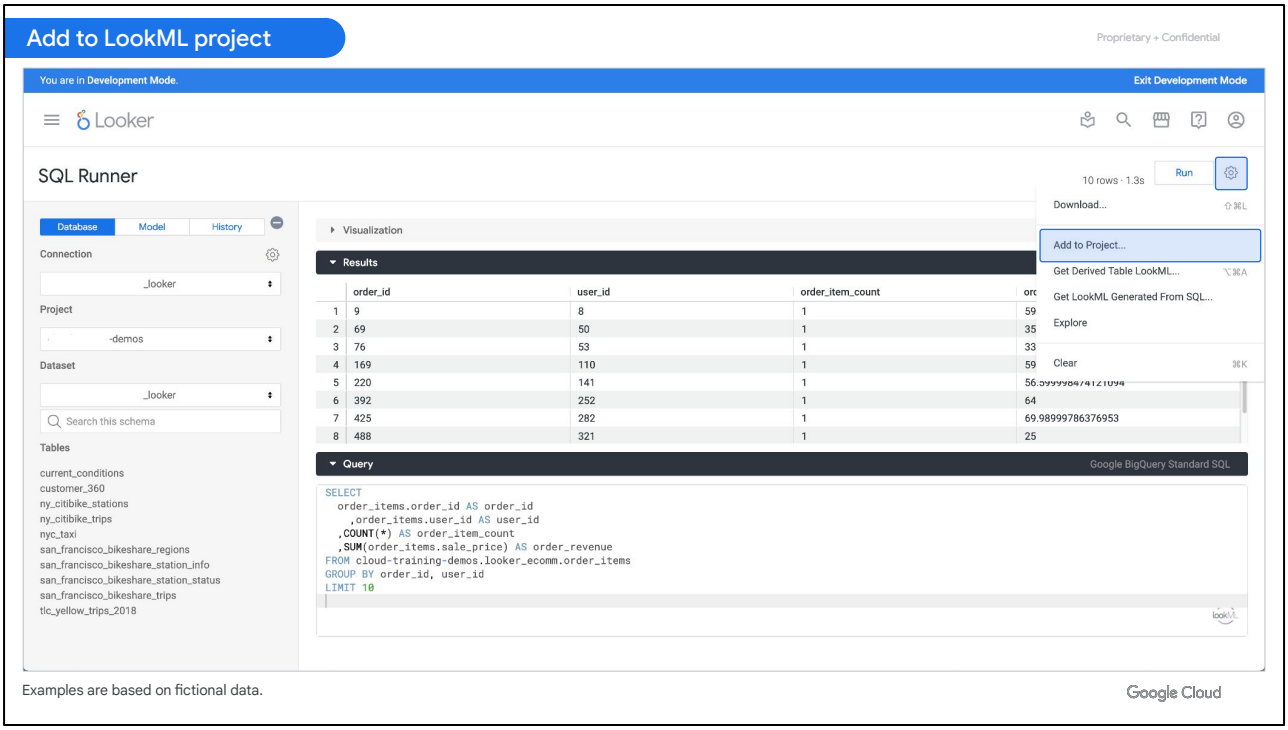
The **GROUP BY** clause is used to group the results by **order_id** and **user_id**, and the **LIMIT** clause is used to limit the results returned, as you only need to review a subset of records to ensure that the query is working successfully.

```
SELECT
      order_items.order_id AS order_id
      ,order_items.user_id AS user_id
      ,COUNT(*) AS order_item_count
      ,SUM(order_items.sale_price) AS order_revenue
FROM cloud-training-demos.looker_ecomm.order_items
GROUP BY order_id, user_id
LIMIT 10
```

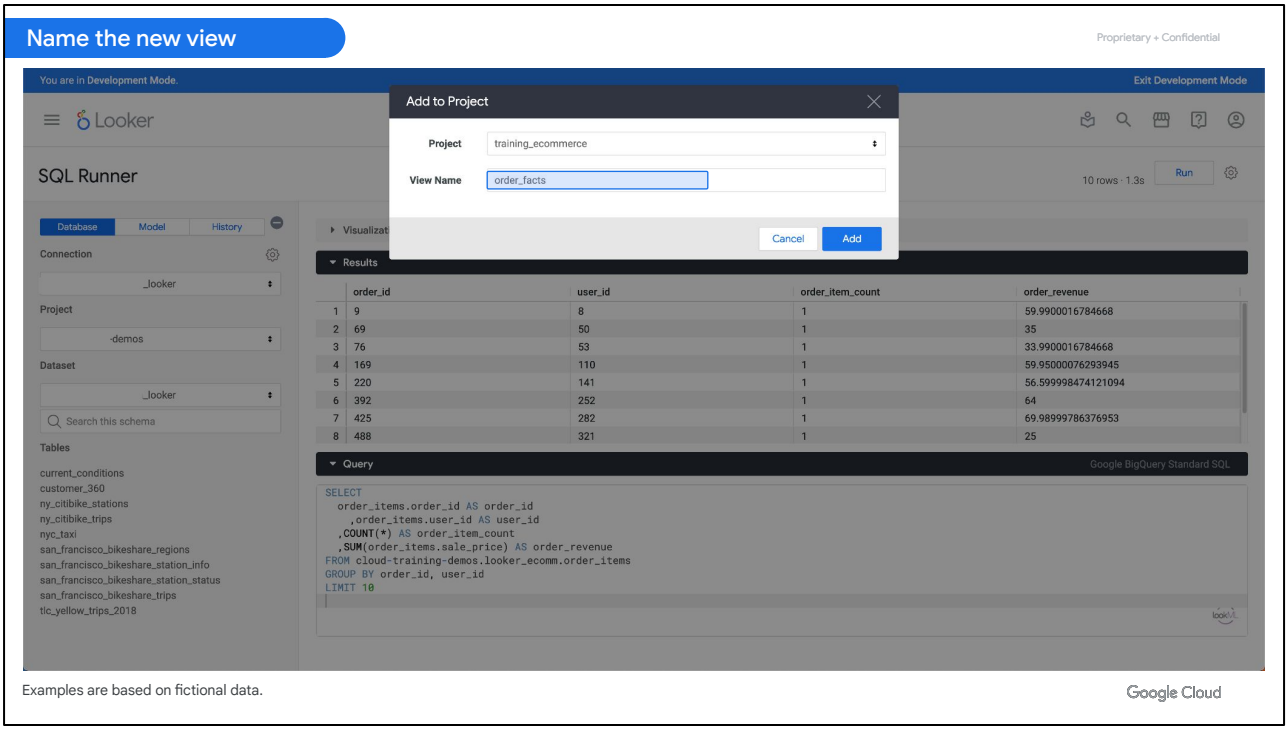


Next, take a moment to review the results of the query to ensure that your syntax is valid and returns the desired results.

In this example, the query is indeed returning the order ID, user ID, and number of items associated with each order as well as the total revenue from each order.

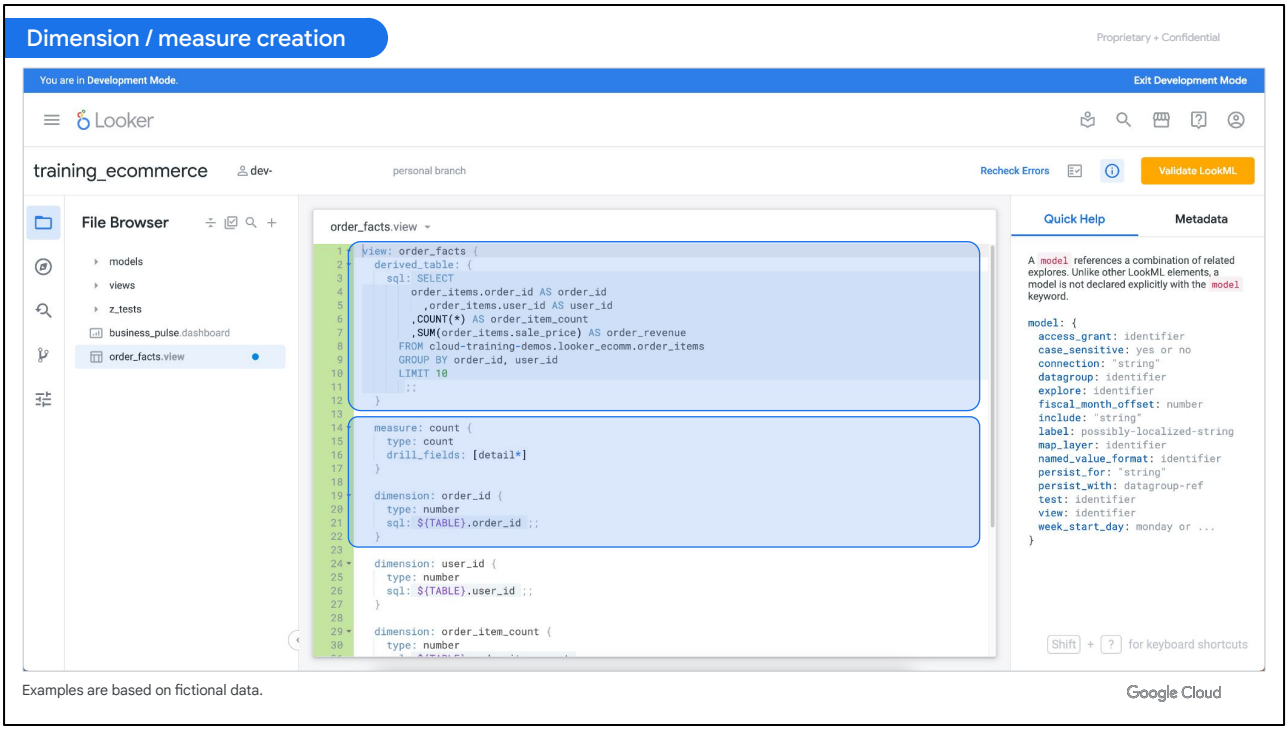


After confirming that the query returned the desired results, click on the gear icon in the top right corner and choose **Add to Project.**



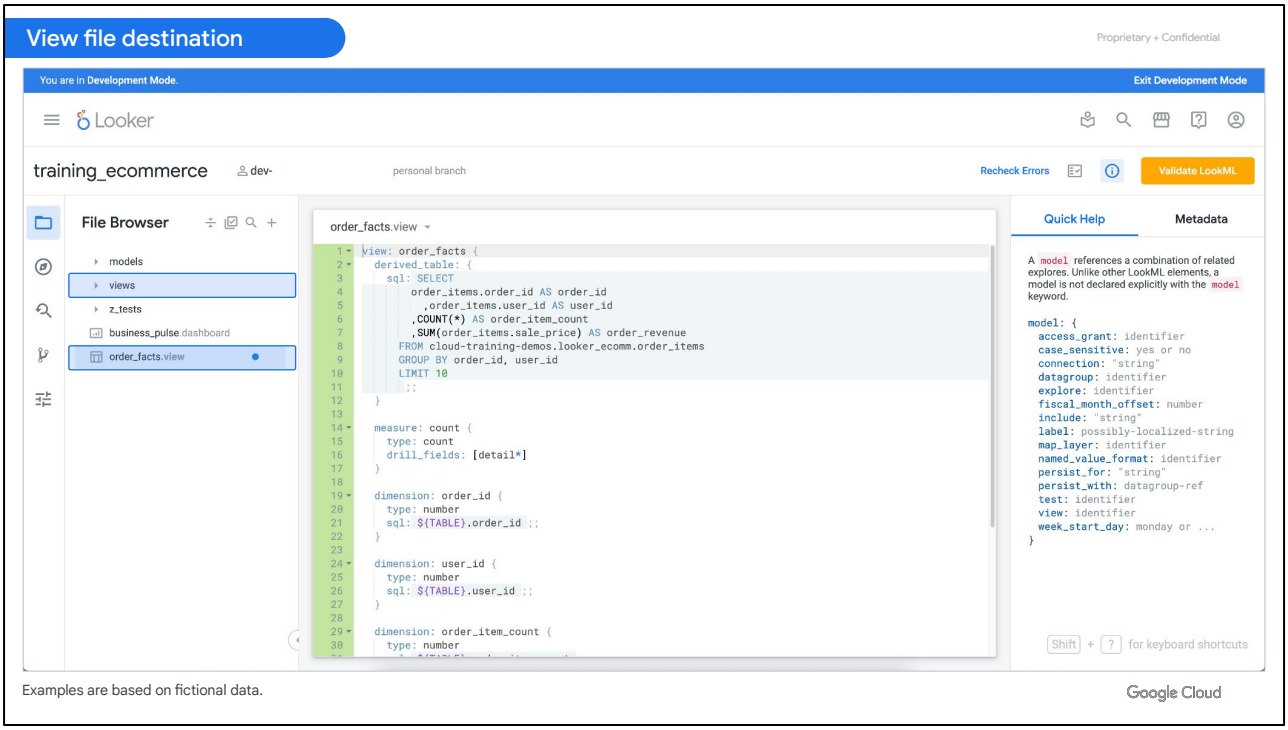
Select the project name **training_ecommerce** from the dropdown, and give your derived table a descriptive name such as **order_facts**.

Then, click **Add**.

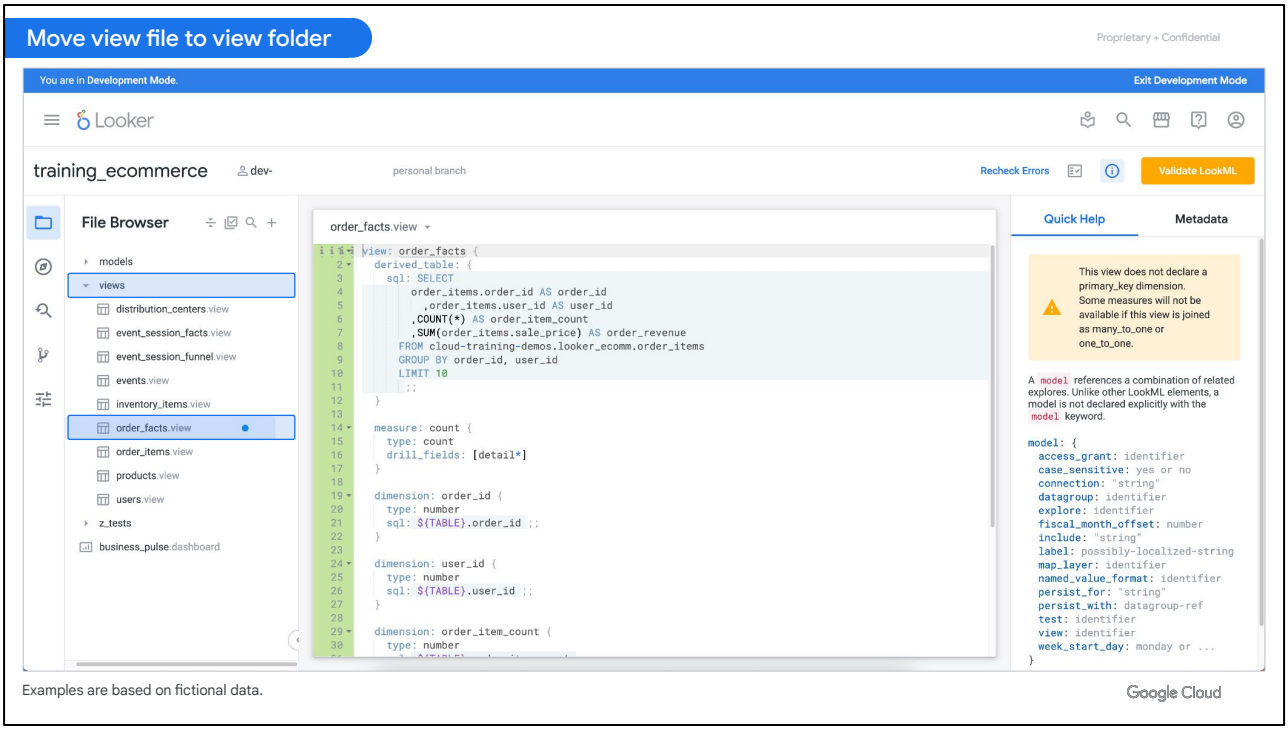


You will be redirected to the Looker IDE to review the newly created view file for your derived table.

Notice that Looker auto-generates a dimension for each column in the **SELECT** clause of the SQL query as well as a default count measure.

