DUMPSBOSS.

Google LookML Developer

Google Google-LookML-Developer

Version Demo

Total Demo Questions: 7

Total Premium Questions: 70

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

QUESTION NO: 1

A developer creates a LookML project that contains model files and view files in separate folders. The developer wants the sales model to make all view files in the `views` folder available, while excluding view files in the `archive` folder.

Which two actions should the developer take? (Choose two.)

A. Add `include: "/views/*.view.lkml"` to the sales model.

B. Do not add an include pattern that matches files in the archive folder.

C. Add `connection: "views"` to the sales model.

D. Add `hidden: yes` to every archived view.

E. Create an Explore for each view file in the views folder.

ANSWER: A B

Explanation:

The developer should add an `include` statement to the sales model that references the view files in the `views` folder, such as `include: "/views/*.view.lkml"`. Model includes determine which LookML files are available when Looker validates and runs that model.

The developer should avoid including the `archive` folder in the model. Files that are present in a project are not automatically available to every model; a model must include the files that define the views and Explores it uses. Omitting the archived files prevents their views from becoming available through the sales model.

Moving files to a different folder without updating model includes can make required views inaccessible. A connection declaration does not determine file availability. See the model include parameter reference and modeling in LookML.

QUESTION NO: 2

A developer is building a native derived table with an `explore_source`. The table must contain monthly revenue by region and must be rebuilt each day.

Which two configurations should the developer include? (Choose two.)

A. Add column declarations for region, month, and revenue from the source Explore.

B. Add an interval_trigger with a daily interval.

C. Add sql_trigger_value to the explore_source definition.

D. Add persist_with at the model level to schedule the table rebuild.

E. Add a sql_table_name parameter to the native derived table.

ANSWER: A B

Explanation:

The native derived table should use `column` declarations to include the region dimension, the month timeframe, and the revenue measure from the source Explore. These columns define the query and grain of the derived table. The developer should also apply an `interval_trigger` with a daily interval so Looker rebuilds the persistent derived table daily.

An `explore_source` defines a native derived table from a Looker Explore query rather than handwritten SQL. The `sql_trigger_value` parameter is used with SQL-based derived tables, not with native derived tables. A model-level `persist_with` setting controls query-result caching and does not by itself schedule a native derived-table rebuild. See the explore_source reference and the derived_table parameter reference.

QUESTION NO: 3

A developer has added a new view file named `returns.view.lkml` to a LookML project. The developer creates an Explore that joins the returns view, but the LookML Validator reports that the view cannot be found.

What should the developer do?

A. Add an include statement for the returns view file to the model.

B. Add required_access_grants to the returns view.

C. Add the returns view as an extension of the base view.

D. Add a datagroup to the model.

ANSWER: A

Explanation:

The developer should add an include statement in the model file that matches the path to `returns.view.lkml`. A model can use only the view files that it includes. For example, if the view file is in the project's views directory, the model can include it with a statement such as `include: "/views/returns.view"`.

Defining a join does not automatically load a view file into the model. Once the view is included, its view definition is available to the model and can be referenced by the Explore join. See the model include parameter reference and LookML model parameters.

QUESTION NO: 4

After running the LookML Validator, a developer sees the following error message in the Looker development environment:

"Measures with Looker aggregations (sum, average, min, max, list types) may not reference other measures".

What could be causing this error?

A. A measure of type: count has a sql parameter defined.

B. A measure of type: sum adds up other measures in the sql parameter.

C. A measure of type: sum has a SUM function written in the sql parameter.

D. A measure of type: number has a SUM function written in the sql parameter.

ANSWER: B

Explanation:

A measure of type: sum adds up other measures in the sql parameter. is correct because a LookML measure whose type performs a Looker-managed aggregation, such as `sum`, must aggregate a row-level field or SQL expression rather than the result of another measure. Referencing another measure with LookML substitution syntax in the `sql` parameter creates an attempt to aggregate an already aggregated result. Looker validates this pattern because the inner measure may be calculated at a different aggregation level, making the resulting SQL semantics ambiguous or invalid. For example, defining a sum measure with SQL resembling `${other_measure} + ${another_measure}` causes this validation error when either referenced field is a measure. The appropriate pattern is to reference the underlying dimensions or raw column expressions in the aggregating measure's SQL definition. If the desired result truly is arithmetic based on aggregated measures, define it as a `number` measure, which is intended for calculations involving other measures and does not apply its own Looker aggregation. Google documents the supported measure types and their aggregation behavior in the LookML measure type reference, and describes field substitution rules in the SQL and referring to LookML objects documentation.

QUESTION NO: 5

Users report that the main dashboard has been slow to show results.

Which two options should the developer evaluate to improve dashboard performance? (Choose two.)

