DUMPSARENA

Looker LookML Developer Exam

Google LookML-Developer

Version Demo

Total Demo Questions: 10

Total Premium Questions: 105

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Data Modeling	18
Topic 2, LookML Development	86
Topic 3, Looker Administration	1
Total	105

QUESTION NO: 1

A developer is connecting a LookML project to a remote Git repository. The developer wants to track which users are committing code changes, creating pull requests, or deploying to production when the different Git commands are initiated from within Looker.

Which type of Git connection should be utilized to meet this business requirement?

A. A bare Git repository

B. Multiple account HTTPS

C. Single account HTTPS

D. SSH

ANSWER: B

Explanation:

Multiple account HTTPS is the appropriate connection type because it associates Git activity initiated in Looker with each individual developer's Git identity rather than a shared repository credential. With this setup, developers authenticate to the remote Git provider using their own accounts. As a result, commits and pull requests created from Looker can be attributed to the person who performed the action, providing the required traceability and a more useful audit history in the Git provider. This approach also supports teams in which users have different repository permissions, because access is evaluated using the authenticated user's Git account. Looker's Git connection configuration supports multiple-account HTTPS specifically for environments where developers need to use distinct Git credentials. The deployment workflow remains governed by Looker's project and deployment permissions, while the corresponding remote Git actions retain individual user attribution. Review Looker's guidance on configuring remote Git connections at Connect Looker to a remote Git repository and its version-control workflow documentation at Version control and deploying changes.

QUESTION NO: 2

After validating LookML code, a developer receives the following error message:

"Unknown or Inaccessible Field users.name"

What is causing this error?

A. There is a missing join.

B. The field is set to "hidden".

C. The join relationship is incorrect.

D. The field uses incorrect SQL syntax.

ANSWER: A

Explanation:

There is a missing join. The reference `users.name` identifies a field named `name` in the `users` view. For Looker to resolve that field while compiling an Explore, the `users` view must be available in that Explore's query context. Typically, this means the Explore needs a `join:` declaration that joins the `users` view, with an appropriate `sql_on` condition, or the field reference must be used in an Explore where that view is already joined. When LookML validation encounters a field reference from a view that is not available through the Explore and its joins, it cannot construct the field's SQL expression and reports it as unknown or inaccessible. Adding the required join makes the view and its fields addressable as `users.name` in the Explore. The join should be defined in the Explore file or in an included LookML file that contributes to that Explore, and its join condition should reflect the actual relationship between the base view and the users table. Google's LookML reference describes how `join` makes another view available to an Explore and how its SQL relationship is defined: Explore join parameter reference. See also Google's guidance on defining models and Explores: Creating a model in Looker.

QUESTION NO: 3

A developer needs to add an Explore built off of the orders view, which surfaces only completed orders. An orders Explore exists that contains all order information. Fields from the orders view arealso referenced in other existing views such as ${orders.fieldname}.

How should developer define a new Explore for completed orders and keep all field references working correctly?

A)

```
explore: completed_orders {
sql_always_where: ${orders.status} = "complete" ;;
view_name: orders
}
```

B)

```
explore: completed_orders {
sql_always_where: ${orders.status} = "complete" ;;
from: orders
}
```

C)

```
explore: completed_orders {
always_filter: {
A field: orders.status
A value: "complete"
}
from: orders
view_name: orders
}
```