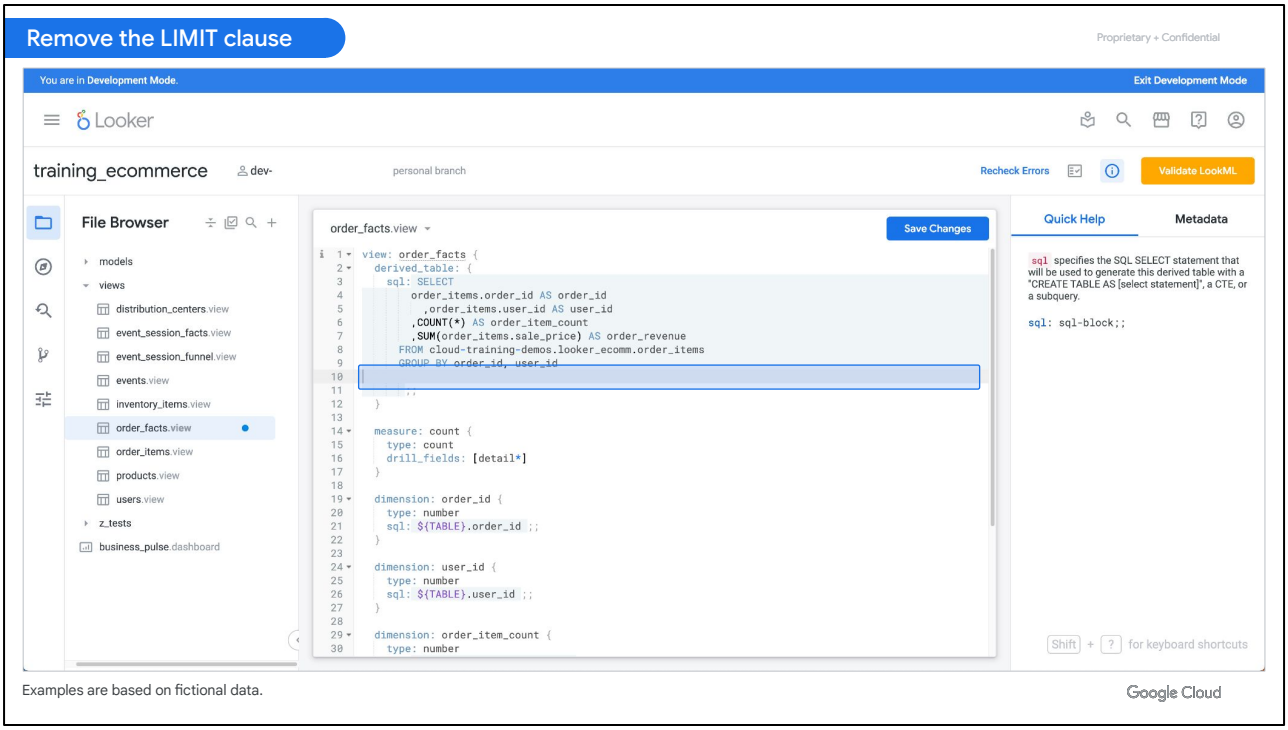


Also notice that the new view file for the **order_facts** view has been created outside of the views folder.



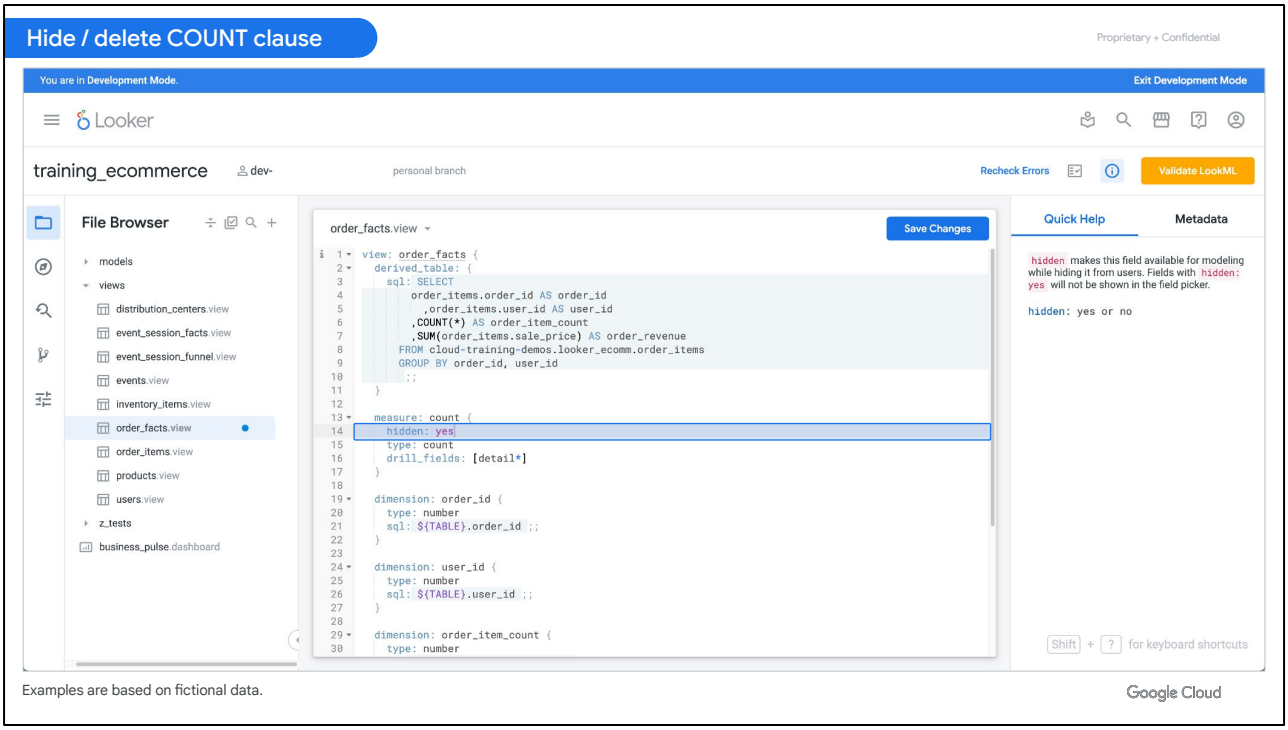
It is a best practice in Looker to keep the view files organized in the views folder of a LookML project.

To move the **order_facts.view** file, simply click on the **order_facts.view** file in the file browser, and drag the file under the folder named **views**.



Another best practice is to remove the **`LIMIT`** clause that you used for testing, since you do not actually want to limit the rows accessible by business users.

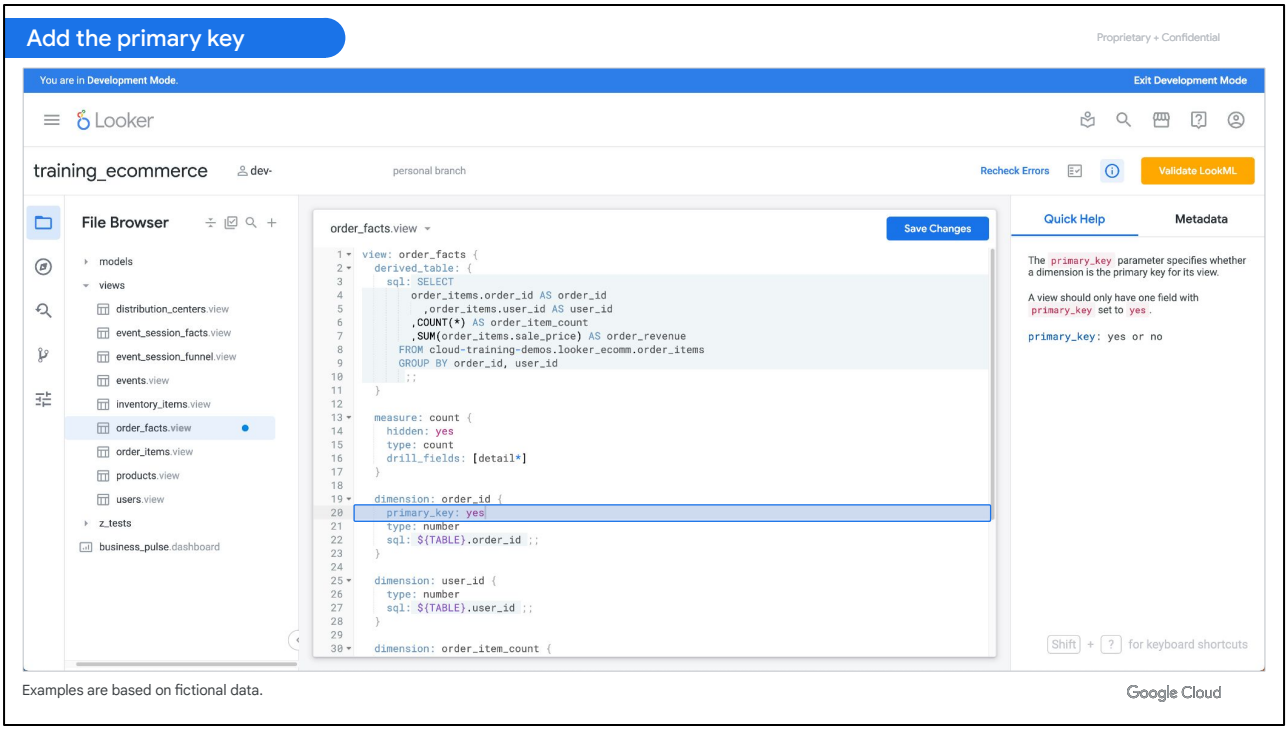
To remove the limit, you can simply delete the line for **`LIMIT 10`** from the **`sql`** parameter of the derived table.



As highlighted previously, Looker auto-generates a **count** measure along with the dimensions used in the derived table.

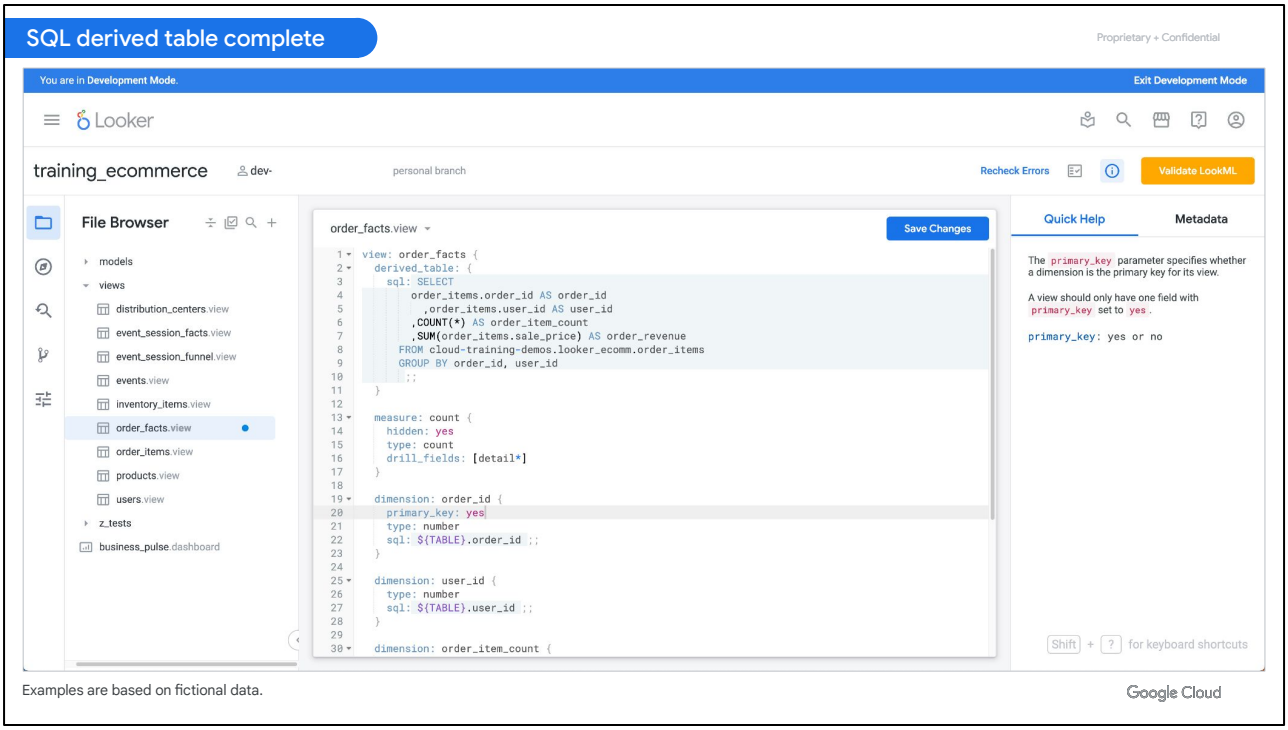
Sometimes this auto-generated **count** measure is not valuable, if you already have a **count** in another view that provides the same number. In this example, the auto-generated **count** measure is counting the order IDs, and there is already a count of orders in the **order_items** view.

So you can either delete the measure, or hide it using the **`hidden: yes`** parameter. Hiding the measure is a good idea if you would like to retain it for validation, but do not want to expose it to users.



A final best practice is to ensure that the new view has a primary key.

In this example, you can add the **primary_key: yes** parameter to the **order_id** dimension, which is the core organizing ID of this view that provides details about each individual order.



And with that, the new SQL derived table called **order_facts** is now ready for you to create new dimensions and measures, join it to the explore in the model file, and/or finish out the Git workflow to send your changes to production.

Pretty easy, right?

Proprietary + Confidential

Lesson summary

Move the view file to the **views** folder.	Remove unnecessary **`LIMIT`** clauses in the SQL.	Remove or hide unnecessary **count** measures.	Establish the **primary key**.

Google Cloud

In summary, let's review the best practices used to create the new SQL derived table.

After generating the new view file from the query in SQL Runner, you moved the new view file for **order_facts** to the folder named **views**, where all of the other view files are stored.

Next, you removed the **LIMIT clause** that was only needed for testing, and then hid the auto-generated count measure for orders because it already exists in the **order_items** view.

Last, you established **order_id** as the primary key for the **order_facts** view.

With these best practices, you are now ready to create new SQL derived tables in your organization's Looker instance.

Proprietary + Confidential

Derived Tables

01 Introduction to Derived Tables

02 Types of Derived Tables

03 Example 5: Creating SQL Derived Tables with LookML

04 Example 6: Creating Native Derived Tables with LookML

05 Useful Parameters for Native Derived Tables

06 Persistent Derived Tables

07 Lab 2: Creating Derived Tables with LookML



Google Cloud

In Looker, you can create custom tables using either SQL derived tables, which depend on manually written SQL queries, or native derived tables, which are written entirely in LookML.

Because native derived tables reuse existing LookML objects to define new custom tables, they promote code reusability and reduce hard-coded references to your underlying database tables.

The easiest way to create a native derived table in Looker is through an Explore.

Proprietary + Confidential

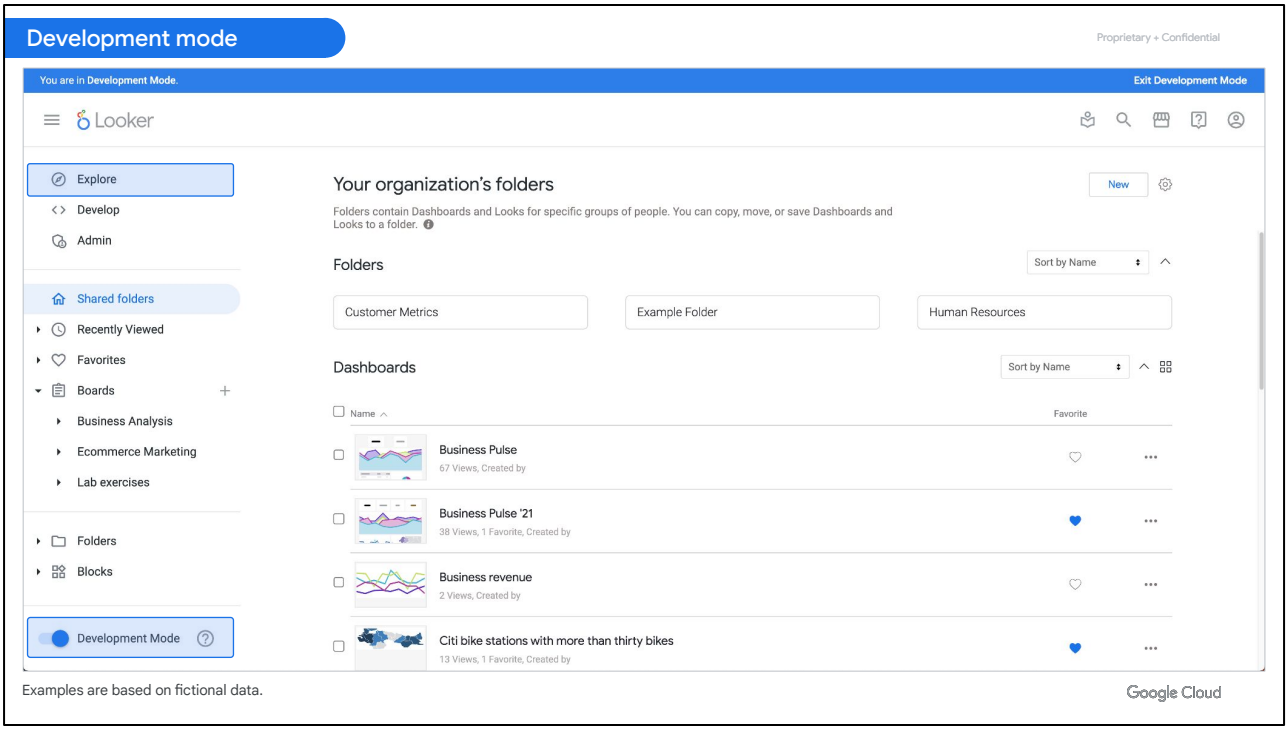
Example: Creating native derived tables with LookML

Create a native derived table that contains the details for each order: order ID, user ID, number of items, and total cost of the items ordered.