A. Number of databases used by dashboard elements

B. Number of queries used by the dashboard

C. Ratio of visualizations to text tiles

D. Format used to deliver these reports

E. Amount of data rendered for each query

ANSWER: B E

Explanation:

Dashboard performance is strongly affected by the **Number of queries used by the dashboard** and the **Amount of data rendered for each query**. Each query tile can generate one or more database queries. Increasing the number of queries increases database workload and can also create query queues when the connection's concurrent-query capacity is reached. Developers should evaluate whether tiles can be consolidated, unnecessary tiles removed, or cached results used where appropriate.

The amount of data returned and rendered also matters. Queries that return large result sets require more database processing, transfer more data to Looker, and require the browser to render more information. Limiting returned rows, selecting only needed fields, and designing visualizations around aggregated results reduces this work while retaining useful dashboard content. Looker recommends minimizing dashboard queries and avoiding unnecessarily large result sets as central practices for responsive dashboards. See Build performant Looker dashboards and Optimize query performance.

QUESTION NO: 6

A developer would like to add a new dimension of type: yesno for the enabled column in their users table. The column is of type: string in the database and returns yes and no values.

How should the developer define the yesno dimension?

A.
```
dimension: is_enabled {

type: yesno

sql: ${TABLE}.enabled IS NOT NULL ;;

}
```

B.

```
dimension: is_enabled {

type: yesno

sql: CASE WHEN ${TABLE}.enabled = ""yes"" then ""Yes"" ELSE ""No""
END;;

}
```

C.

```
dimension: is_enabled {

type: yesno

sql: ${TABLE}.enabled ;;

}
```

D.

```
dimension: is_enabled {

type: yesno

sql: ${TABLE}.enabled = ""yes"" ;;

}
```

A. dimension: enabled {
type: yesno
sql: ${TABLE}.enabled = 'yes' ;;
}

B. Option B

C. Option C

D. Option D

ANSWER: A

Explanation:

The correct definition is `dimension: enabled { type: yesno sql: ${TABLE}.enabled = 'yes' ;; }`. A LookML field with `type: yesno` must have SQL that evaluates to a Boolean condition, rather than simply returning the underlying string column. Comparing the `enabled` column to the string literal `'yes'` produces a true result when the database value is `yes` and a false result when it is `no`. Looker can then treat the field as a native yes/no dimension in the Explore interface, including presenting appropriate yes/no filter values and labels. The comparison belongs in the field's `sql` parameter and is terminated with LookML's required double-semicolon delimiter. This approach also keeps the database's existing string representation intact while exposing a correctly typed semantic field to Looker users. Google's LookML reference documents that `yesno` dimensions are based on SQL expressions that return true or false: LookML field type reference. The `sql` parameter syntax and use of `${TABLE}` for a view's underlying table are documented in the LookML sql parameter reference.

QUESTION NO: 7

A developer is building an e-commerce Explore with the following datasets: orders and users. The business user needs to be able to answer questions about sellers and buyers within the same Explore. Each order in the orders table reports a buyer and seller ID. The users table has the detailed information about the individual buyer and seller.

How should the Explore be defined to meet this requirement?

A.
```
explore: orders
join: buyers {
view_name: users
sql_on: ${orders.buyer_id} = ${buyers.id} ;;
relationship: many_to_one
}
join: sellers {
view_name: users
sql_on: ${orders.seller_id} = ${sellers.id} ;;
relationship: many_to_one
}
```

B.
```
explore: orders
join: users {
sql_on: ${orders.buyer_id} = ${users.id} AND ${orders.seller_id} = ${users.id} ;;
A relationship: many_to_one
}
```

C.
```
explore: orders
join: buyers {
from: users
sql_on: ${orders.buyer_id} = ${buyers.id} ;;
relationship: many_to_one
}
join: sellers {
from: users
sql_on: ${orders.seller_id} = ${sellers.id} ;;
relationship: many_to_one
}
```

D.
```
explore: orders
join: users {
sql_on: ${orders.buyer_id} = ${users.id} OR ${orders.seller_id} = ${users.id} ;;
relationship: many_to_one
}
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: B

Explanation:

The selected configuration is correct because the Explore must join the user data twice, once for the buyer role and once for the seller role. Although both joins ultimately use the same underlying `users` table or view, each joined instance needs a distinct join name (or a distinct aliased view) so Looker can expose separate buyer and seller fields in one query. Each join condition should connect the corresponding foreign key in `orders`—the buyer ID or seller ID—to the user ID. Since many orders can be associated with one user in either role, the joins should be modeled as `many_to_one` from orders to each user instance. This relationship declaration lets Looker understand the intended row cardinality and supports correct aggregate behavior when users add fields from either role. In LookML, a join can use `from:` to create a uniquely named instance of an existing view, avoiding the need to duplicate the user-table definition. This design allows a business user to select buyer attributes and seller attributes together while analyzing the same order records. See the LookML join parameter reference and Google's guidance on using the from parameter for aliased views.