D)

```
explore: completed_orders {
always_filter: {
A field: orders.status
A value: "complete"
}
from: completed_orders
view_name: orders
}
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: C

Explanation:

The correct approach is to define a separate `completed_orders` Explore with `from: orders` and apply a required SQL condition for completed records, such as `sql_always_where: ${orders.status} = 'completed' ;;`. The `from` parameter makes `orders` the base view for the newly named Explore without renaming the view itself. Consequently, Looker continues to recognize the base view as `orders`, so existing cross-view field references written as `${orders.fieldname}` remain valid. This also leaves the original, unfiltered `orders` Explore available for use cases requiring all orders.

Using `sql_always_where` is appropriate because the completed-order restriction must be included in every query generated from this Explore. Looker adds the expression to the SQL `WHERE` clause, including when users choose fields or filters that otherwise might not explicitly reference the status field. The result is a clearly named Explore that consistently returns only completed orders while preserving the established view namespace and its reusable field references. See Google Cloud's documentation for Explore `from` and Explore `sql_always_where`.

QUESTION NO: 4

A company stores data for multiple regions in a shared orders table. Each user has a user attribute named region_code. Users must be restricted to rows for their assigned region, and they must not be able to remove the restriction from an Explore.

Which two LookML approaches can enforce this requirement? (Choose two.)

A. Add an access_filter that maps the region_code user attribute to the orders.region_code dimension.

B. Add a sql_always_where condition that restricts orders.region_code using the region_code user attribute.

C. Add required_access_grants to the orders view.

D. Add an always_filter on orders.region_code.

E. Hide the region_code dimension from the Explore field picker.

ANSWER: A B

Explanation:

An `access_filter` can apply a mandatory row-level filter by matching a user attribute to a dimension in an Explore. Looker applies the filter to every query from the Explore for users who have an assigned user attribute value.

An Explore-level `sql_always_where` condition can also enforce a mandatory restriction when it uses the user attribute in its SQL expression. Unlike a user-selected Explore filter, users cannot remove a `sql_always_where` condition. The developer should ensure that the user attribute is configured securely and that the SQL expression correctly handles the expected attribute values.

See the access_filter parameter reference and the sql_always_where parameter reference.

QUESTION NO: 5

A developer wants users to choose whether a metric displays revenue or cost. The selected option must change the SQL used by a measure rather than only changing the measure label.

Which two LookML components should the developer use? (Choose two.)

A. A parameter with allowed_value entries for Revenue and Cost

B. Liquid conditional logic in the metric measure that evaluates the parameter value

C. A suggestions parameter on the metric measure

D. An always_filter on the metric measure

E. A value_format_name parameter on the metric measure

ANSWER: A B

Explanation:

A `parameter` with defined `allowed_value` entries provides the controlled list of metric choices in the Explore. This prevents users from supplying arbitrary values and gives the developer known values to evaluate in LookML.

The measure can use Liquid logic that evaluates the parameter value and returns the appropriate measure expression. For example, a `type: number` measure can use a Liquid conditional to reference revenue for one allowed value and cost for another. This changes the SQL generated for the selected metric.

See the parameter reference and Liquid variable reference documentation.

QUESTION NO: 6

A developer has several measures that should appear together under a Revenue section in the field picker. The same collection of measures must also be reused in a drill and in another LookML definition.

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

A. Define a named set containing the relevant measures.

B. Apply group_label: "Revenue" to the relevant measures.

C. Apply hidden: yes to the relevant measures.

D. Create a separate Explore for the Revenue measures.

E. Add the measures to an always_filter parameter.

ANSWER: A B

Explanation:

Define a named set containing the relevant measures. A `set` creates a reusable collection of fields that can be referenced elsewhere in LookML. This avoids repeating a long list of field names and makes future maintenance easier.

Apply `group_label: "Revenue"` to the relevant measures. The `group_label` parameter organizes fields in the Explore field picker under a common heading, improving discoverability for business users. A set alone does not control the field picker grouping, and a group label alone does not create a reusable field collection. See the set parameter reference and the group_label parameter reference.

QUESTION NO: 7

A developer wants to create a new Explore based on the order_items view. The developer creates an Explore in the ecommerce model file with the following definition:

explore: order_items {}

After saving and validations, the developer receives this LookML validator error:

Inaccessible view “inventory_items”, “inventory_items” is not accessible inexplore” “order_items”. Check for typos and missing joins in explore “order_items”.

What caused this error to appear?

A. A field in the order_items view references a field in the inventory_items view.

B. A field in the inventory_items view references a field in the order_items view.

C. There is an Explore named inventory_items which references the order_items view.

D. There is another Explore named order_items which references the inventory_items view.

ANSWER: A

Explanation:

A field in the order_items view references a field in the inventory_items view. When an Explore is declared as `explore: order_items {}`, its base view is available, but another view is available to that Explore only when it is brought into the Explore through an appropriate `join`. LookML fields can use a cross-view reference, such as a SQL expression containing `${inventory_items.some_field}`, but the referenced view must be accessible in the Explore's join graph. In this case, validation finds a reference from `order_items` to `inventory_items`, while no join makes `inventory_items` available in the `order_items` Explore. That produces the inaccessible-view validation message. The developer should identify the cross-view reference and add a valid join for `inventory_items` to the Explore, including the correct relationship and SQL join condition for the data model. Google's LookML documentation explains that joins add views to Explores and define how their records relate: Explore join parameter reference. The documentation on field references also describes using fields from another view in LookML expressions: LookML dimension parameter reference.

QUESTION NO: 8

A developer is configuring an incremental persistent derived table. The source table receives new records daily, and the table contains a date column that can be used to identify newly loaded rows.

Which two statements are true about incremental PDTs? (Choose two.)

A. The derived table should use increment_key to identify the incremental time field.

B. The derived table must use a persistence trigger, such as datagroup_trigger or sql_trigger_value.

C. The derived table must use persist_for as its only persistence setting.

D. The increment_key can be any measure that returns a numeric value.

E. The derived table cannot use increment_offset.

ANSWER: A B

Explanation:

An incremental PDT uses `increment_key` to identify the time-based field that Looker uses when appending new data. Looker uses this field to limit incremental rebuild queries to the relevant new time period instead of rebuilding the entire derived table.

An incremental PDT also requires a persistence trigger, such as `datagroup_trigger` or `sql_trigger_value`. The trigger determines when Looker should evaluate and build the PDT. The optional `increment_offset` parameter can be used when recently loaded source data may be updated after its initial arrival. See the increment_key reference and the derived tables documentation.

QUESTION NO: 9

A developer needs to show the elapsed number of days between a customer's signup date and first purchase date. Both fields are stored as timestamps.

Which LookML field type should the developer use?

A. A dimension_group with type: duration

B. A dimension_group with type: time

C. A dimension with type: tier

D. A dimension with type: yesno

ANSWER: A

Explanation:

A `dimension_group` with `type: duration` is designed to calculate the difference between two time or date expressions. The developer can specify the start and end SQL expressions and define the desired interval, such as days. Looker then exposes duration dimensions that can be used in Explores.

This is preferable to manually repeating database-specific date-difference SQL in multiple fields because the duration dimension group centralizes the calculation and supports consistent interval definitions.

See the dimension_group type: duration reference.

QUESTION NO: 10

A developer needs to make a new view available in an existing model. The view file is stored in the same LookML project as the model file, but it is not currently included by the model.

Which two actions are required before the developer can use the view in an Explore? (Choose two.)

A. Add an include statement in the model file for the view file.

B. Add the view as an Explore base view or join it into an existing Explore.

C. Add the view file to a model set.

D. Create a dashboard based on the view file.

E. Add a datagroup to the view definition.

ANSWER: A B

Explanation:

The model file must include the view file by using an `include` statement that matches the file path. Includes make LookML definitions from view files available to the model during validation and query generation.

The developer must add the view as the Explore base view or add it through a join in an existing Explore. Including a view file alone does not expose its fields to users. A view becomes queryable only when it is part of an Explore's base view or join graph. See the include parameter reference and the creating Explores documentation.