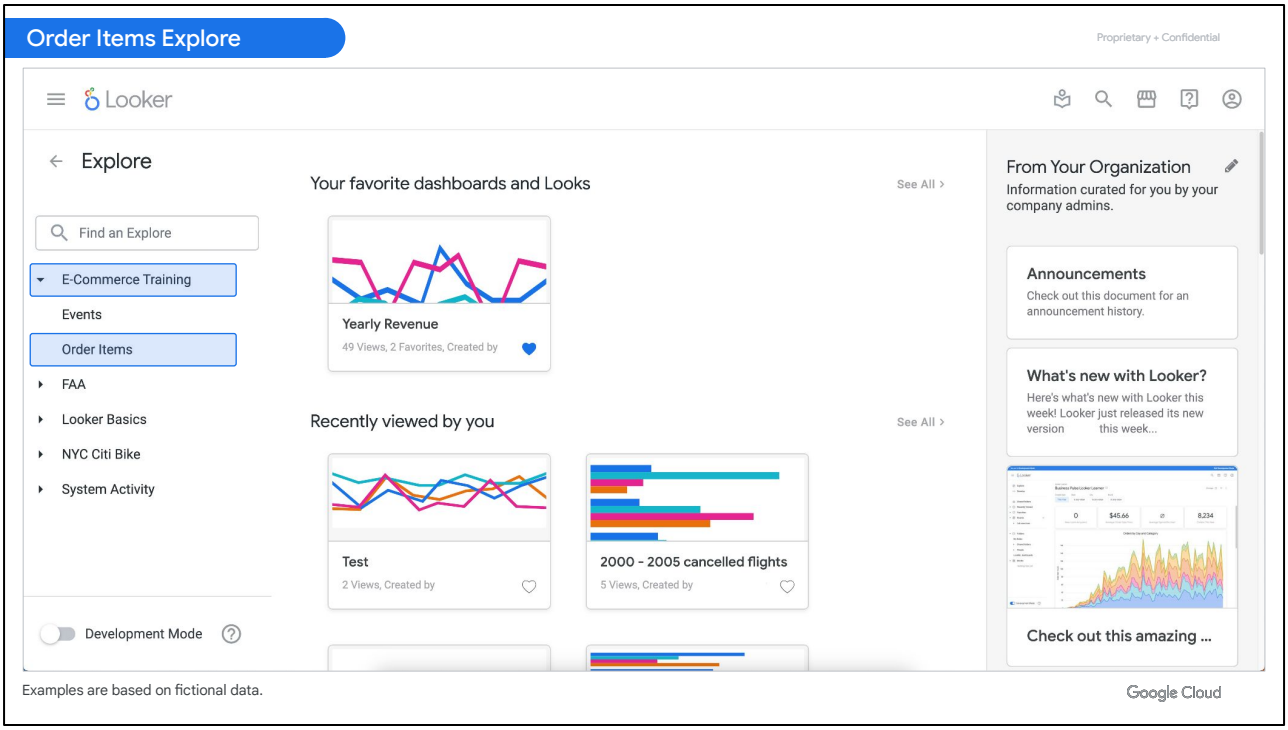
Google Cloud

Let's walk through an example of using the **Order Items** Explore to create a native derived table that summarizes the following details for each order: order ID, user ID, the number of items, and the total cost of items ordered.

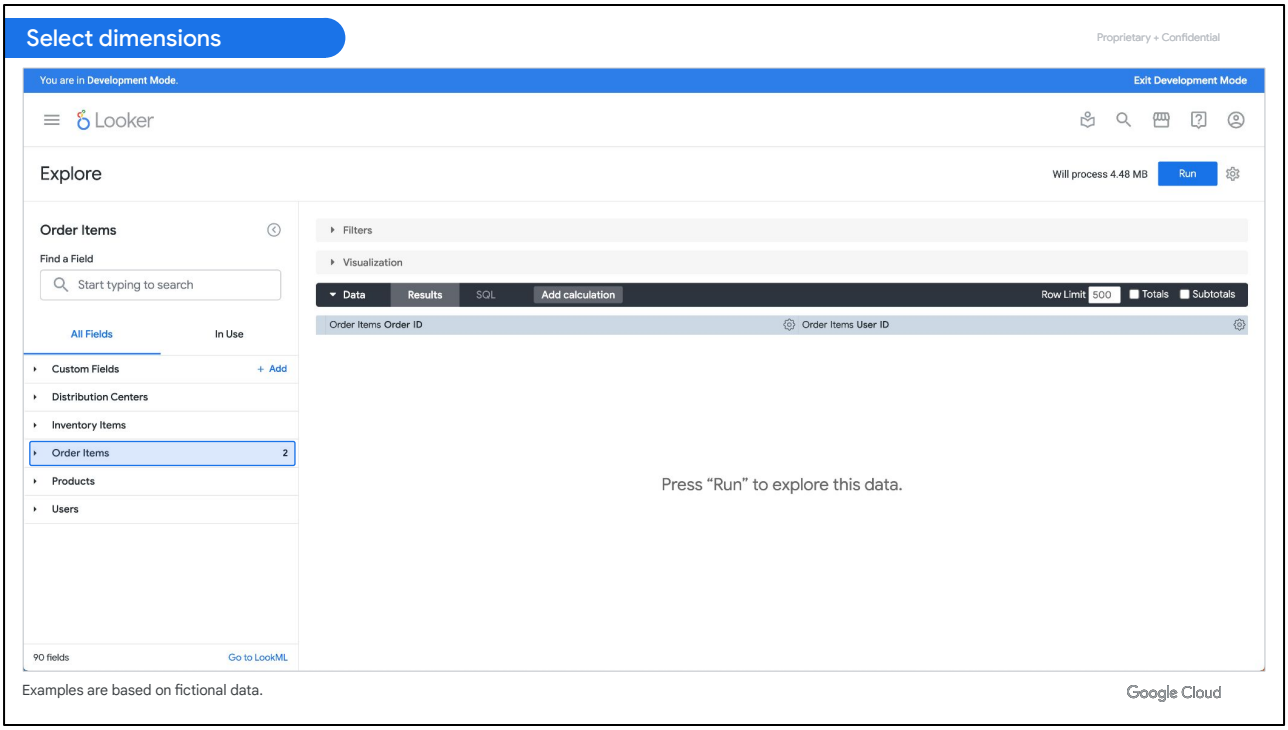


Let's dive right in! Begin by ensuring that you are in Development Mode. You can enable Development Mode by toggling the button for Development Mode on the bottom left corner of the Looker home page.

Then, click **Explore** on the left side navigation menu to see the develop options.

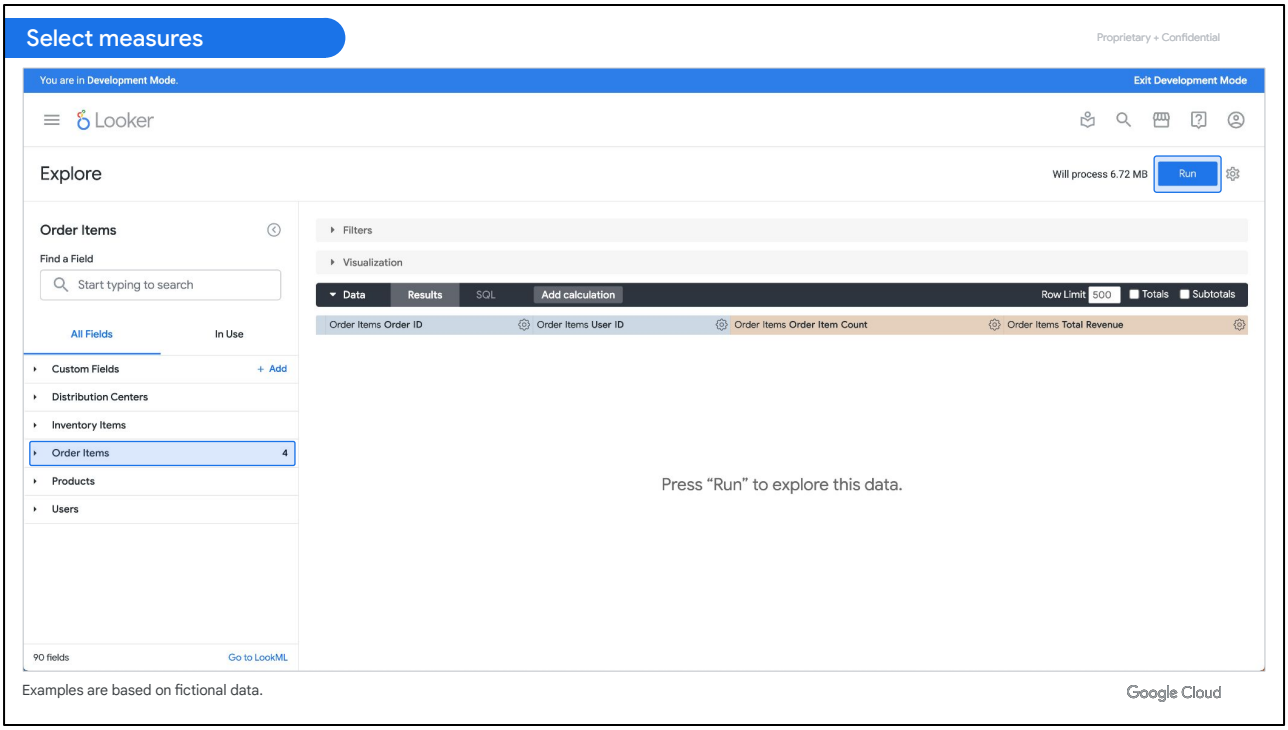


Under the **Explore** options, click on **Order Items** under the **E-Commerce Training** heading.



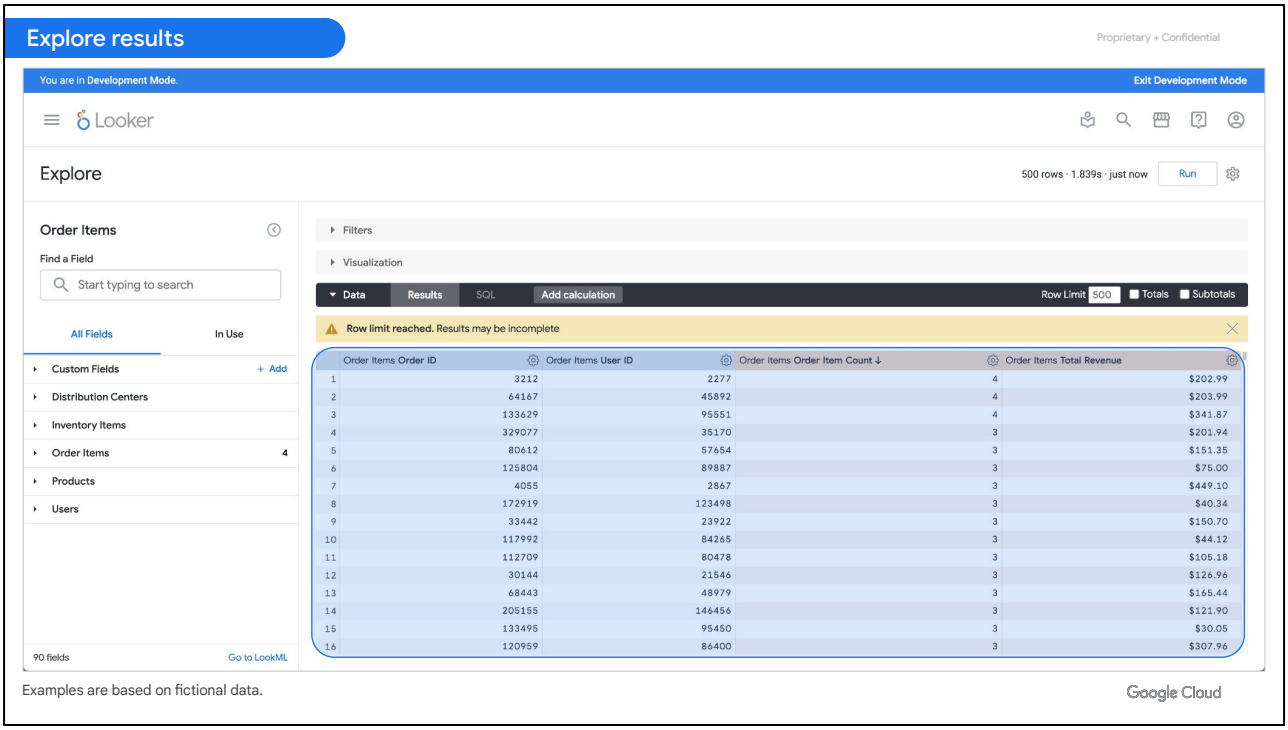
In the **Order Items** Explore, select the dimensions and measures you need for your native derived table. Remember that the Explore will automatically generate a valid, performant SQL query for you.

For this example, you want to first select the **Order ID** and **User ID** dimensions from the **Order Items** view**.**



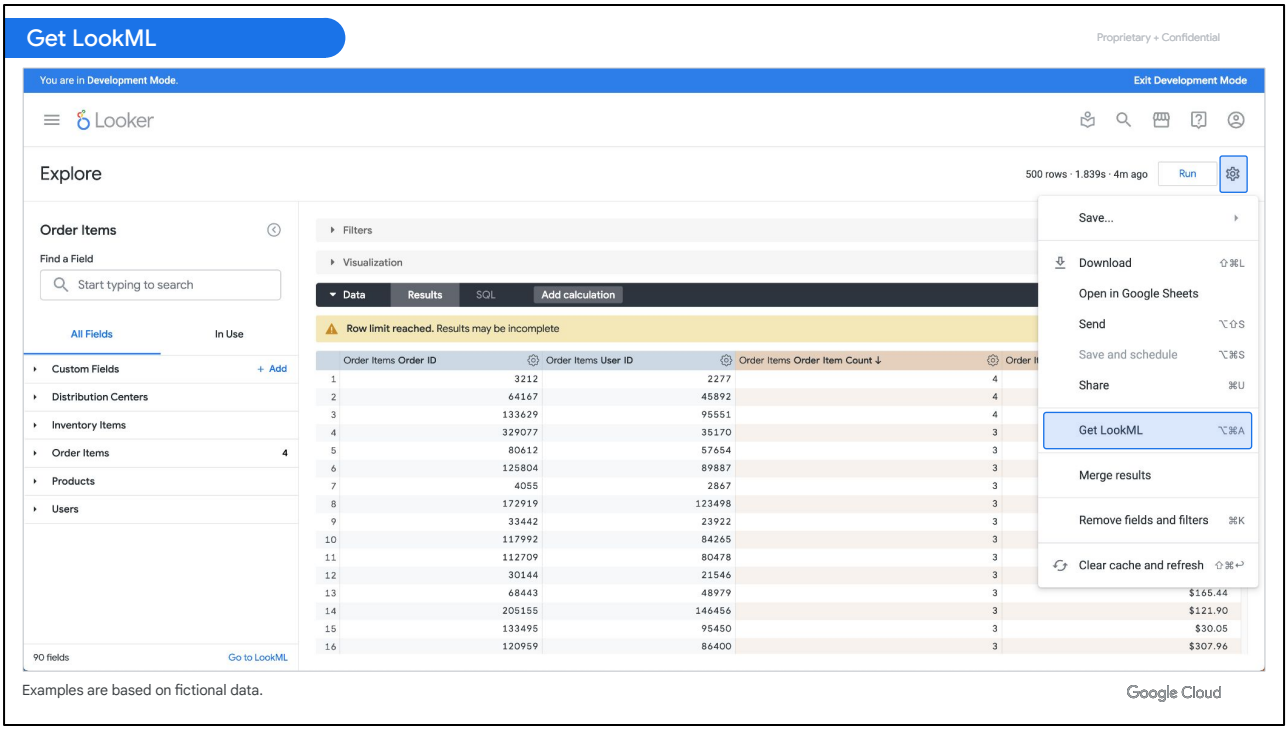
Then, also within the **Order Items** view, select the **Order Item Count** and **Total Revenue** measures.

Click the **Run** button to review the results.

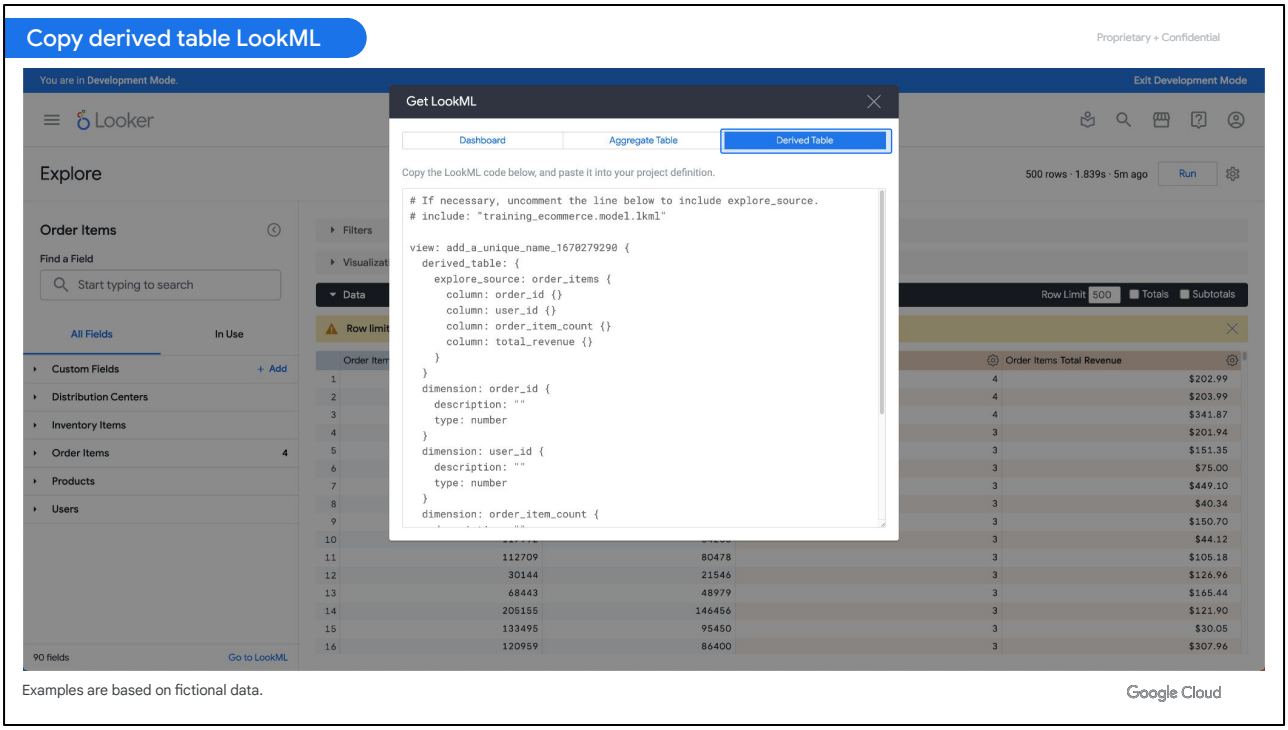


Now, take a moment to review the results of the query to ensure that it is returning the desired results.

For this example, the request correctly returns the **Order ID**, **User ID**, number of items for each order in **Order Item Count**, and the total price of each order in **Total Revenue**.

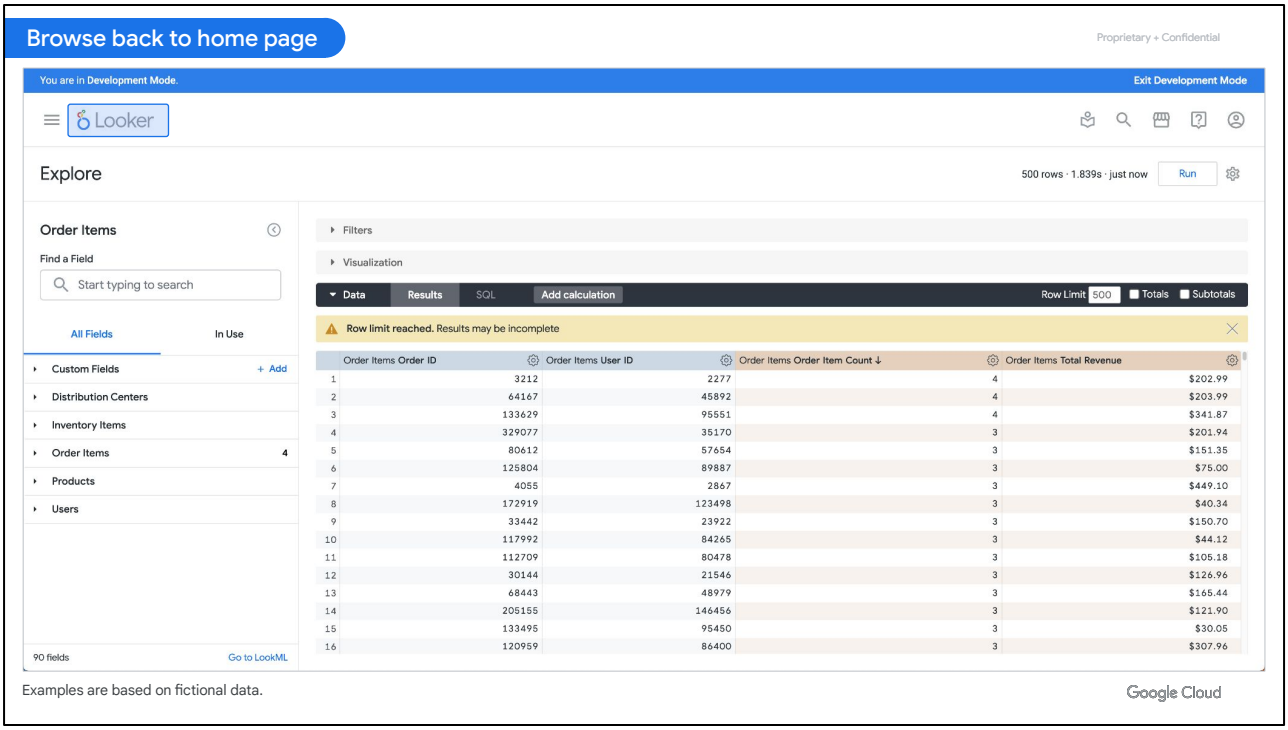


After you have confirmed the request returns the desired results, click on the gear icon in the top right corner and select **Get LookML…**



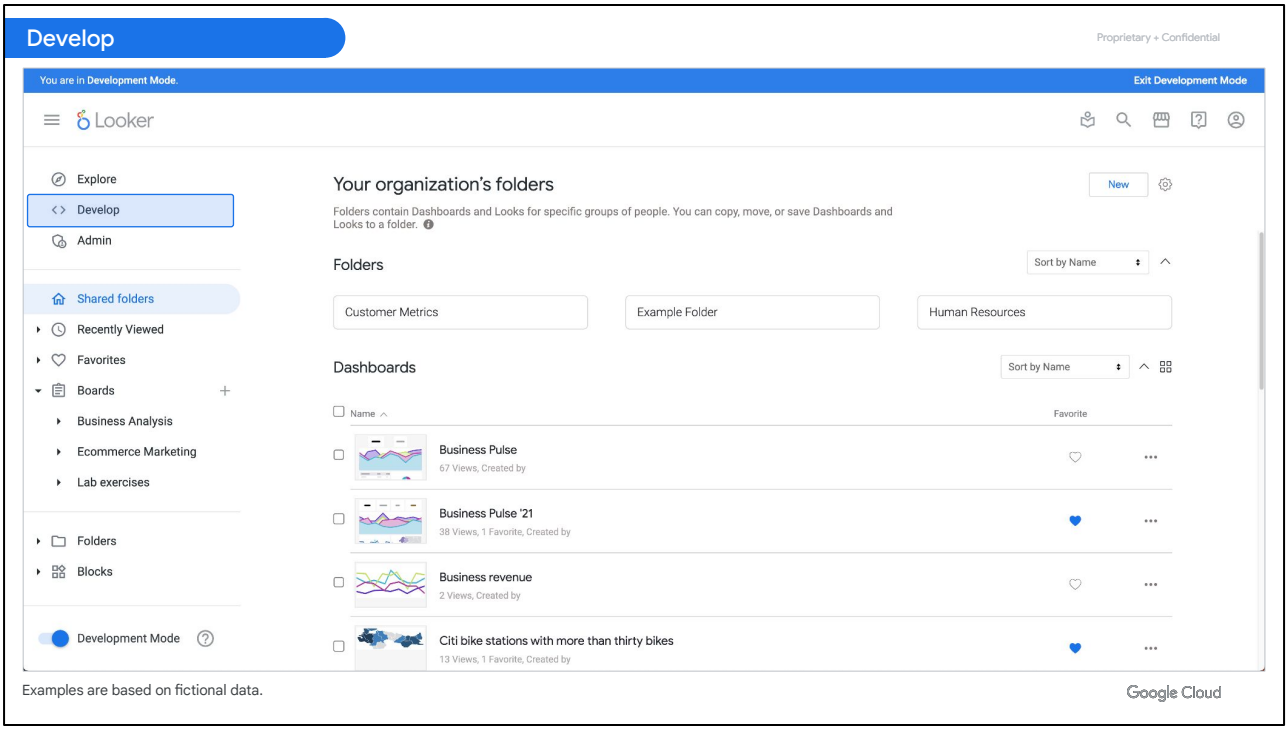
Within the popover window for **Get LookML**, click on the **Derived Table** tab, and copy the LookML code to your computer clipboard.

In the next steps, you will paste this LookML code into a new view file for this native derived table.

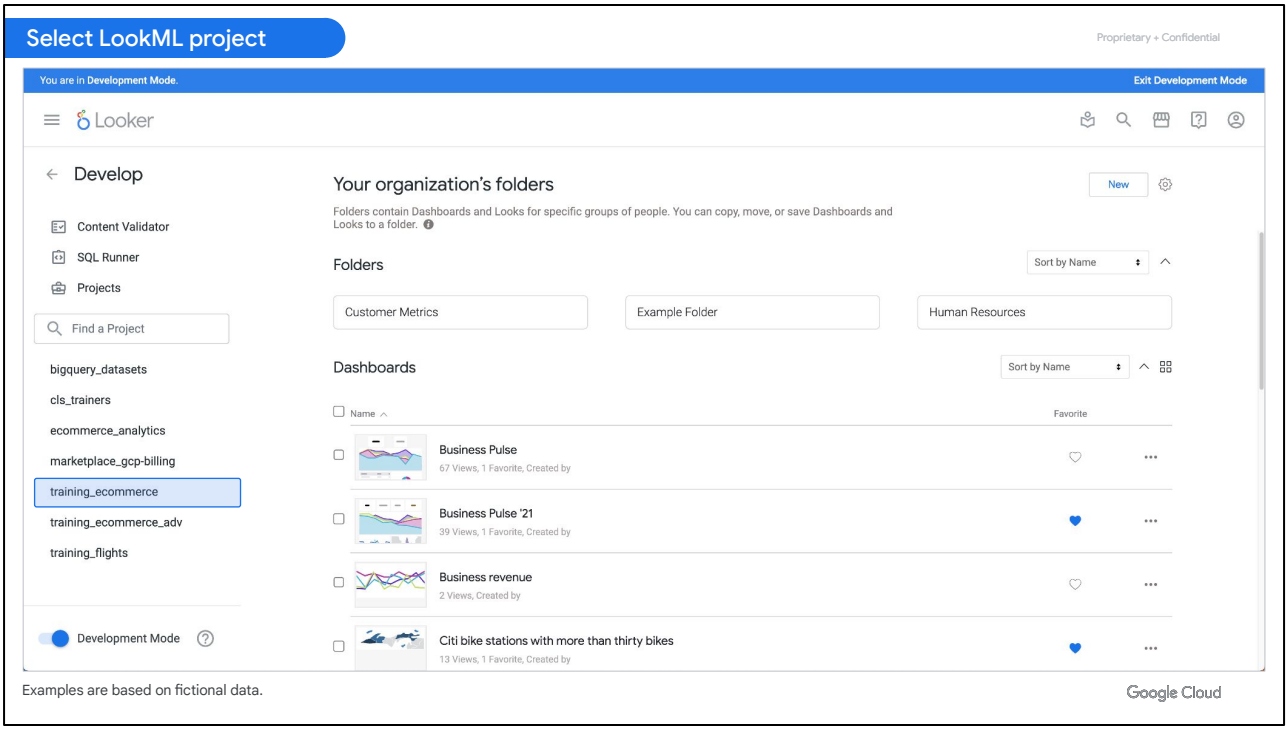


Now, you need to create a new view file to contain the LookML code that you just copied.

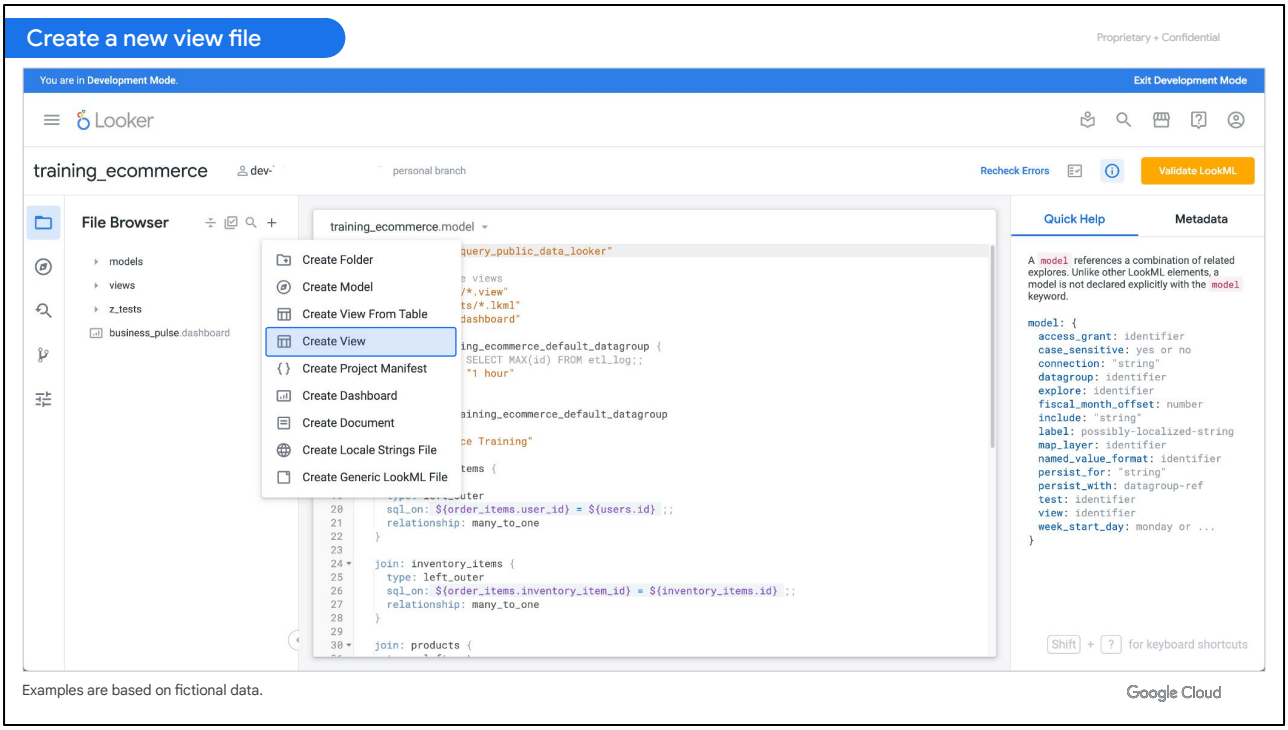
To get to the Looker IDE where you can create new view files, return to the Looker home page by clicking **Looker** on the top left-side of the user interface



Then, click **Develop** to see the options.



Now, click on the **training_ecommerce** project to open the Looker IDE in this project.

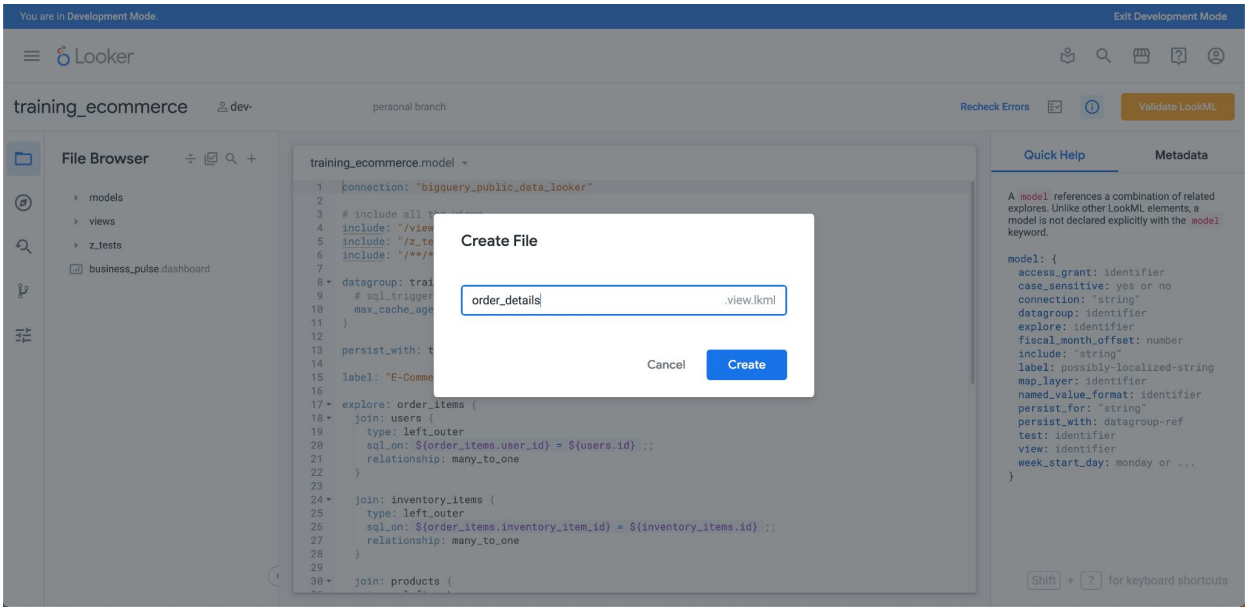


In Looker, it is a best practice to always create a new view for a new native derived table.

To create a new view file in the file browser, click on the plus (**+**) icon, and choose **Create View**.

Name the new view file

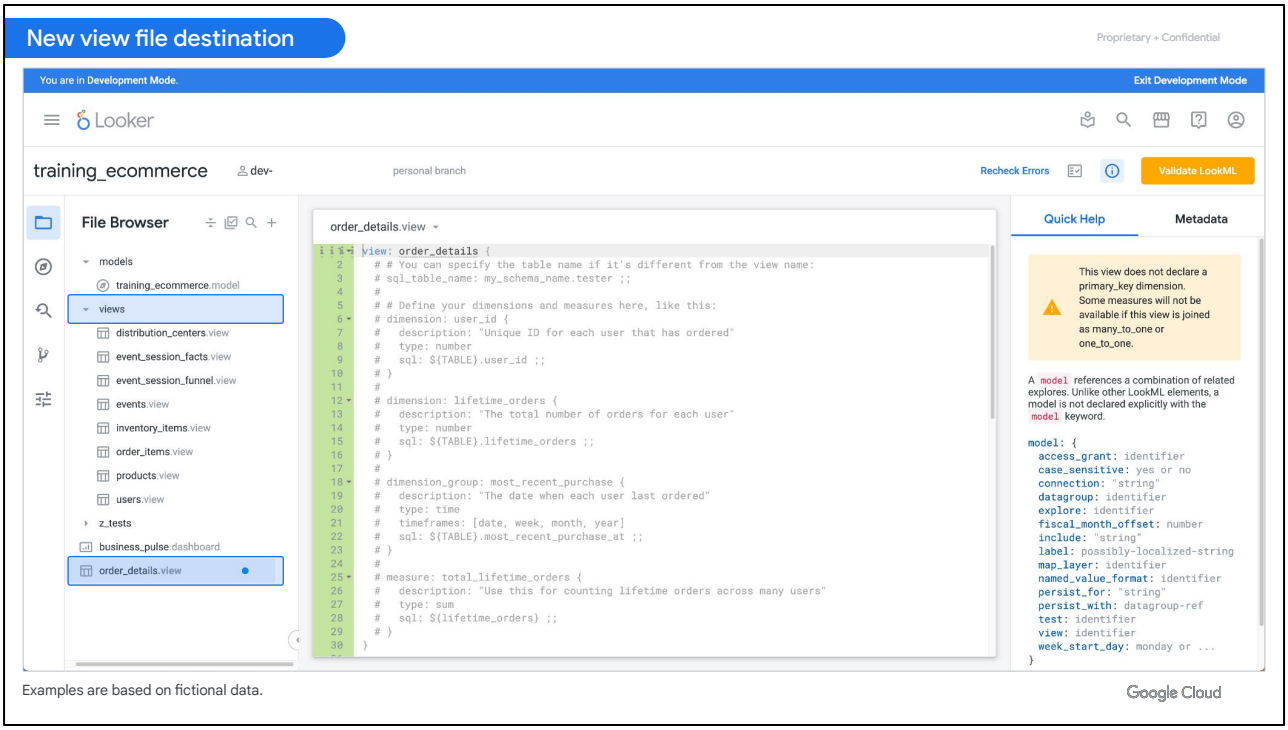
Proprietary + Confidential



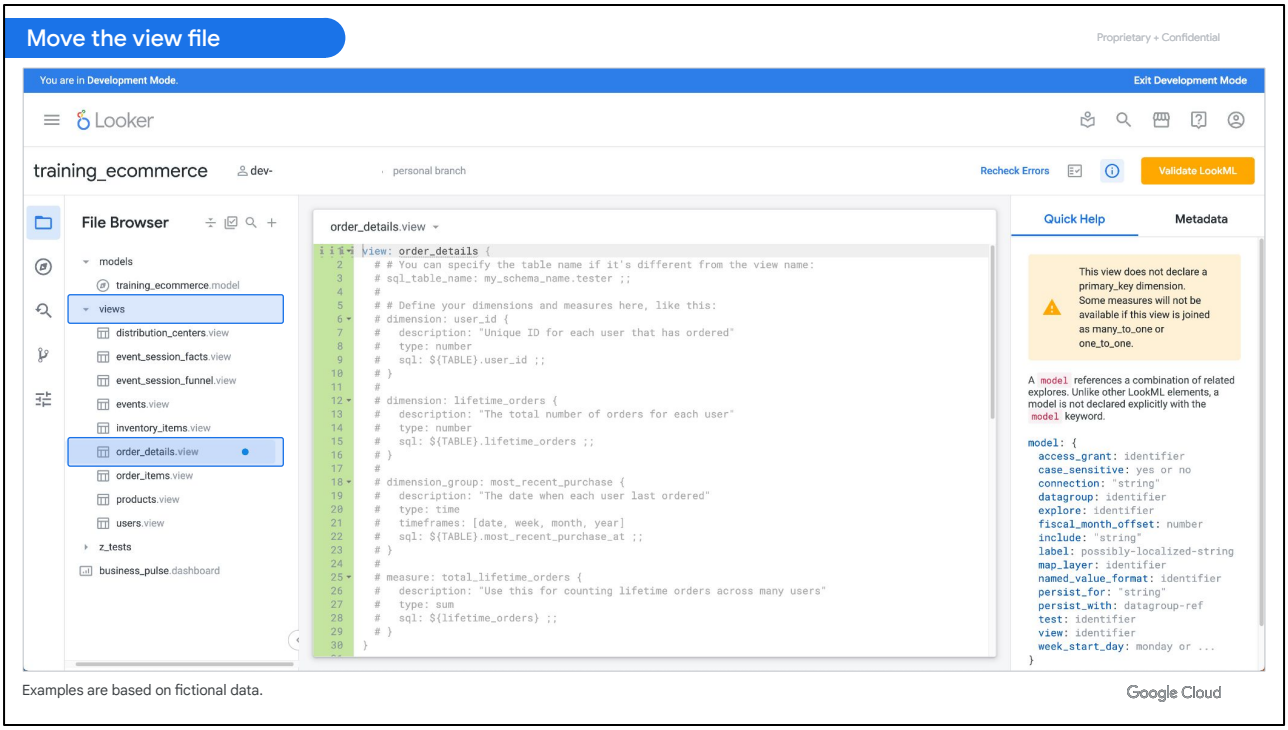
Examples are based on fictional data.

Google Cloud

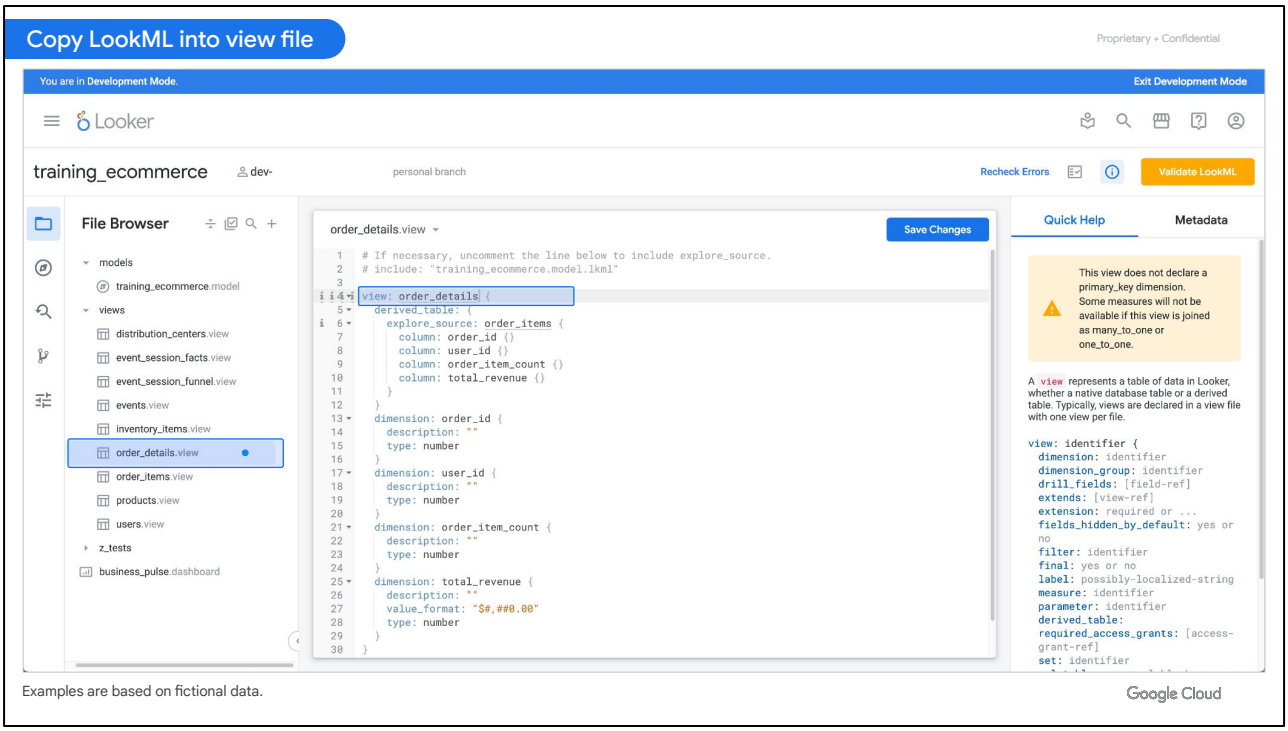
Name this new view file **order_details**. Then, click **Create**.



Notice that the **order_details.view** file is automatically created outside of the **views** folder in the file browser.

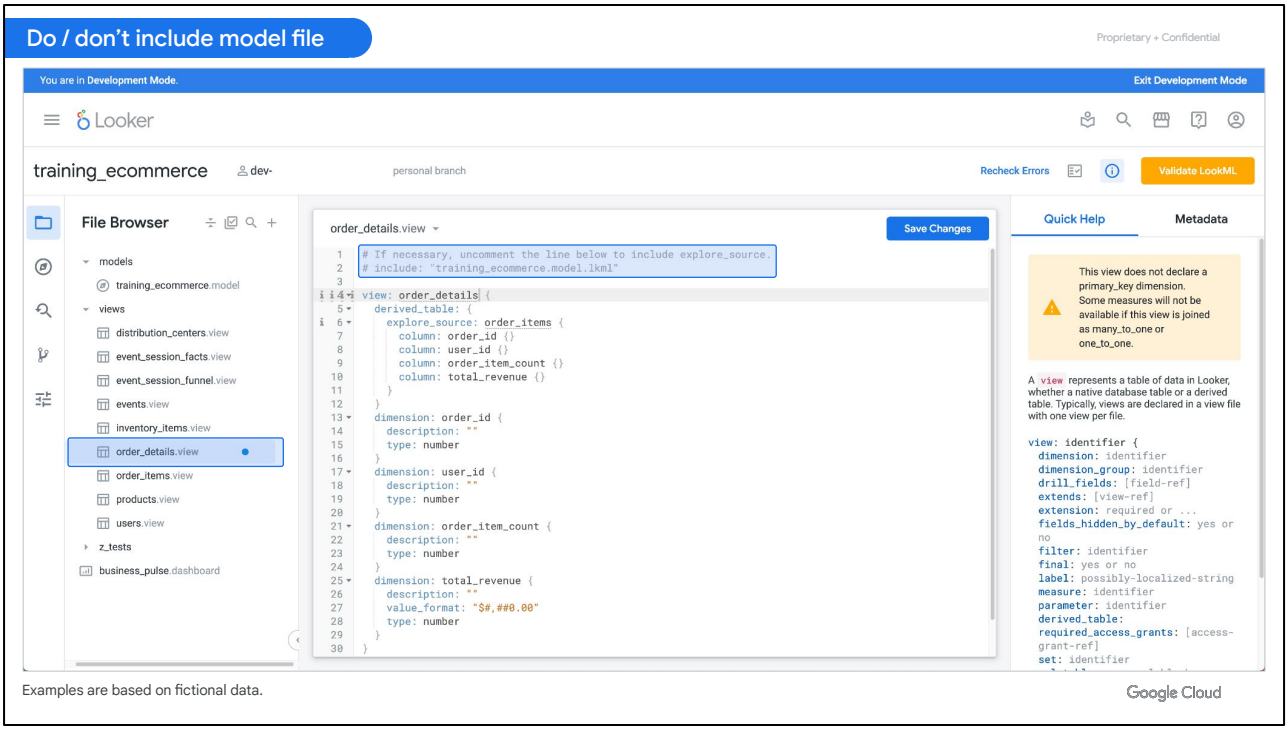


To follow the best practice of storing all views together in a **views** directory, click on the **order_details.view** file in the file browser and drag it under **views**.



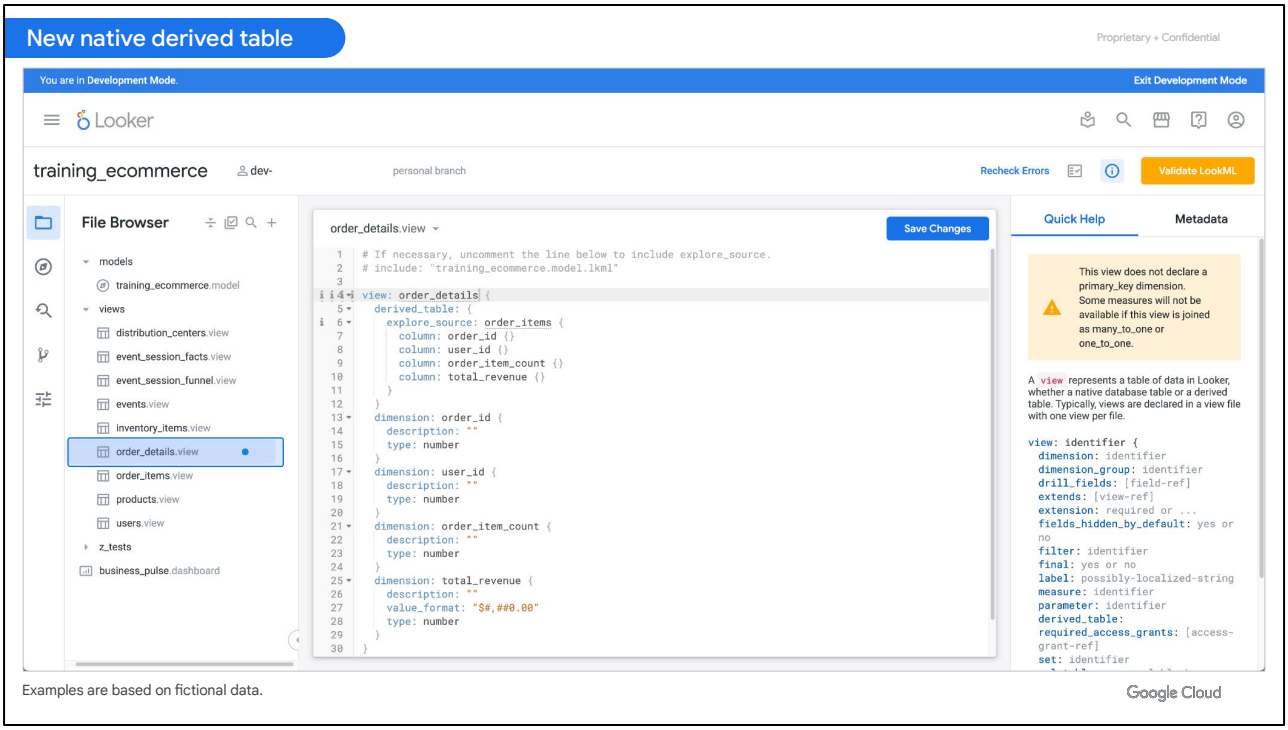
Now, replace the auto-generated LookML in the view file with your previously copied LookML code.

Following another best practice, be sure to change the auto-generated view name to **order_details** to match the view file name.



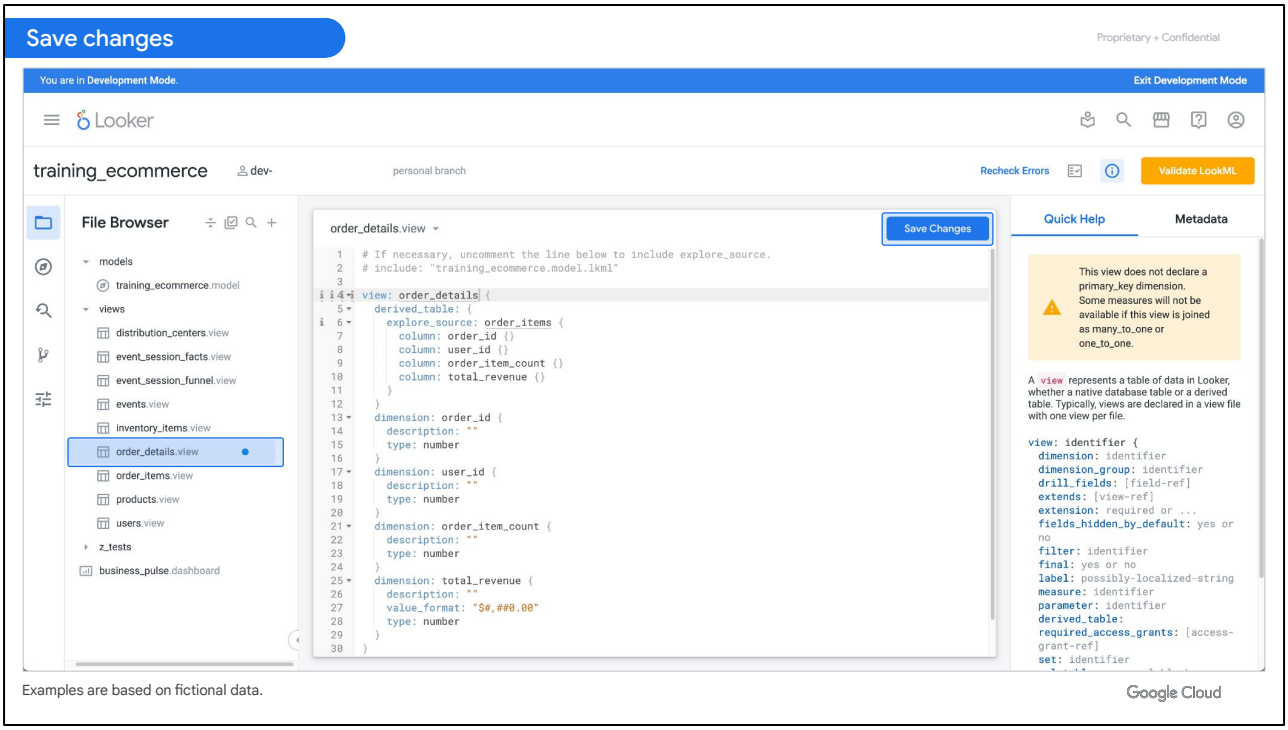
Notice that Looker makes a suggestion to include the model file but the lines are commented out. It is a best practice to leave the line for the model file commented out.

Otherwise, you risk creating circular dependencies in your model and causing validation errors because model files typically include many other files.

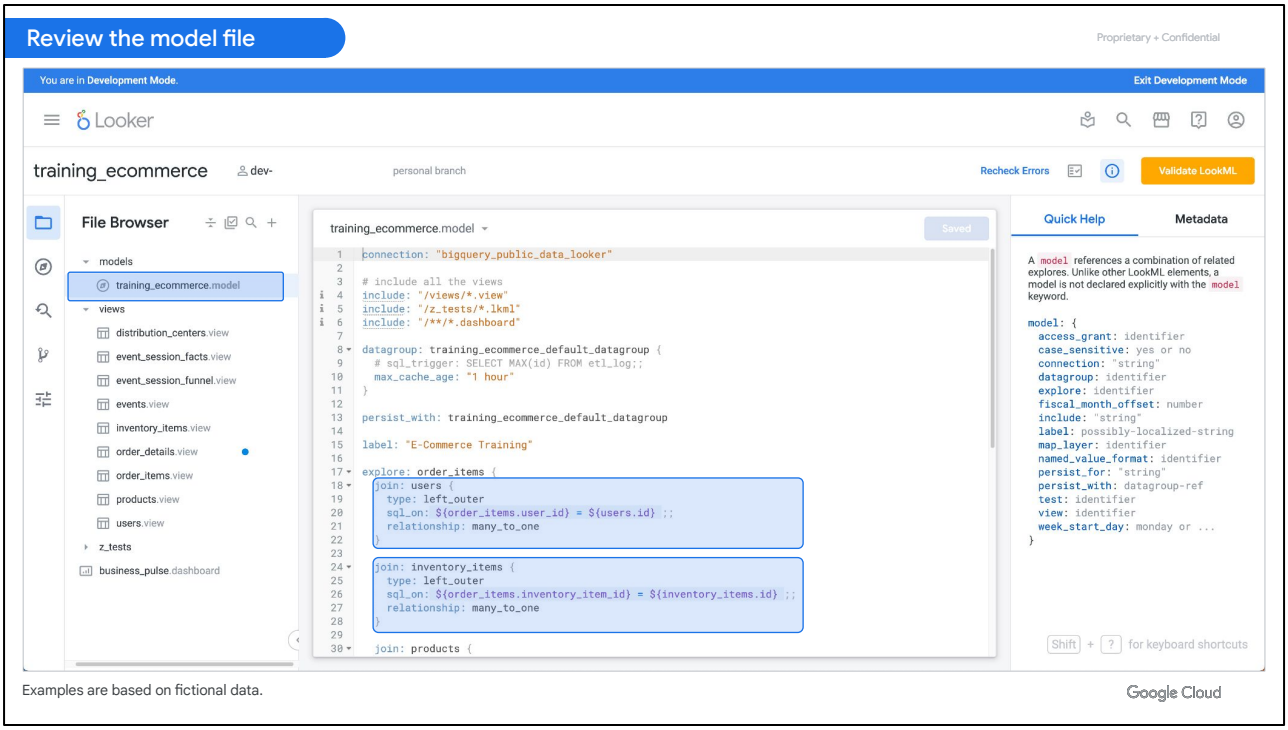


The new native derived table called **order_details** is now ready for you to create new dimensions and measures, join it to the explore in the model file, and/or finish out the Git workflow to send your changes to production.

For this example, you will join it to the **order_items** explore definition in the model file, and then use the **Order Items** Explore to review what business users would see if you pushed the changes to production.

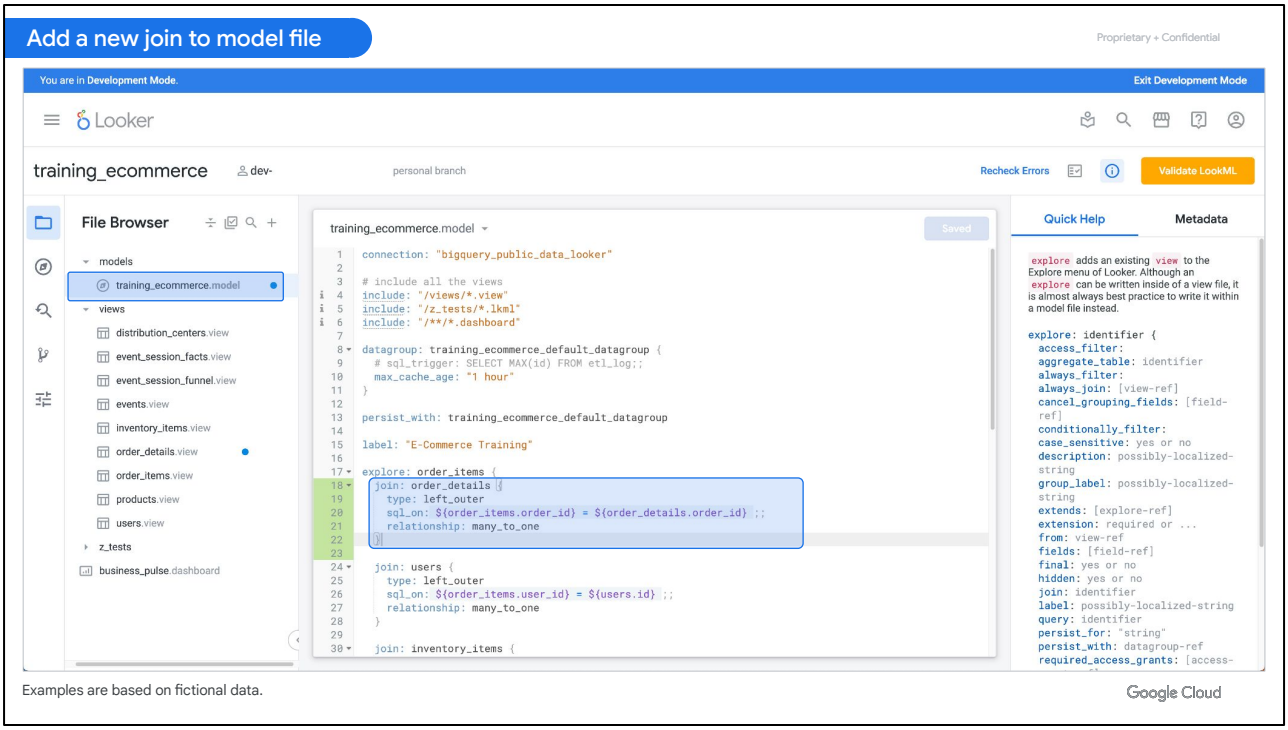


First, click **Save Changes** to save the updates to the **order_details** view file.



Then, click on the **training_ecommerce.model** file to add a new join to the **Order Items** Explore.

Locate the explore definition for **order_items**. Notice that there are several joins already defined such as the one for the **users** view.

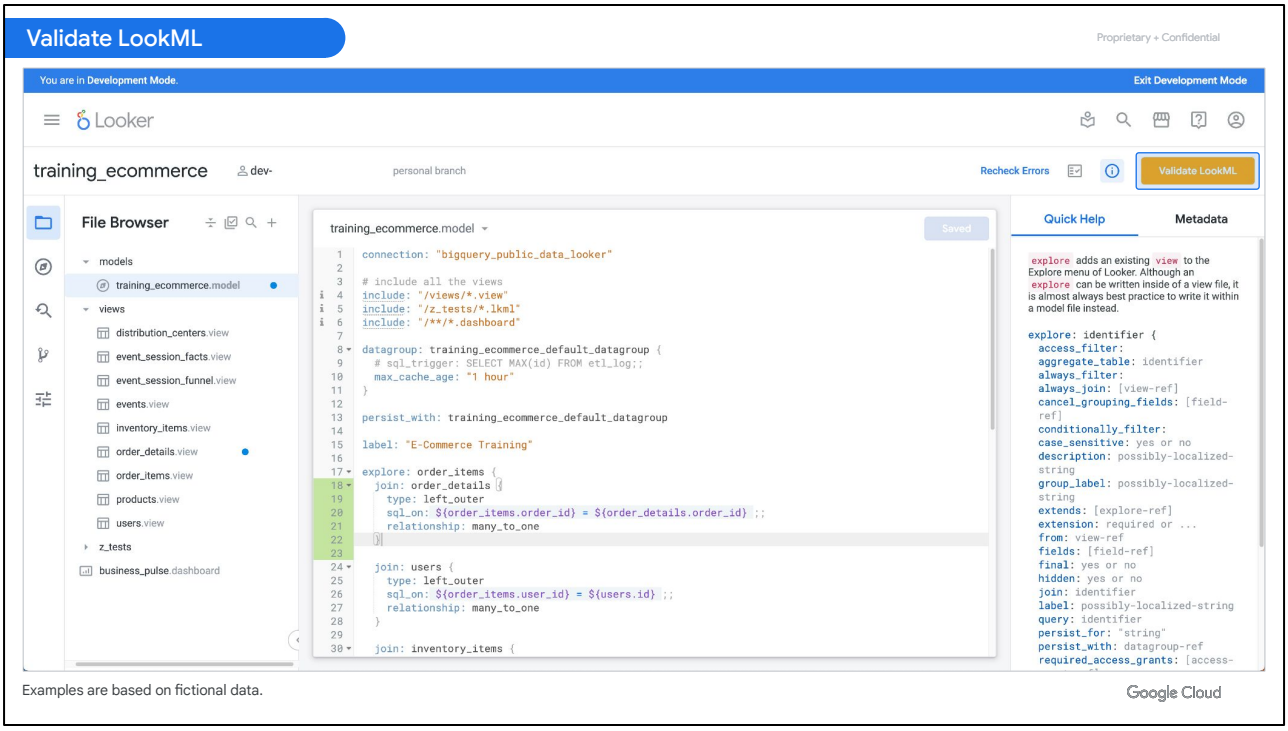


Above the existing join for **users**, add a new join for **order_details** by specifying:

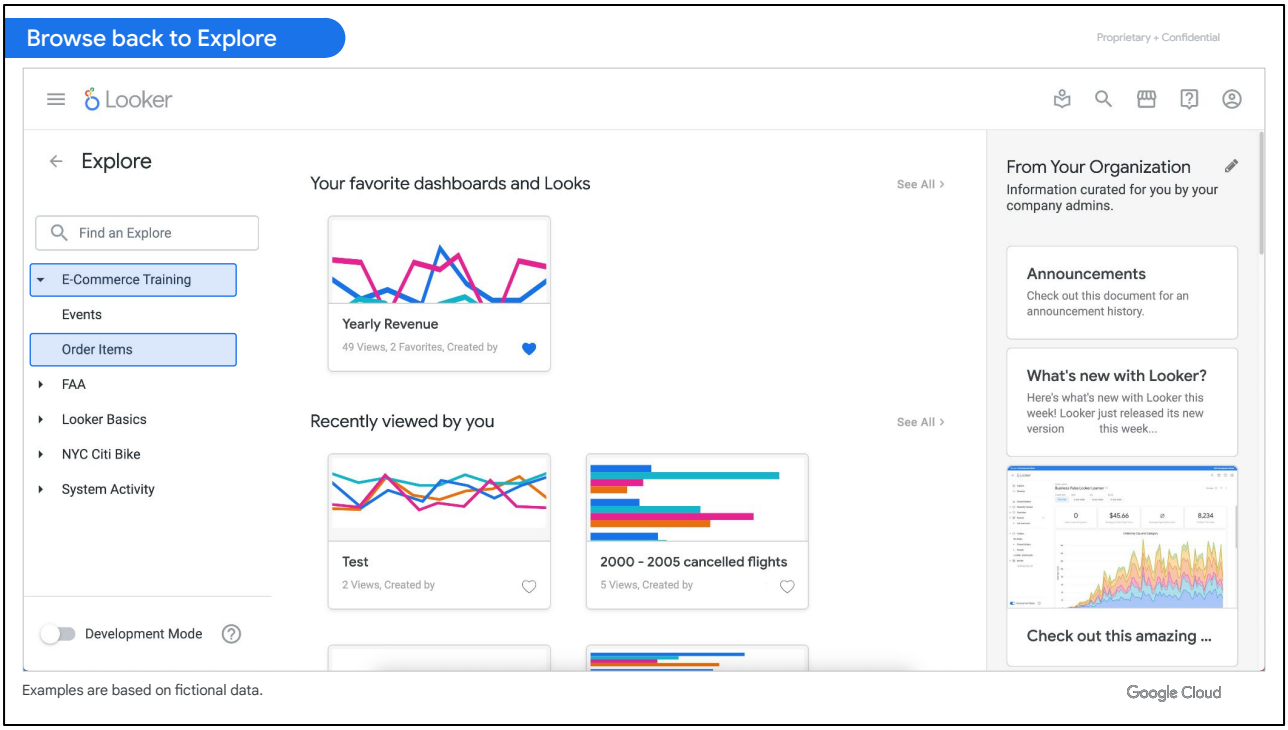
- **`join`** name as **order_details**
- **`type`** as **`left_outer`**
- **`sql_on`** as **${order_items.order_id} = ${order_details.order_id}**
- **`relationship`** as **`many_to_one`** (because **order_details** is organized as one summary row for each order)

Then, click **Save Changes.**

```
join: order_details {
    type: left_outer
    sql_on: ${order_items.order_id} =
${order_details.order_id};;
    relationship: many_to_one
  }
```

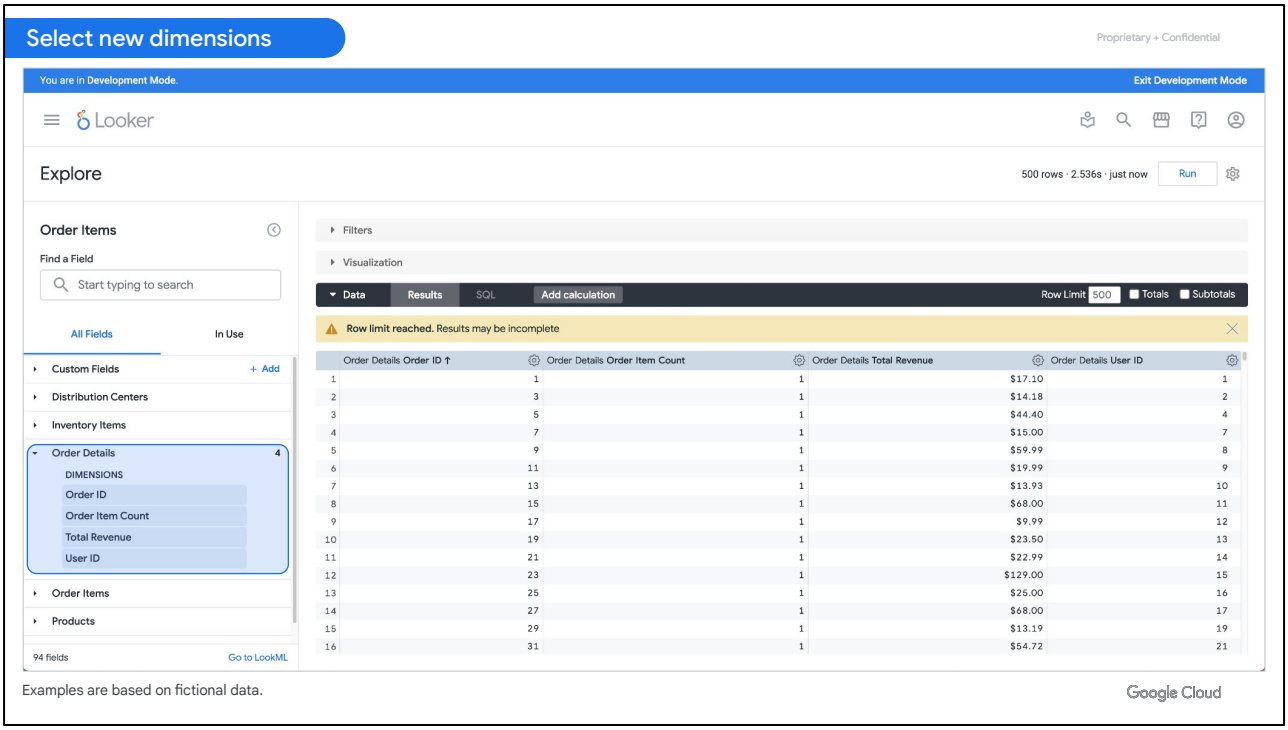


Next, click **Validate LookML** to check your new LookML.



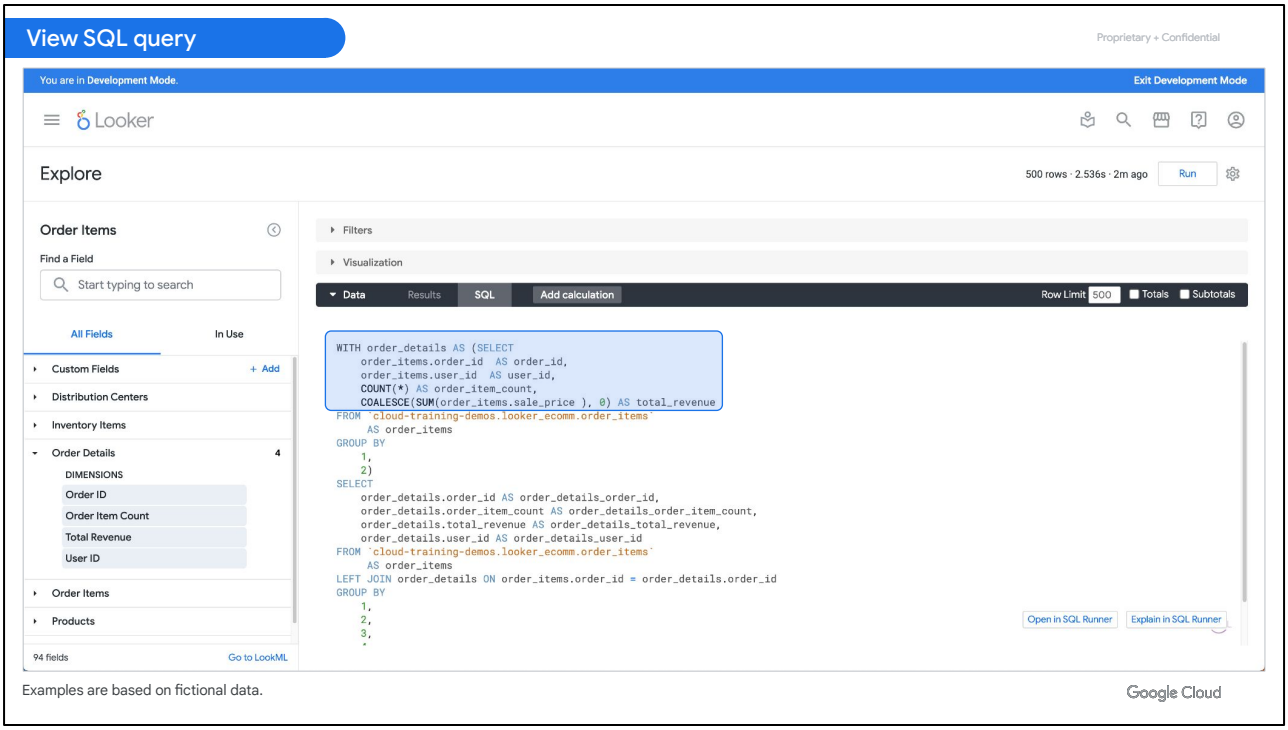
Now you can review the revised Explore to view and query the native derived table.

Within the Explore tab, click on **Order Items** under E-Commerce Training.



In the **Order Items** Explore, select the dimensions for **Order Details**: **Order ID, Order Item Count, Total Revenue,** and **User ID.**

Then, click **Run** to see the results.



Finally, click the **SQL tab** to view the generated SQL query.

Notice the common table expression (CTE) identified by the **WITH** clause. Though there are options in Looker to persist derived tables, this new derived table is considered ephemeral because it is generated at run-time.

Proprietary + Confidential

Lesson summary

Paste the generated LookML into a new **view file** within the appropriate project.	Move the view file to the **views** folder.	Change the auto-generated **view name** to match the **view file name**.	Avoid including the model file that contains the Explore being used.

Google Cloud

In summary, let's review the best practices used to create the new native derived table.

First, you created a new view file for the native derived table, and then moved the new view file to the folder named **views**.

Next, you pasted the copied LookML code into the new view file and replaced the auto-generated view name with the name of the view file. And last, you left the model file commented out in the view file.

Following these best practices, you can use native derived tables to create new custom tables in your organization's Looker instance.

Proprietary + Confidential

Derived Tables

01 Introduction to Derived Tables

02 Types of Derived Tables

03 Example 5: Creating SQL Derived Tables with LookML

04 Example 6: Creating Native Derived Tables with LookML

05 Useful Parameters for Native Derived Tables

06 Persistent Derived Tables

07 Lab 2: Creating Derived Tables with LookML



Google Cloud

In Looker, native derived tables allow you to create new custom tables that reuse existing objects in your project, such as dimensions and measures, and are expressed entirely using LookML.

To fully understand how native derived tables function, let's review the parameters in detail.

Proprietary + Confidential

Key parameters: explore_source

```
view: order_details {
  derived_table: {
    explore_source: order_items {
      column: order_id {
        field: order_items.order_id
   }
      column: user_id{}
      column: order_item_count{}
      column: total_revenue{}
    }
  }
}
```

Google Cloud

A key parameter for native derived tables is **`explore_source`**, which points to the Explore in your model that serves as the foundation for the derived table.

In other words, it's the **FROM** part of your SQL query and specifies any required joins.

Proprietary + Confidential

Key parameters: column

```
view: order_details {
  derived_table: {
    explore_source: order_items {
      column: order_id {
        field: order_items.order_id
   }
      column: user_id{}
      column: order_item_count{}
      column: total_revenue{}
    }
  }
}
```

Google Cloud

Another key parameter is **`column`**, which specifies an output column for the native derived table. In other words, it is a field that is being **SELECT**ed.

Sometimes, you will see a **`field`** parameter inside the curly braces {} for a column. If you have more than one field with the same name throughout the Explore, you can use the **`field`** parameter to tell Looker exactly which dimension or measure to use. If your field name is unique throughout the Explore, you can leave the curly braces {} empty.

This also means that you can use the **`field`** parameter to name new column differently from the underlying field name.

Proprietary + Confidential

Additional useful parameters

filters:
Used to apply filters to the derived table using the same syntax as a filtered measure.

bind_filters:
Used to apply templated filters to the native derived table.

derived_column:
Specify one or more new columns that don't exist in the Explore specified by the **explore_source** parameter.

expression_custom_filter:
Specify one or more custom filter expressions on an explore_source query.

Google Cloud

In addition to the key parameters for **explore_source** and **columns,** there are many other helpful parameters for native derived tables.

—————

First, **filters** can be used to apply filters to the derived table, similar to a filtered measure.

In essence, it adds a **WHERE** or **HAVING** clause to the query.

—————

If you need to add a dynamic or templated **WHERE** or **HAVING** clause to your derived table logic, you could use **bind_filters**.

This parameter will allow the native derived table to take on the filters that a user applies on a dimension or measure in the Explore query, and makes the filters “trickle down” to the derived table query itself.

—————

Now, imagine that you would like to use a field that does not currently exists in the Explore or model.

In a native derived table you can use the **derived_column** parameter to define a new column that does not yet exist in the Explore specified by the **explore_source** parameter.

—————

Traditional filters in Looker only let you specify what are basically key-value pairs and then it forces an **AND** conjunction between multiple inputs.

`expression_custom_filter`, on the other hand, lets you define nuanced logic with your own choice of **`AND`**/**`OR`** conjunctions and order of operations. Very powerful!

Proprietary + Confidential

Lesson summary

```
view: order_details {
  derived_table: {
    explore_source: order_items {
      column: order_id {
        field: order_items.order_id
   }
      column: user_id{}
      column: order_item_count{}
      column: total_revenue{}
    }
  }
}
```

Google Cloud

In summary, the key parameters for native derived tables are **`explore_source`** and **`column.`**

`explore_source` points to the Explore in your model that serves as the foundation for the derived table and is the basis of the **`FROM`** statement in your SQL query.

`column` specifies an output column for the native derived table and represents a field that is being **`SELECT`**ed in your SQL query.

In addition, there are other useful parameters for native derived tables, such as **`filters`** and **`derived_columns,`** which you can use to apply filters and define new columns that do not yet exist in your Explore.

Have fun customizing native derived tables in your organization’s Looker instance!

Proprietary + Confidential

Derived Tables

01 Introduction to Derived Tables

02 Types of Derived Tables

03 Example 5: Creating SQL Derived Tables with LookML

04 Example 6: Creating Native Derived Tables with LookML

05 Useful Parameters for Native Derived Tables

06 Persistent Derived Tables

07 Lab 2: Creating Derived Tables with LookML

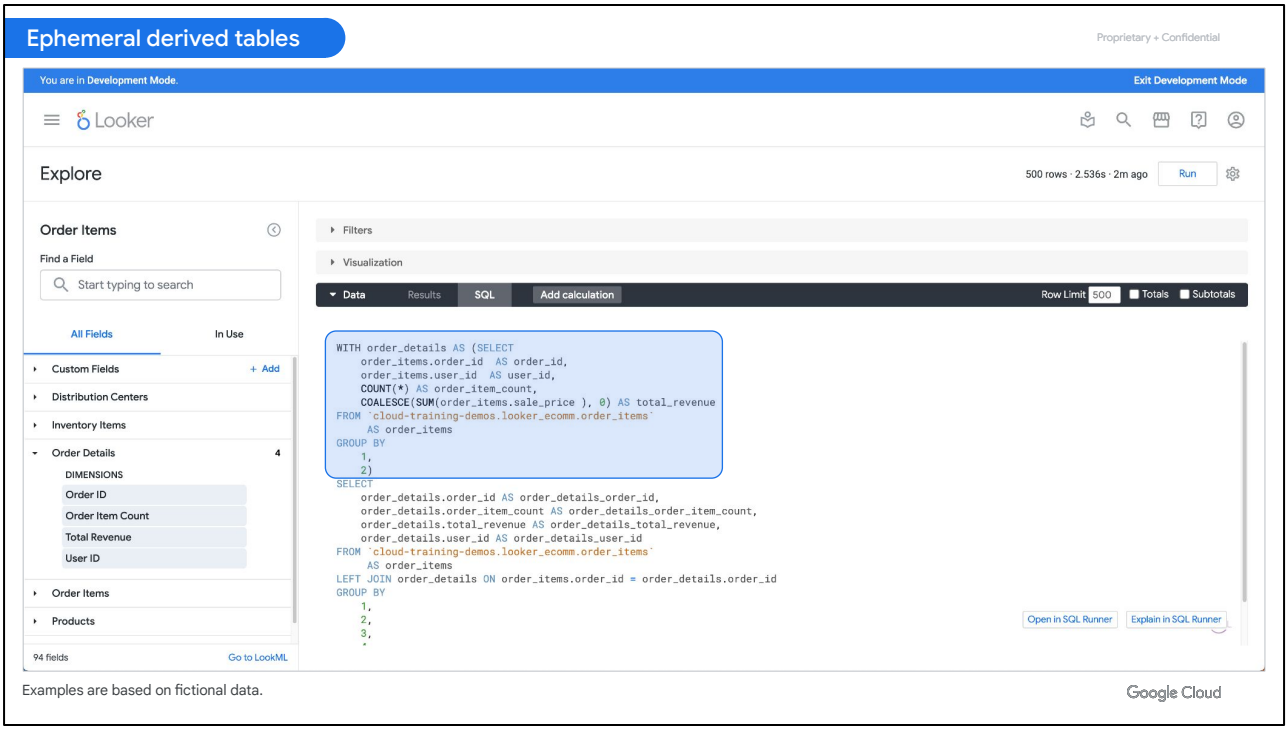


Google Cloud

In Looker, derived tables can be ephemeral, meaning that Looker builds them at run-time, or they can be persistent, meaning that they are written back to the connected database.

The benefit of persistent derived tables, or PDTs, is that they are ready to go when business users need them; while the main downside is that they take up storage space in your database, which may correlate to cost.

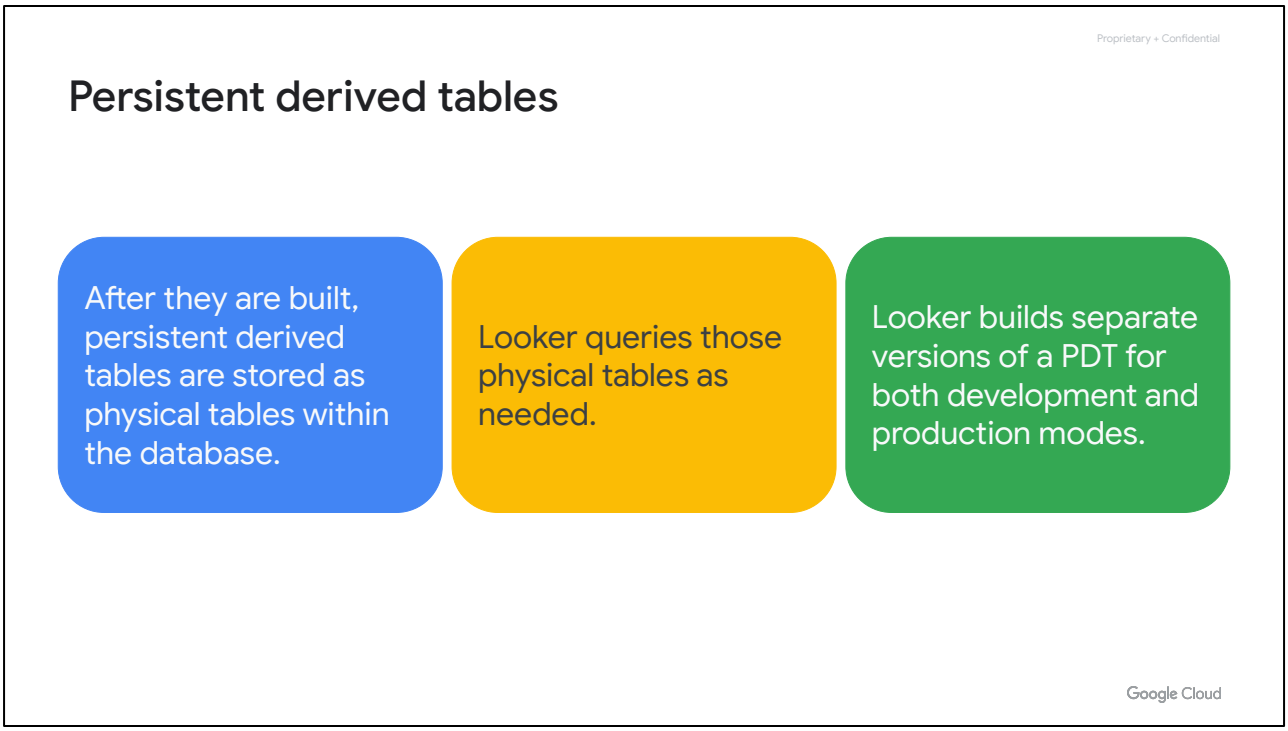
In this section, we will explore how PDTs are written back to and stored in the connected database in Looker.



In Looker, ephemeral derived tables build at runtime as temporary tables or common table expressions (CTEs) that are defined in the generated SQL with the following syntax:

`WITH derived_table_name AS (SELECT column_1 ...`

and so on. Note the subsequent SQL represents the Explore itself.



By contrast, a PDT will run a SQL **CREATE** statement in the underlying database to create a physical table.

Both SQL derived and native derived tables can be saved as PDTs.
—--
When a business user selects a field or filter from a PDT, the generated SQL will join to the PDT, just as it would to a regular table in the database.
—---
Another key point about PDTs is that Looker will build a version of the PDT specifically for your Git branch in Development Mode.

After you have deployed your code to production, or if the PDT already existed in production and you were modifying it, business users querying the PDT in production mode will be pointed to a separate version for production.

Proprietary + Confidential

SQL for persistent derived tables

Naming convention

```
-- use existing user_order_facts in teach_scratch.LR$KDYI2NQM4DW046XHR9XQH_user_order_facts
SELECT
    user_order_facts.lifetime_orders  AS "user_order_facts.lifetime_orders",
    COUNT(DISTINCT order_items.order_id ) AS "order_items.order_count"
FROM public.order_items  AS order_items
LEFT JOIN teach_scratch.LR$KDYI2NQM4DW046XHR9XQH_user_order_facts AS user_order_facts ON user_order_facts.user_id = order_items
    .user_id

GROUP BY 1
ORDER BY 2 DESC
LIMIT 500
```

Examples are based on fictional data.

Google Cloud

Let's review the generated SQL for a persistent derived table. First, notice the schema of the PDT, which is **teach_scratch** in this example.

We recommend naming the schema for PDTs something like "X_scratch", but you could call it anything you want. The important thing is to configure a schema dedicated to Looker PDTs in your database.

Typically, a database admin would create the schema, and then a Looker admin would specify the schema name in the connection settings of the Looker instance.

Proprietary + Confidential

SQL for persistent derived tables

Auto naming

```
-- use existing user_order_facts in teach_scratch.LR$KDYI2NQM4DW046XHR9XQH_user_order_facts
SELECT
    user_order_facts.lifetime_orders  AS "user_order_facts.lifetime_orders",
    COUNT(DISTINCT order_items.order_id ) AS "order_items.order_count"
FROM public.order_items  AS order_items
LEFT JOIN teach_scratch.LR$KDYI2NQM4DW046XHR9XQH_user_order_facts AS user_order_facts ON user_order_facts.user_id = order_items
    .user_id

GROUP BY 1
ORDER BY 2 DESC
LIMIT 500
```

Examples are based on fictional data.

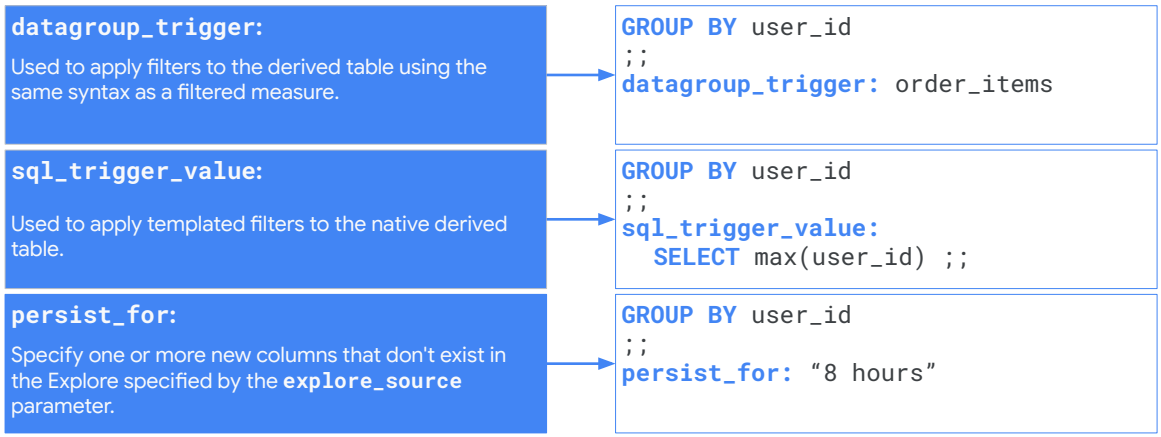
Google Cloud

Second, although the derived table is named **user_order_facts**, Looker will auto-generate the name for the PDT, so it can keep track of different versions of the PDT in development or production modes.

In the table name, a text string of a long jumble of letters and numbers will be prepended to the derived table name. PDT names always start with this hash, which encodes information such as the database connection and SQL that it uses.

Proprietary + Confidential

Persistent derived table parameters



Examples are based on fictional data.

Google Cloud

To persist a derived table, you must use at least one of the following parameters in the definition.

First, **`datagroup_trigger`** uses a datagroup, or caching policy configured in the model. As a best practice, we recommend using **`datagroup_trigger`** if datagroups are already defined in your model.

—------

Second, **`sql_trigger_value`** uses a pre-written **SELECT** statement that returns one value such as the maximum value of a user ID column.

Looker sends that **SELECT** statement to the database repeatedly, and when it discovers the result has changed, it takes this as a cue to rebuild the PDT.

—------

Last, **`persist_for`** instructs the PDT to be available for a set duration, such as “1 hour” or “4 hours”.

There are a few things to consider before choosing this option to persist your derived tables. First, because **`persist_for`** does not contain any rebuild logic, the PDT does not get updated at any time within the specified duration.

In addition, once time is up, the PDT is dropped and is not recreated until the next business user needs it for a query.

—------

As the primary benefit of PDTs is having data readily available to minimize query

runtimes, we recommend that you use **`persist_for`** in conjunction with **`sql_trigger_value`** to ensure that data updates within the duration, or simply use **`datagroup_trigger`** or **`sql_trigger_value`** on their own.

Proprietary + Confidential

Persistent derived tables

Cascading PDTs

```
view: user_facts {
  derived_table: {
    sql: SELECT
        order_items.user_id as user_id
        , COUNT(DISTINCT order_items.order_id) as lifetime_orders
        , SUM(order_items.sale_price) AS lifetime_revenue
        , MIN(NULLIF(order_items.created_at,0)) as first_order
        , MAX(NULLIF(order_items.created_at,0)) as latest_order
        , SUM(order_items.sale_price) AS order_value
      FROM order_items
      GROUP BY user_id
       ;;
    datagroup_trigger: ecommerce_case_study_default_datagroup
    sortkeys: ["user_id"]
    distribution: "user_id"
  }
```

```
view: cascading_derived_table{
  derived_table: {
    sql:
    SELECT
    lifetime_orders
    , AVG(lifetime_revenue) as average_revenue
    FROM ${user_facts.SQL_TABLE_NAME}
    GROUP BY lifetime_orders
    ;;
  }
```

Examples are based on fictional data.

Google Cloud

If you have a use case for it, you can also make SQL derived tables select **FROM** one or more other derived tables. This would create *cascading* derived tables due to dependencies between the tables.

Given that PDT names always start with a long hash, you can use **.SQL_TABLE_NAME** (in capital letters) as a substitution operator to refer to the view name. This enables Looker to plug in the physical table name at runtime.

Proprietary + Confidential

Persistent derived tables

Rebuild order

1

```
view: user_facts {
  derived_table: {
    sql: SELECT
        order_items.user_id as user_id
        , COUNT(DISTINCT order_items.order_id) as lifetime_orders
        , SUM(order_items.sale_price) AS lifetime_revenue
        , MIN(NULLIF(order_items.created_at,0)) as first_order
        , MAX(NULLIF(order_items.created_at,0)) as latest_order
        , SUM(order_items.sale_price) AS order_value
      FROM order_items
      GROUP BY user_id
      ;;
    datagroup_trigger: ecommerce_case_study_default_datagroup
    sortkeys: ["user_id"]
    distribution: "user_id"
  }
```

2

```
view: cascading_derived_table {
  derived_table: {
    sql:
    SELECT
    lifetime_orders
    , AVG(lifetime_revenue) as average_revenue
    FROM ${user_facts.SQL_TABLE_NAME}
    GROUP BY lifetime_orders
    ;;
  }
```

Examples are based on fictional data.

Google Cloud

If you persist the cascading derived tables, then it is important to ensure they rebuild in the correct sequence, so the dependent PDTs rebuild with the most recent data.

You cannot explicitly control the sequence that Looker users to rebuild PDTs.

Now, one way you could try to make all your PDTs build in the right order is to stagger the **`sql_trigger_value`**, based on the typical build time of the preceding derived table. As you might imagine, this isn't foolproof.

This is another reason why **`datagroup_trigger`** is better than **`sql_trigger_value`** for applying persistence. If all your cascading PDTs use the same **`datagroup_trigger`**, then Looker will actually be able to detect the dependencies and rebuild the PDTs in the guaranteed correct sequence.

Proprietary + Confidential

Indexing persistent derived tables



https://docs.looker.com/reference/view-reference

Google Cloud

Last, we also recommend that you add indexing to your PDTs.

Indexing is like a card catalog in a library, where you could look up the location of a book to quickly find it among the shelves. Similarly, indexes help databases process queries more quickly.

Looker accepts the indexing parameters for most database dialects. We suggest discussing indexing of PDTs with a data engineer or database administrator at your company to determine the best option for your use case.

More information on these parameters can be found in the Derived Table Parameters section of Looker's View parameters documentation page.

Proprietary + Confidential

Lesson summary

After they are built, persistent derived tables are stored as physical tables within the database.

Looker queries those physical tables as needed.

Looker builds separate versions of a PDT for both development and production modes.

Google Cloud

In summary, after PDTs are written back to your databases, they are stored as physical tables that be queried and joined just like any other table in your database.

Looker builds a separate version of the PDT in development and production modes to ensure that you can create and test PDTs without impacting the production environment until you are ready to merge your updates.

Be sure to create a schema specifically for Looker PDTs. Within this schema, Looker auto-generates the name for the PDT, so it can keep track of different versions and encode information such as the database connection and SQL that it uses.

With all of these features, Looker makes it very easy to create PDTs and leverage their great benefits. We hope that you give them a try in your Looker instance!

Proprietary + Confidential

Derived Tables

01 Introduction to Derived Tables

02 Types of Derived Tables

03 Example 5: Creating SQL Derived Tables with LookML

04 Example 6: Creating Native Derived Tables with LookML

05 Useful Parameters for Native Derived Tables

06 Persistent Derived Tables

07 Lab 2: Creating Derived Tables with LookML



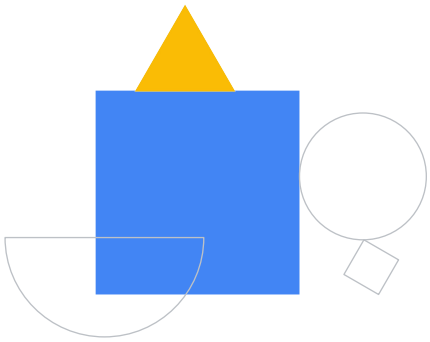
Google Cloud

Now that we have examined how to define both SQL derived and native derived tables, it's time for you to get some hands-on practice with creating derived tables using LookML.

Proprietary + Confidential

Lab Intro

Creating Derived Tables with LookML



Google Cloud

In this lab, you will define both SQL derived and native derived tables in an existing LookML project to address business users' questions.

You will also test your LookML changes by using the Explore interface to query the derived tables that you have defined in the modified LookML project.

Hope you enjoy the lab!

Proprietary + Confidential

Summary

Introduction to Derived Tables

Types of derived tables

Example: Creating SQL derived tables with LookML

Example: Creating native derived tables with LookML

Useful Parameters for Native Derived Tables

Persistent Derived Tables

Lab: Creating derived tables with LookML

Google Cloud

In this module, we began with an introduction to how derived tables work in Looker and reviewed the two types of derived tables that you can create using LookML.

Next, we walked through an example of creating each type of derived table in Looker: SQL derived tables and native derived tables.

We also reviewed some additional parameters for native derived tables that you can use to further customize these tables.

Then, we examined how to create persistent derived tables, so that you can write back and store your derived tables in the underlying database.

Last, you got some hands-on experience with creating both SQL derived and native derived tables in Looker using Qwiklabs.

In the next module, we will focus on how you can leverage caching in Looker to reduce database load and query optimize performance.

Hope you’re excited to keep going!

